e-Informatica

software engineering journal

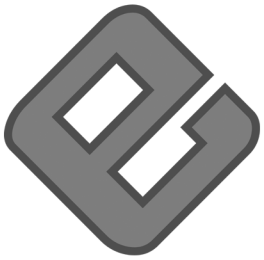
2014 volume 8 issue 1

e-Informatica

e-Informatica

software engineering journal

2014 volume 8 issue 1



e-Informatica

Wrocław University of Technology

Editors

Zbigniew Huzar (*Zbigniew.Huzar@pwr.edu.pl*)
Lech Madeyski (*Lech.Madeyski@pwr.edu.pl, http://madeyski.e-informatyka.pl/*)

Institute of Informatics
Wrocław University of Technology, 50-370 Wrocław, Poland

e-Informatica Software Engineering Journal
www.e-informatyka.pl/wiki/e-Informatica/, DOI: 10.5277/e-informatica

Editorial Office Manager: Wojciech Thomas

Typeset by Wojciech Myszka with the LaTeX 2_ε Documentation Preparation System

Printed in the camera ready form

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
http://www.oficyna.pwr.edu.pl;
e-mail: oficwyd@pwr.edu.pl; zamawianie.ksiazek@pwr.edu.pl

ISSN 1897-7979

Printed by EXPOL, P. Rybiński, J. Dąbek, sp. j., ul. Brzeska 4, 87–800 Włocławek
tel. 54 2323723, e-mail: sekretariat@expol.home.pl

Editorial Board

Contents

Editorial

It is a pleasure to present to our readers the eight volume of the e-Informatica Software Engineering Journal (EISEJ). It includes five papers carefully reviewed by Editorial Board members, as well as external reviewers, and then selected by the editors.

The first of the papers by Bartoszuk et al. (*On Visual Assessment of Software Quality*) concerns visual assessment of software quality and describes an innovative method of software analysis and visualization using graph-based approach. The benefits of this approach are shown through experimental evaluation using a proof-of-concept implementation – the Magnify tool.

The second paper is *The Use of Aspects to Simplify Concurrent Programming* by Michał Negacz and Bogumiła Hnatkowska. The presented results indicate that the use of aspects does not increase the complexity of a program while in some cases aspects can reduce the complexity.

The third paper by Rybiński et al. (*Generating Graphical User Interfaces from Precise Domain Specifications*) is about turning requirements into working systems in general and generating user interfaces directly from requirements models in particular. Syntax and semantics of a comprehensible yet precise domain specification language and the process of generating code for the user interface elements are presented.

The forth paper entitled *Supporting Analogy-based Effort Estimation with the Use of Ontologies* by Kowalska and Ochodek is on effort estimation of software development projects and proposes a new approach to model project data to support expert-supervised analogy-based effort estimation. The proposed approach can potentially help experts in estimating non-trivial tasks that are often underestimated.

The fifth paper by Canfora et al. is on detecting malicious JavaScripts (*Malicious JavaScript Detection by Features Extraction*) by feature extraction based on five features that capture different characteristics of a script. Mixing different types of features led to improvements empirically evaluated by the authors.

We look forward to receiving high quality contributions from researchers and practitioners in software engineering for the next issue of the journal. As always, we are interested to hear if you have any suggestions or comments; please send them to us at e-informatica (at) pwr.edu.pl.

Editors
Zbigniew Huzar
Lech Madeyski

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 software 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.

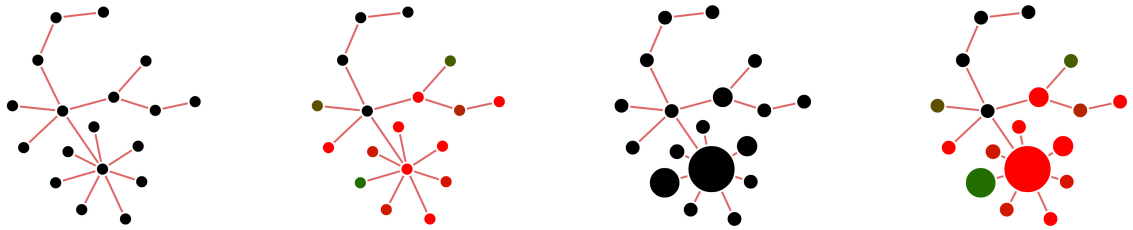
4.1. Visualization

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 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).

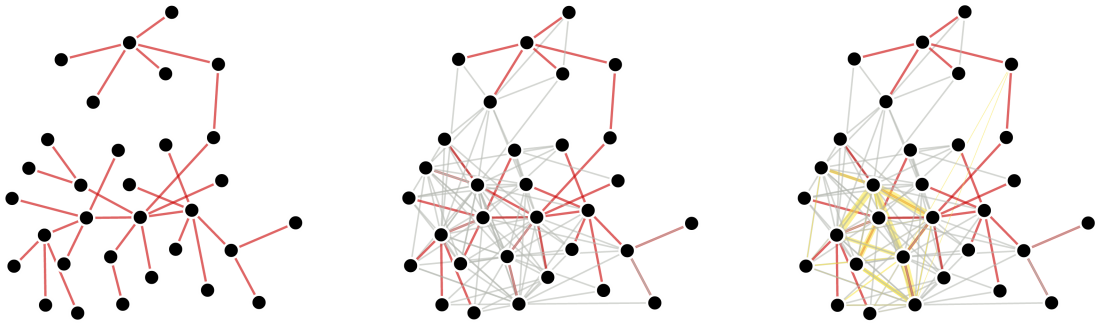
4.2. Analysis

In this section we assume a software project with the following properties. (1) Static data, like the



(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

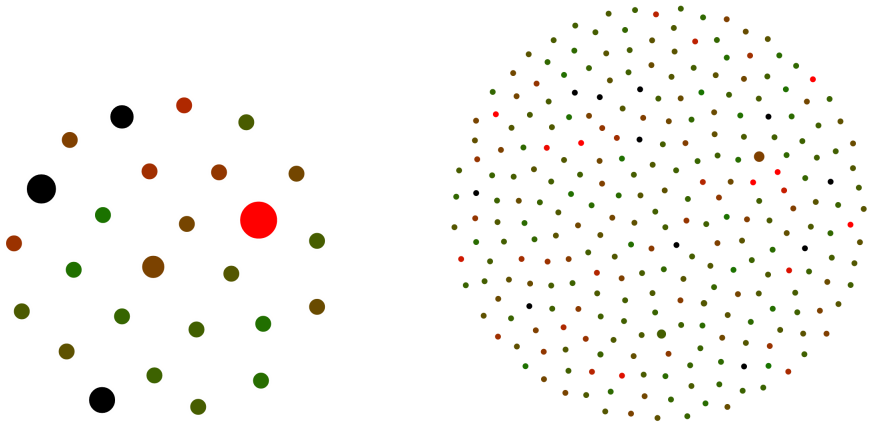
Figure 2. Relationships between software artifacts – static and dynamic

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.

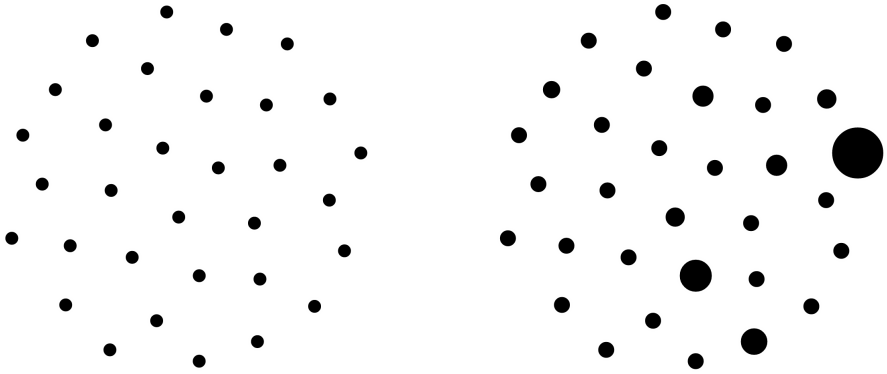
There can be multiple intuitions behind the definition of the **importance** of artifacts, e.g. the amount of work needed to adjust the rest of the system if this part of code gets changed. Consequently, there can be different algorithms that implement those intuitions with different semantics. In particular *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 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



(a) Aggregated artifacts (b) Fine grained artifacts

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



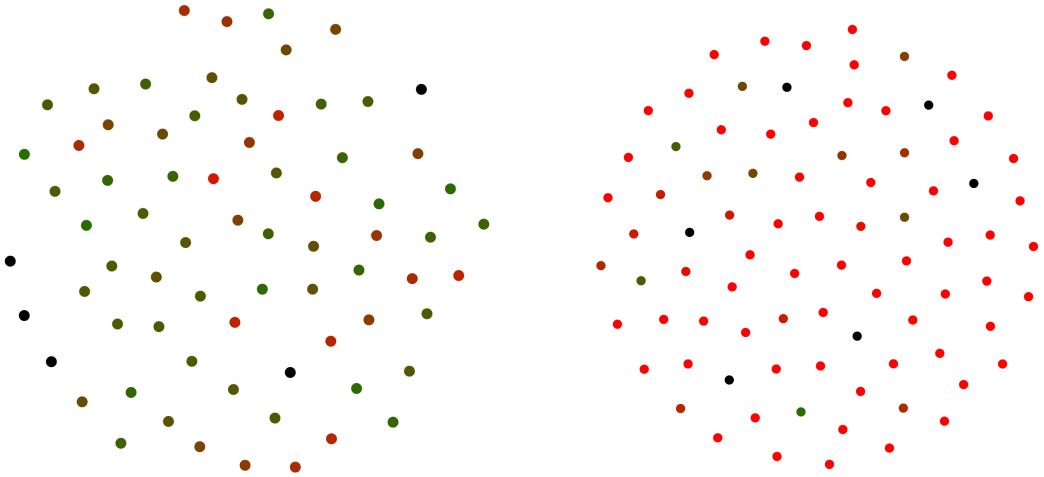
(a) No importance measure applied (b) PageRank as importance measure

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

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 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



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

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

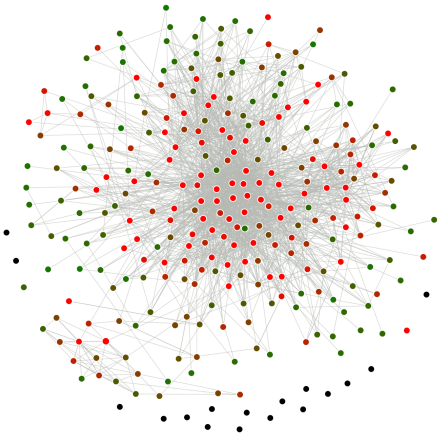


Figure 6. The complexity of relations may require applying model transformations

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 *Magnify*. 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, con-

tinuous 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 (see Fig. 7).

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

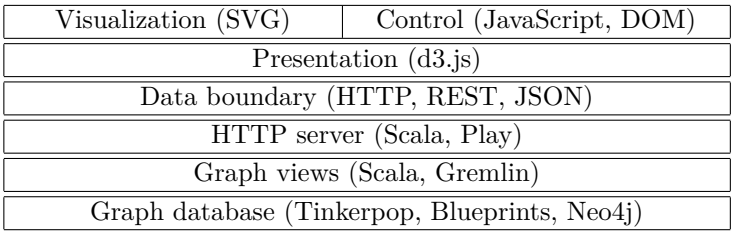


Figure 7. The architecture of the visualisation part of *Magnify*

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.*

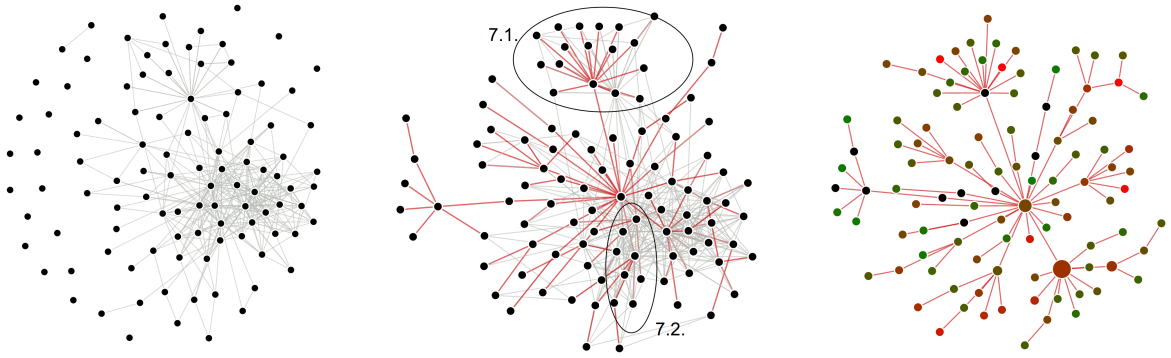
When we look at Figure 9, we can graphically divide the system into two parts. The bottom part has dense dependencies. The top part contains few aggregates of packages. The groups of packages on top have numerous internal edges, i.e. dependencies inside the aggregate. However, the dependencies between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



(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*

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 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.

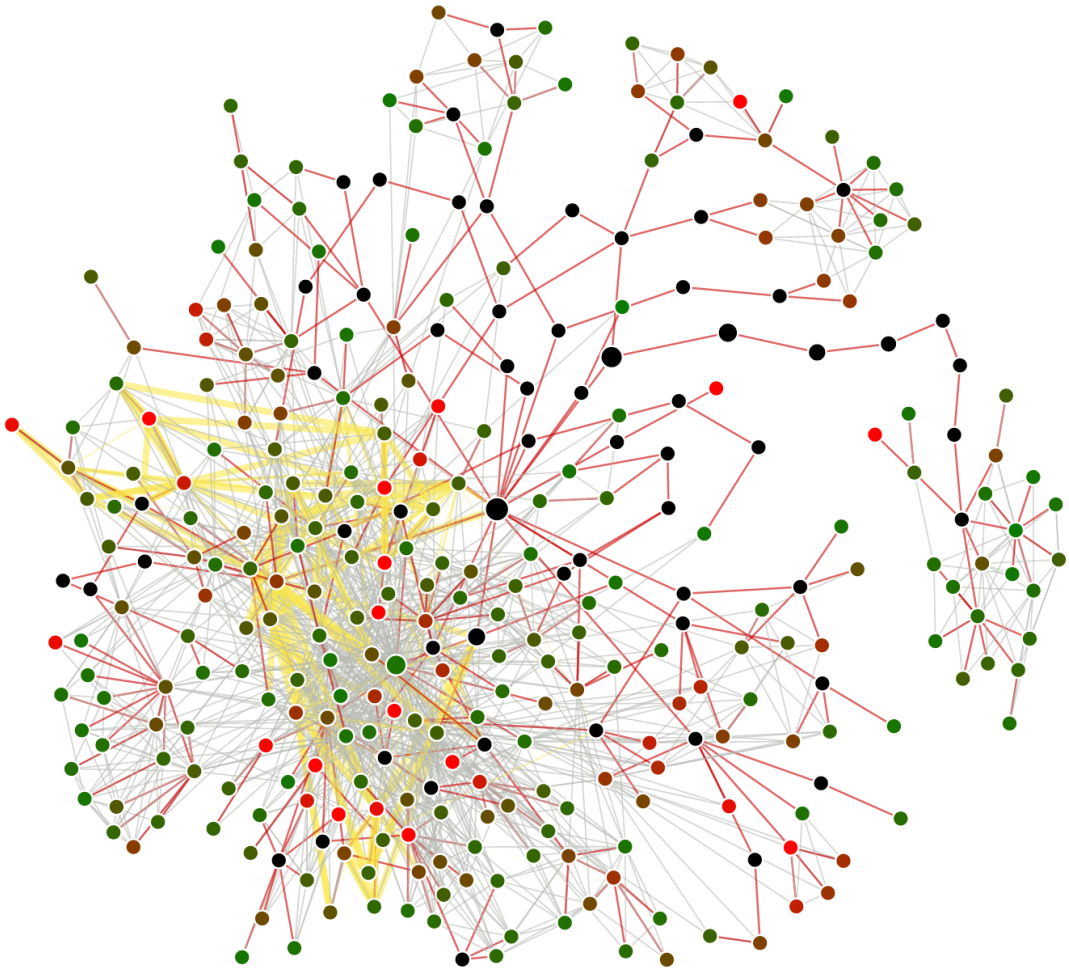


Figure 9. The visualisation of JLoXiM r2580 using *Magnify*

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.

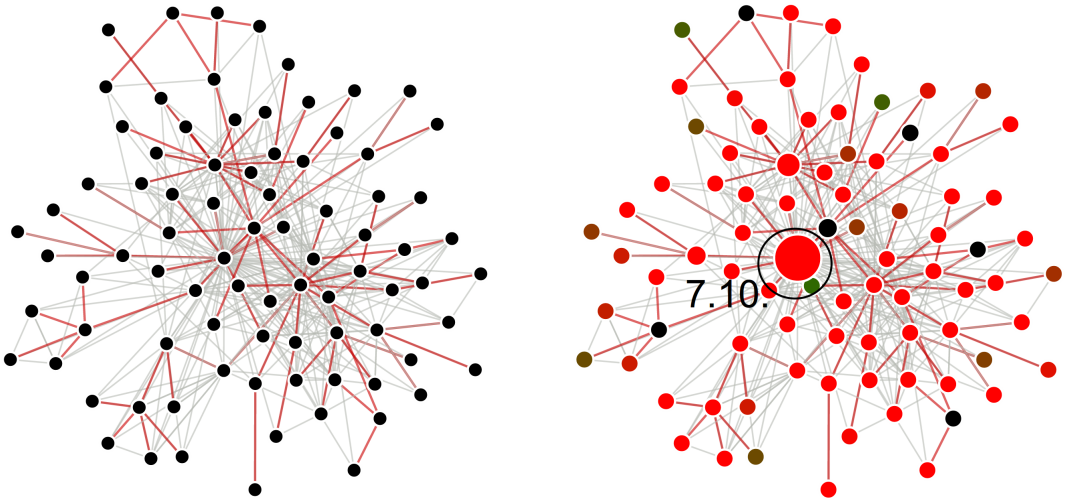
Observation 7.10. *`weka.core` might be a God Module.*

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 like string manipulation, statistics, comparisons and so on.

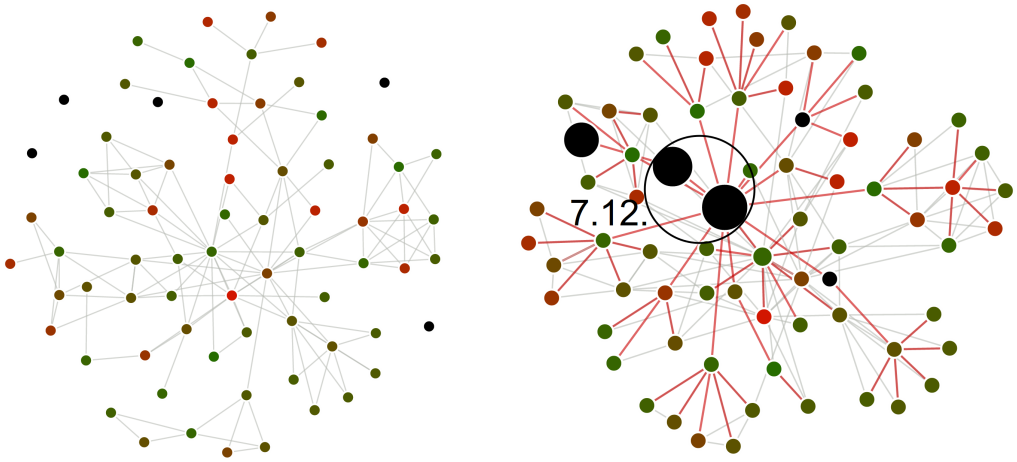
7.4. Spring Context 3.2.2

Spring is one of the most popular enterprise application frameworks in Java community. It pro-



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

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



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

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

vides 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 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.

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.*

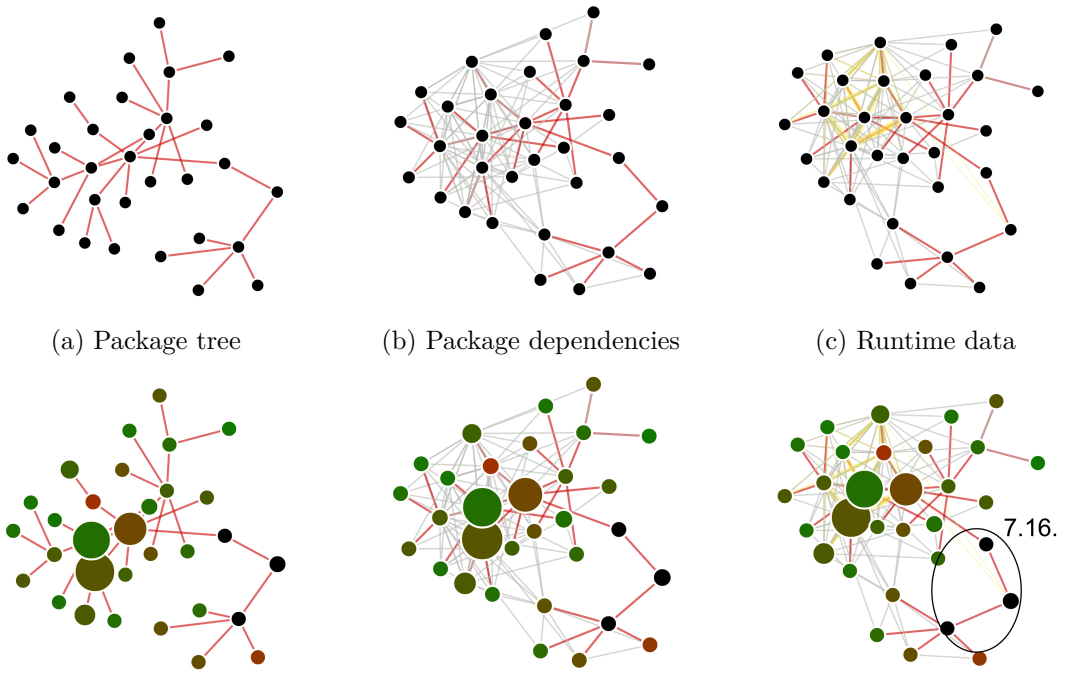
Figure 13b shows few packages in which classes are big on average. That means that overall complexity of the classes themselves is acceptable.

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

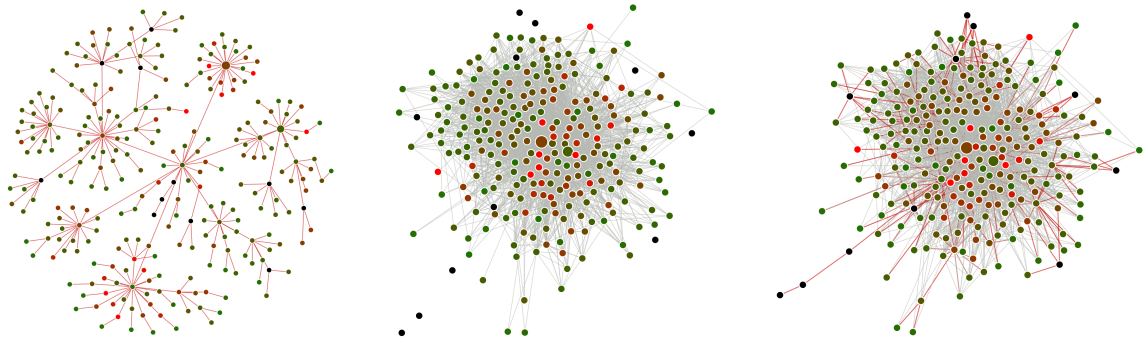
Play is a popular Scala and Java web framework. It is built on a lightweight, stateless and web friendly architecture. Play is heavily influenced by dynamic language web frameworks like



(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*



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

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

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.*

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

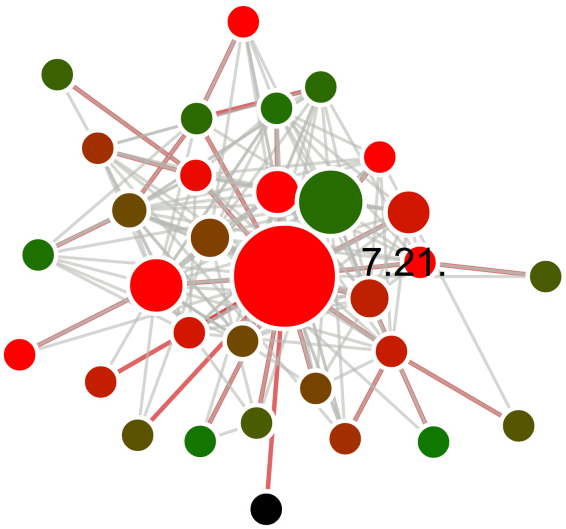


Figure 14. The visualization of Play 1.2.5 using *Magnify*

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.*

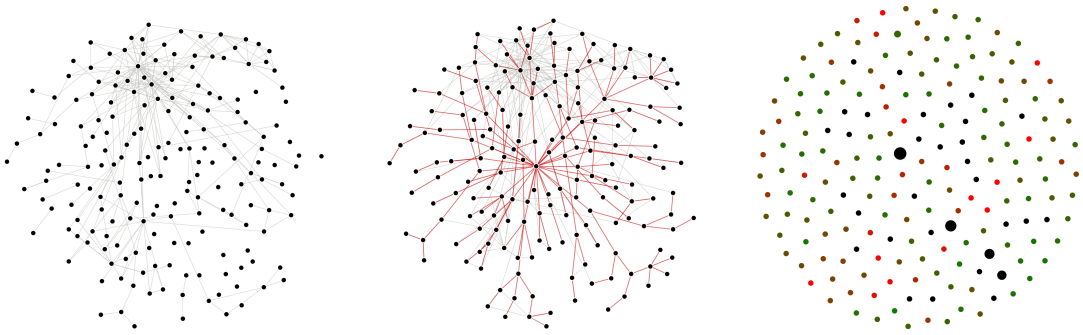
Figure 15 shows that overall code quality in Karaf is good. There are only few packages where average class size is alarming. The only refactoring we can suggest is to encapsulate subpackages of `org.apache.karaf.shell` which tend to spread a web of dependencies in the top part of the picture.

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 importance of the façade will be significantly higher



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

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

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, *LISP 1.5 Programmer's Manual*. MIT Press, 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 *Future of Software Engineering (FOSE), 2007*. 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. Schmerl, "Foundations and tools for end-user architecting," in *Large-Scale Complex IT Systems. Development, Operation and Management*, ser. Lecture Notes in Computer Science, R. Calinescu and D. Garlan, Eds. 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. Finkel-

stein, 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.

e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 27–37, DOI 10.5277/e-Inf140102

The Use of Aspects to Simplify Concurrent Programming

Michał Negacz*, Bogumiła Hnatkowska*
*Faculty of Computer Science and Management, Institute of Informatics, Wrocław University of Technology
michal@negacz.net, bogumila.hnatkowska@pwr.edu.pl

Abstract

Developers who create multi-threaded programs must pay attention to ensuring safe implementations that avoid problems and prevent introduction of a system in an inconsistent state. To achieve this objective programming languages offer more and more support for the programmer by syntactic structures and standard libraries. Despite these enhancements, multi-threaded programming is still generally considered to be difficult. The aim of our study was the analysis of existing aspect oriented solutions, which were designed to simplify concurrent programming, propose improvements to these solutions and examine influence of concurrent aspects on complexity of programs. Improved solutions were compared with existing by listing differing characteristics. Then we compared classical concurrent applications with their aspect oriented equivalents using metrics. Values of 2 metrics (from 7 considered) decreased after using aspect oriented solutions. Values of 2 other metrics decreased or remained at the same level. The rest behaved unstably depending on the problem. No metric reported increase of complexity in more than one aspect oriented version of program from set. Our results indicate that the use of aspects does not increase the complexity of a program and in some cases application of aspects can reduce it.

1. Introduction

Multi-core processors and supporting them systems are widely used at home. It is expected that the number of available cores will continue to grow in the next years [1].

The importance and number of programs that run concurrently has increased with the advance of technology. However, support for multi-core systems forces the use of concurrent programming techniques that are different from those known from single-threaded applications.

Aspect-Oriented Programming is a programming paradigm proposed by Gregor Kiczales. Its purpose is to enable and support a developer in separation of intersecting concerns and their modularization [2]. A costs of development and maintenance of concurrent programs can be reduced if a concurrent behavior is implemented in a modular manner, with minimum changes to an original source code.

The aim of this study is to analyze existing aspects, which solve concurrent programming problems, to propose improvements of existing mechanisms and the construction of a library that implements the existing solutions with the proposed improvements. The created aspect library is available at [3].

The library was used to implement typical programming problems and these implementations were compared with classical non-apsect solutions with the use of metrics. Then we answer the research question: *Does the use of aspects to concurrent programming reduce the complexity of application?*

The remainder of this paper is structured as follows. In section 2 we list the problems that a programmer may encounter when developing a concurrent application. Then, section 3 briefly describes previous research in the field of concurrent programming with aspects. In section 4 we present the use of aspects for concurrent

programming and show the introduced improvements. Section 5 presents the results of complexity comparison of the solutions with aspects and without them. After that, in section 6, we present the conclusions and future work.

2. Problems of Concurrent Programming

When designing concurrent programs, in addition to the traditional issues related to the design, one have to deal with a parallel part of an application. That is how tasks are divided between available resources and how to communicate and synchronize them with each other. Typical problems occurring in concurrent programming are:

- Code scattering [4–6].
- Code tangling [4–6].
- Deadlocks [7–9].
- Livelocks [7–9].
- Starvation [7, 9].
- Race conditions [9].
- Synchronizing access to shared resources, which consists of [9]:
  - Restriction of simultaneous access;
  - Visibility of data;
  - Publication of objects.

With these problems, reuse, debug or change the functionality of existing components become a difficult task [4,6,10]. Moreover, because of the concurrent code scattering between the various components, the understanding of the whole structure of concurrency in application is also tough [6]. Costs of developing and maintaining concurrent applications can be reduced if the concurrency is added in a modular manner, with the least possible changes to a code.

3. The Use of Aspects in Concurrent Programming

3.1. Asynchronous Method Execution

Laddad in his book [11] presented the *Worker object creation* pattern. In his solution an aspect is responsible for creating an anonymous class of type *Runnable*, which wraps an original method call. To use his solution, a programmer should define a pointcut in the aspect, which indicate the method for asynchronous execution. For each call the aspect creates an instance, which is passed to a new thread. As a result, instead of a direct synchronous execution, it is moved to a separate thread.

Cunha et al [4] proposed an improved solution. Unlike the previous, the presented mechanism allows threads, which are created in the aspect, to be assigned to a specific group of processes other than the current one. Programmer can optionally define a pointcut, where the current thread waits for spawned threads. It is also possible to define pointcuts for the interruption of thread. In addition, instead of explicitly declare a method as a pointcut, it is possible to give an asynchronous behavior only by marking it with an annotation.

Listing 1. Asynchronous method execution

```
1 @Asynchronous
2 void method() {
3     // instructions
4 }
```

Hohenstein and Gleim also presented their own version of an asynchronous method execution [10] (Listing 1). The authors recommended to perform concurrent code in the thread pool instead of creating a new thread for each execution. Concurrent executions are then limited to the upper limit of threads and do not reach the physical limits of the machine. When pool is used, one have to take into account the necessity of closing it. In the paper [10] authors proposed to use an additional annotation that indicates the place where the pool should be closed.

3.2. Asynchronous Method Execution which Returns a Result

A separate mechanism has been proposed for a concurrent execution of the methods that return a result. In the solution proposed by Cunha et al [4], there are two pointcuts. The first pointcut defines place where a calculation method is invoked, while the second indicates location

where a result is used. A thread that calls the method will be blocked at the second pointcut as long as the method does not return the result. The authors also mentioned a possibility of creating a fake object as the result, which represents it until it is not available.

Hohenstein et al [10] also created a separate aspect for methods that return a result. They noted that an exception thrown from a Future object requires unwrapping, which is an additional effort imposed on a programmer. They found that it could be possible to solve this problem with an aspect, which uses a generic type to represent the result (Listing 2). However, in the examples presented by them one can not see the way in which they achieve this unwrapping behavior. As in the previous case, also in this an annotation can be used.

Listing 2. Asynchronous method execution which return a result

```
1 @Asynchronous
2 Result<Object> method() {
3     Object object = // create an object
4     return object;
5 }
```

3.3. Asynchronous Execution of Recursive Methods

A solution proposed for asynchronous execution method with a result works well for recursive calls. Its major disadvantage is that it creates many threads – one for the root call and one for each of recursive method calls. a better solution gives Fork/Join Framework, which is a part of Java since version 7. Hohenstein et al [10] proposed to use this framework with an aspect. To simplify its application one can use an annotation. The aspect uses two pointcuts – the first captures the root call and the second recursive calls. In this case the generic type is also used to obtain the results of calculations.

3.4. Barrier

Cuncha et al [4] proposed an aspect oriented mechanism to implement a barrier. Aspect uses two pointcuts – both define methods where, respectively, the first blocks the thread before and the second after the method execution. The barrier can be added by marking the appropriate method with an annotation. The programmer should specify the number of threads that barrier will stop in parameter of the annotation. Optionally he may provide the name of the thread group, to which stopped threads belong.

3.5. Resource Synchronization

Cuncha et al [4] suggested two ways to simplify a resource synchronization at the method level. The first solution wraps intercepted method call into a Java synchronized block. The aspect provides two possibilities – the first uses a target object as the monitor, while the second uses an aspect object. The second resource synchronization solution allows a thread to only read or write to shared resources. This distinction allows for simultaneous multiple readings, but only one single write to the resource. It is possible to use an annotation for easier determination of synchronized methods.

Hohenstein and Gleim also studied the problem of resource synchronization [10]. They found that blocking can be dangerous and prone to errors due to forgetting to release a lock. An aspect can solve this problem and ensure the final release of any lock. In the proposed solution the following annotation is used *@RWProtect (reads = { "resourceA" }, writes = {"resourceB", "resourceC" }).* Parameters of this annotation are resource identifiers in the aspect. The *@RWProtect* annotation specifies resources to read and write in order to coordinate concurrent access. If different annotated methods reference to the same resource, their access is synchronized – simultaneous reading is allowed at the same time, but writing excludes other writings and readings. Locks at resources are always applied in a specific order to avoid deadlocks. However, in the proposed aspect, despite of use of non-blocking map, there is a race condition. In addition, in certain circumstances a thread starvation may appear, when the thread is waiting for a lock.

3.6. Conditions of Method Execution

Execution of some methods may depend on the state of an object. Cuncha et al [4] proposed *waiting guards* mechanism, which is based on an aspect. When the condition is not met, a thread is blocked until there is an action that changes the state of the object, which will trigger a condition reevaluation. Additionally, the reevaluation may occur after a defined timeout. The concrete aspect defines pointcuts, which indicate methods for which conditions are checked and a method that can change the state of the object, forcing the reevaluation of conditions.

3.7. Active Object

The active object pattern separates method call from its execution. It allows multiple threads to access data which is modeled as a single object. Traditional implementations of the pattern are divided into three layers. The first layer contains a client object, which makes a call, the second layer includes a mechanism to transfer the call to a target object and the third layer is the target active object running in a separate thread, which is still waiting for method calls [12]. The implementation of the active object in an aspect way [4] moves the second and the third layer to aspects. In addition, this solution makes participating classes unaware of their roles in the pattern. To give an object the behavior of the active object one should use specified annotation.

4. Proposed Solution

4.1. Asynchronous Method Execution

To perform an asynchronous method execution, a programmer should mark it with the annotation *@Asynchronous*. By default, the method is performed in a thread pool created by *Executors.newCachedThreadPool()*. All method calls marked with this annotation will be executed in one common pool shared for the entire program. Methods that are annotated with the optional parameter *standalone = true* are executed in their own, single threaded, private pool that is immediately closed after the call. The common thread pool can be controlled by the annotation *@Startup*. The pool is created before calling the method marked with this annotation.

Annotation attributes which can be modified are:

- threadPool: ThreadPool – type of pool:
  - FIXED – pool with a fixed number of threads coming from the method *Executors.newFixedThreadPool(...)*. Number of threads is taken from the parameter maxThreads.
  - CACHED – pool with a dynamic number of threads coming from the method *Executors.newCachedThreadPool()*.
  - CUSTOM – pool with the characteristics defined by a programmer.
- maxThreads: int – the number of threads for FIXED type pool and maximum number of threads for CUSTOM type pool. If not specified, it is assumed to be a maximum value from the set {1, the number of available processors - 1}.
- coreThread: int – the working number of threads for CUSTOM type pool. If not specified, the default value is calculated from the formula 1.
- timeout: int – time after an unused thread is killed. Attribute is used exclusively by the CUSTOM type pool and it is measured in seconds. The default value is 60 seconds.
- shutdownAfterMainMethod: boolean – attribute specifies whether to automatically close the pool after leaving a main method of a program.

$$\text{coreThread} = \text{maxThreads}/3 + 1 \qquad (1)$$

The annotation *@Shutdown* is used for closing the common thread pool. After completing marked by this annotation method, the pool will not accept new tasks. The attribute *now = true* results in an immediate closing the pool, calls waiting in a queue will not be executed.

If the method declares an opportunity to throw controlled exceptions, they are softened by an aspect. This facility is dictated by a lack of an exception handling capabilities, which will be thrown in a separate thread. The code placed

in the *catch* part of the *try {} catch {}* structure would be unreachable (Listings 3 and 4).

Listing 3. Example of an unreachable code

```
@Asynchronous
void method() throws Exception {
    // ...
}

void callMethod() {
    try {
        method();
    } catch(Exception e) {
        // this code cannot be reached
    }
}
```

To specify where asynchronous method should join to a calling thread, methods can be annotated with *@JoinBefore* or *@JoinAfter* annotations.

Listing 4. Asynchronous method execution in a pool

```
@Startup(threadPool = ThreadPool.FIXED,
    maxThreads = 3,
    shutdownAfterMainMethod = true)
@Asynchronous
void method() throws Exception {
    // instructions, which we want
    // to call asynchronously
}

void callMethod() {
    method(); // there is no need for
              // handling thrown
              // exception
}
```

Table 1 compares features of previous aspect oriented solutions with our proposal.

4.2. Asynchronous Method Execution which Return a Result

The proposed aspect oriented solution considers two cases. The first case are methods that return an object type, which is not final. As in the case of methods that do not return a result, it is sufficient to mark a method with the annotation *@Asynchronous*. This method will immediately return automatically created *Proxy* object (Listing 5). Any call to a method on this object is delegated to the correct result and if it is not yet available, an execution is blocked until it is available. The second case is a situation where the return type is final. In this case a change in a structure of a program is needed. The function result should be wrapped with a generic type. Methods marked with the *@Asynchronous* annotation execute in the same thread pool that methods, which do not return a result. When, during the execution of the method, it will encounter an exceptional situation, an exception will be thrown in its original form when one tries to fetch the result. Aspects are not capable of dynamic declaring new exceptions to methods, so special property of generic type has been used to work around this limitation.

Listing 5. Example of an asynchronous method execution with a proxy as result

```
@Asynchronous
ExampleObject method() throws
                  ExampleException {
    // instructions, which we want
    // to call asynchronously
}

void callMethod() {
    try {
        ExampleObject proxy = method();
                    // asynchronous
                    // method call

        // instructions that you want
        // to do before the result
        // is available

        String something =
                  proxy.getSomething();
    } catch (ExampleException e) {
        // exception handling
    }
}
```

In Table 2 we presented comparison of features of previous aspect oriented solutions with our proposal.

4.3. Asynchronous Execution of Recursive Methods

A method may be performed recursively in three ways. Each of them requires marking the method with the annotation *@AsynchronousRecursively*. For each recursive call of the marked method

Table 1. Comparison of asynchronous method execution solutions

Property	Previous solution	Proposed solution
Usage of a thread pool	No [11], No [4], Yes [10]	Yes
The need to handle exceptions in a calling code	Yes	No

Table 2. Comparison of asynchronous method execution solutions which return a result

Property	Previous solution	Proposed solution
Usage of a thread pool	No	Yes
Usage of a proxy object	No	Yes
The need to unwrap exceptions	Yes [4], No [10]	No

an aspect creates a separate Fork/Join pool. It is possible to control the number of threads in the pool by the parameter *threads = 2*. The default number of threads is equal to the number of available processors.

The first possibility is to use generic object *Result*, which wraps an original result returned from the method. In order to better use the *Fork/Join* pool, in the second possibility, one can use the method *Result.scheduleWith(...)* proposed in [10]. Presented in this article method can take only one parameter. We have extended it to any number of parameters. It creates a fork for each result passed, but the result object, on which the method was called, is calculated in a current thread. However, a disadvantage of this solution is the need to change the program source code and adding the call which is not directly related to the application logic. Last, the third possibility is to use auto generated *Proxy* objects (Listing 6). This case allows one to make an application completely independent from the library.

Listing 6. An aspect oriented calculation of 10th Fibonacci number with a proxy object

```
1 void callMethod() {
2   Number proxy = fibonacci(10L);
3
4   // instructions that you want to
5   // do before the 10th fibonacci
6   // number is available
7
8   Long result = proxy.longValue();
9 }
```

```
10 @AsynchronousRecursively
11 Number fibonacci(Long n) {
12   if (n <= 1) {
13         return n;
14   } else {
15   return fibonacci(n - 1).longValue()
16        + fibonacci(n - 2).longValue();
17   }
18 }
```

Methods, which are performed recursively, use the same concept of exception handling as asynchronous methods that return result. This means that exceptions will be thrown unchanged when one tries to fetch a result.

Comparison of features of previous aspect oriented solutions with our proposal is presented in Table 3.

4.4. Barrier

To implement a barrier in an aspect oriented approach it is sufficient to mark a method with annotations *@BarrierBefore* or *@BarrierAfter* with the number of threads that the barrier stops. Barriers can also be named with the *name* parameter of the annotation. The default name of the barrier is *thisMethod*, which means that the barrier is assigned only to the annotated method. If for the one named barrier there are many annotations with different number of threads, then created barrier has a limit indicated in the first method, which is called in a flow of a program.

The problem may be a situation in which a method uses two or more barriers. Then it is

Table 3. Comparison of asynchronous execution of recursive methods solutions

Property	Previous solution	Proposed solution
The need to use specific methods (scheduleWith(...))	Yes	No
Usage of a proxy object	No	Yes

not known which barrier a thread has to consider first. In this case, a developer must determine an order by creating an artificial cascade of methods marked with barrier annotations (Listing 7).

Listing 7. A cascade of two methods with two barriers

```
@BarrierBefore(value = 3,
                name = "firstBarrier")
void method() {
    otherMethod();
}

@BarrierBefore(value = 3,
                name = "secondBarrier")
void otherMethod() {
    // instructions executed after
    // reaching "firstBarrier" and
    // "secondBarrier" by 3 threads
}
```

The solution does not include restrictions for groups of threads, because they are obsolete and it is not recommended to use them [7].

Table 4 compares barrier features of previous aspect oriented solutions with our proposal.

4.5. Resource Synchronization

To synchronize the whole method it is sufficient to mark it with the annotation *@Synchronize*. This will perform a synchronization on a lock assigned to a current object or an object of class *Class* in the case where the method is static. If one wants to synchronize the method using an another lock, then he should specify its identifier. To facilitate the connection of identifiers with resources, they should be marked by *@SharedResource* with a resource name (Listing 8), although for proper operation of a program it is not required. Resource identifiers are global to the program. If one wants to synchronize multiple resources, then their identifiers should be listed in the annotation. An aspect acquires locks always in the same order, so the order of identifiers in the annotation is not important. To set up locks, that distinguish between reading and writing to resources, identifiers should be specified in appropriate parameters. Default parameter assumes two types of synchronization.

Following keywords can be also used as a name of identifier in the *@Synchronize* annotation:

- this – a lock is assigned to a current object or an object of class *Class*. The behavior is analogous to precede a method with the word *synchronized*.
- this.name – a lock is assigned as in the case *this*, but also supplemented by the given *name*. The behavior can be understood as embracing the body of a method with the synchronized block with an object field as the argument.
- global – a global lock.

Listing 8. Resource synchronization

```
@SharedResource("sharedResource")
Object sharedResource;

@Synchronize(reads = "sharedResource")
void readResourceMethod() {
  // instructions that read resource
}

@Synchronize(writes = "sharedResource")
void writeResourceMethod() {
  // instructions that write
  // to resource
}
```

In Table 5 we presented comparison of features of previous aspect synchronization solutions with our proposal.

4.6. Conditions of Method Execution

In the proposed aspect oriented solution it is sufficient to mark a method with the annotation *@WaitUntilPreconditions*, then define precondition methods (with the annotation

Table 4. Comparison of barrier solutions

Property	Previous solution	Proposed solution
A possibility of sharing barrier between the methods and objects through its naming	No	Yes
A possibility to restrict a barrier only to a select group of threads	Yes	No

Table 5. Comparison of resource synchronization solutions

Property	Previous solution	Proposed solution
The ability to synchronize static methods	No	Yes
Mark resources with an identifying annotation	No	Yes
Keywords	No	Yes

@Precondition) and a method for re-evaluation of the conditions (the annotation *@EvaluatePreconditions*). A thread, which tries to execute the method marked with the annotation *@WaitUntilPreconditions* will be slept until all preconditions are not met. Evaluation of conditions can be automatically executed at a time interval set in the annotation parameter *@WaitUntilPreconditions(waitingTime = 1000)* in milliseconds or by calling from a program code the method marked with the annotation *@EvaluatePreconditions*. The precondition can be named and then the annotation *@WaitUntilPreconditions* could specify its identifier (Listing 9). If the method is annotated with no parameters, then by default is assumed that all of conditions marked with *@Precondition* must be met in order to execution. As preconditions are considered only methods annotated with *@Precondition* and which return boolean expression.

Listing 9. Method execution after fulfilling preconditions

```
1 private boolean state;
2 @WaitUntilPreconditions({
3         "onePrecondition",
4         "anotherPrecondition" })
5 public void method() {
6     // instructions executed after
7     // fulfilling the preconditions
8 }
9 @Precondition("onePrecondition")
10 public boolean precondition1() {
11     return state;
12 }
13
14 @Precondition("anotherPrecondition")
15 public boolean precondition2() {
16     return true;
17 }
18
19 @EvaluatePreconditions
20 public void notifyMethod() {
21     state = true;
22 }
```

Comparison of features of previous aspect oriented solutions with our proposal is presented in Table 6.

4.7. Active Object

In the proposed, aspect oriented solution to implement active object it is sufficient to mark a class with the annotation *@ActiveObject*. If the execution of a method has preconditions, one has to list its identifiers in the annotation *@GuardedBy* and to mark an appropriate predicate method with the annotation *@Precondition*. Marking the class with a parameter *terminateAfterMainMethod = true* will automatically close a thread of the active object after leaving a main method of a program.

Table 7 compares features of previous aspect oriented solutions with our proposal.

Table 6. Comparison of precondition solutions

Property	Previous solution	Proposed solution
Usage only a metadata from a program	No	Yes

Table 7. Comparison of active object solutions

Property	Previous solution	Proposed solution
Full implementation of the pattern (guard conditions)	No	Yes
Automatically termination of the active object	No	Yes

5. Comparison of the Applications

To be able to compare traditional and proposed solutions we found concurrent programs, which solve the classic problems:
- Dining philosophers problem [13],
- Producer – consumer problem [14],
- Calculation of the n-th Fibonacci number [15].

The next step was to write our own versions of the applications, which solve the above problems, using created aspects. After that we calculated selected metrics with the use of Chechstyle 5.7 and STAN 2.1.2 (see Table 8). For all applications following count metrics were calculated:
- LOC/NCSS – Lines Of Code/Non Commenting Source Statements (Checkstyle),
- NOF/NOA – Number Of Fields/Number Of Attributes (STAN),
- NOM – Number Of Methods (STAN),
- TLC – Top Level Classes (STAN).

And the complexity metrics:
- CC – Cyclomatic Complexity (Checkstyle),
- DAC – Data Abstarction Coupling (Checkstyle),
- CFOC – Class Fan Out Complexity (Checkstyle).

Count metrics (LOC/NCSS, NOF/NOA, NOM, TLC) were chosen because of their quantitative representation of the complexity and additive behavior. CC is a classic measure of the complexity of methods. For this metric values below 7 are considered to be acceptable, while above this value metric indicate the need for refactoring. The motivation for choice of DAC and CFOC metrics was, that they measure the complexity of individual classes, they are able to demonstrate differences in relationships of classes. Both are supported by tools. The Checkstyle tool in a default configuration allows 7 for DAC and 20 for CFOC. For all selected metrics, the smaller is the value, the less is the complexity of the examined class.

For each of three problems the number of lines of code and CFOC values are smaller in the aspect than in the traditional solution. For metrics NOF and DAC two aspect oriented programs are less complex than their traditional counterparts, while both versions of the third program are equally complex. For the remaining metrics (NOM, TLC, CC) in one problem the aspect oriented version is less complex, in the second problem the traditional and in third both versions are equally complex.

Aspect oriented versions are more complex in three cases. In the case where the number of methods is higher in the aspect than in the classical solution the increase is because of the need to create separate predicates method. In found classical dining philosophers solution, *Philosopher* class is nested and not considered by the metric, while in the aspect oriented version *Philosopher* is a separate class. In traditional, concurrent calculation of the n-th Fibonacci number, there are 4 more methods than in the aspect oriented solution. These methods mostly have CC metric value equal to 1, thus they are lowering the average. The maximum CC metric value is equal in both applications.

No metric had shown that in all three cases, the complexity of the aspect oriented solution was higher than a classic application. Also, there

Table 8. Comparison of programs using metrics

Metric	Pr1 (cla)	Pr1 (asp)	Pr1 (chg)	Pr2 (cla)	Pr2 (asp)	Pr2 (chg)	Pr3 (cla)	Pr3 (asp)	Pr3 (chg)
LOC	69	36	**−33**	86	63	**−23**	39	10	**−29**
NOF	6	4	**−2**	6	6	**0**	5	0	**−5**
NOM	7	7	**0**	11	13	**+2**	7	3	**−4**
TLC	1	2	**+1**	3	3	**0**	3	1	**−2**
CC	1.71	1.71	**0**	1.7	1.17	**−0.53**	1.33	1.5	**+0.17**
DAC	1	1	**0**	1.2	1	**−0.2**	1.33	0	**−1.33**
CFOC	2	1.5	**−0.5**	2.75	1.5	**−1.25**	1.33	1	**−0.33**

where
- Pr1 denotes the dining philosophers problem,
- Pr2 denotes the producer – consumer problem,
- Pr3 denotes the calculation of the n-th Fibonacci number,
- cla denotes a classic version of application (downloaded from the Internet),
- asp denotes an aspect oriented version of application (written by us with the use of aspect library),
- chg denotes a change between an aspect and a traditional version,
- Values of LOC, NOF, NOM, TLC were counted as a sum of metrics for all classes in the application,
- Values of CC, DAC, CFOC were counted as means of metrics for all classes in the application.

was no increase of complexity in more than one aspect oriented version of program per a metric.

In response to the research question, it can be concluded that the use of aspects to the simplification of concurrent programming does not increase complexity of a program and in some cases application of aspects can reduce it.

6. Conclusions

This paper presented an effort to develop an aspect library which simplifies concurrent programming. We improved the previously proposed solutions and presented new features. Then, we conducted research and have shown that the use of aspects may reduce the complexity of concurrent application.

In general, using aspects for the concurrent programming can improve selected maintainability sub-characteristics, i.e. analysability and modifiability. But maintainability also includes testability sub-characteristic. While the proposed aspects may help in understanding and implementing concurrent applications, an open problem is how to test a correctness of the solution.

It should be noted that our research was conducted on a small sample of programs. These programs are small applications and do not come from an industry. In addition, credibility of research is highly influenced by a quality of programs, both those created by the authors and those collected.

Therefore, in the future we are going to repeat the research with bigger number of programs. Moreover, we want explore the use of aspects in Proactor and Reactor concurrent patterns.

References

[1] B. Schauer, “Multicore processors–a necessity,” *ProQuest discovery guides*, 2008, pp. 1–14.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[3] M. Negacz, “concurrent aspects library,” 2014. [Online]. https://github.com/mnegacz/concurrent-aspects

[4] C. A. Cunha, J. a. L. Sobral, and M. P. Monteiro, “Reusable aspect-oriented implementations of concurrency patterns and mechanisms,” in *Proceedings of the 5th international conference on Aspect-oriented software development*, ser. AOSD’06. New York, NY, USA: ACM, 2006, pp. 134–145. [Online]. http://doi.acm.org/10.1145/1119655.1119674

[5] B. Harbulot and J. R. Gurd, “Using AspectJ to separate concerns in parallel scientific java code,” in *Proceedings of the 3rd international conference on Aspect-oriented software development*, ser. AOSD’04. New York,NY, USA: ACM,

2004, pp. 122–131. [Online]. http://doi.acm.org/10.1145/976270.976286

[6] J. L. Sobral, "Incrementally developing parallel applications with AspectJ," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 116–116. [Online]. http://dl.acm.org/citation.cfm?id=1898953.1899048

[7] J. Bloch, *Effective Java (2Nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[8] B. Eckel, *Thinking in Java*, 3rd ed. Prentice Hall Professional Technical Reference, 2006.

[9] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[10] U. D. Hohenstein and U. Gleim, "Using aspect-orientation to simplify concurrent programming," in *Proceedings of the tenth international conference on Aspect-oriented software development companion*, ser. AOSD'11. New York, NY, USA: ACM, 2011, pp. 29–40. [Online]. http://doi.acm.org/10.1145/1960314.1960324

[11] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications Co., 2003.

[12] R. G. Lavender and D. C. Schmidt, "Active object – an object behavioral pattern for concurrent programming," 1995.

[13] "Dining philosophers problem implmentation," 2013. [Online]. https://github.com/vonhessling/DiningPhilosophers

[14] D. Ryan, "Producer-consumer problem implementation," 2014. [Online]. https://github.com/dcryan/Producer-Consumer

[15] I. Tsagklis, "Java Fork/Join for Parallel Programming," 2011. [Online]. http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html

e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 39–52, DOI 10.5277/e-Inf140103

Generating Graphical User Interfaces from Precise Domain Specifications

Kamil Rybiński*, Norbert Jarzębowski*, Michał Śmiałek*, Wiktor Nowakowski*, Lucyna Skrzypek*, Piotr Łabęcki*

*Faculty of Electrical Engineering, Institute of Theory of Electrical Engineering, Measurement and Information Systems, Warsaw University of Technology

rybinskk@iem.pw.edu.pl, jarzebon@iem.pw.edu.pl, smialek@iem.pw.edu.pl, nowakoww@iem.pw.edu.pl, skrzypel@ee.pw.edu.pl, labeckip@ee.pw.edu.pl

Abstract

Turning requirements into working systems is the essence of software engineering. This paper proposes automation of one of the aspects of this vast problem: generating user interfaces directly from requirements models. It presents syntax and semantics of a comprehensible yet precise domain specification language. For this language, the paper presents the process of generating code for the user interface elements. This includes model transformation procedures to generate window initiation code and event handlers associated with these windows. The process is illustrated with an example based on an actual system developed using the presented approach.

1. Introduction

Requirements Engineering (RE) is a very distinct area of Software Engineering, because requirements define the problem space while other software artifacts operate in the solution space. Problems with requirements usually get amplified in later stages of software development leading to project failures [1]. This makes RE research especially important and challenging. When defining research directions for RE [2], we need to bear in mind that RE starts with ill-defined and often conflicting ideas and have to be handled by very varied groups: from domain experts and end-users to downstream developers. Challenges in this broad research field include finding ways to effectively elicit and formulate requirements and then turn them into other SE artifacts (design, code, tests, etc.).

A very promising approach to meet the RE challenges is Model-Driven Requirements Engineering (MDRE) [3]. MDRE is an emerging area of Model Driven Software Development (MDSD) [4, 5]. The basis for constructing an MDRE approach is a model-based language for expressing requirements. Probably the first such language is the Requirements Modeling Language proposed by Greespan et al [6, 7]. More recent languages include the Requirements Specification Language [8] and the Unified Requirements Modeling Language [9].

Building on the success of MDSD for design and implementation, Requirements Engineering can benefit from its techniques when properly balancing flexibility for capturing varied user needs with formal rigidity required for model transformations [10]. MDRE makes it possible that the requirements models define the real scope and all details of the envisioned software system, furthermore that the whole development [11, 12], testing [13] and documentation process will be driven and controlled by these requirements models as well.

ReDSeeDS [14] is a tool representing the MDRE approach by offering an open framework consisting of a scenario-driven development

method and domain vocabulary management. It implements the Requirements Specification Language (RSL) [8, 15] meta-model which uses constrained natural language sentences allowing the end-users to understand specifications presented as precise requirements models. Moreover, the precisely written platform-independent specification allows to translate it directly to code using one of the platform-specific transformations. The latest ReDSeeDS transformations generate not only the entire code of the application logic layer, and the method stubs for the model layer, but also a fully functional graphical user interface. This paper concentrates on this last topic. It presents an approach to generate fully functional code of the UI elements from precisely specified domain models, expressed in RSL.

The solution separates essential complexity connected with the domain description such as business rules and application logic, from the accidental (technological) complexity related with platform specific design and implementation [16]. The complexity of software development process using ReDSeeDS is significantly reduced from the user and developer point of view. Most of the accidental complexity is hidden within a special model transformation program, used to convert requirements specifications into code.

2. Related Work

Transition from requirements to design or implementation is considered as a difficult activity during software development. The complexity related to it can cause various errors mainly caused by ambiguity of requirements. To eliminate this ambiguity, some form of constrained language could be used. This would allow for providing semi-automated ways to generate analysis and design models or code artifacts. There exist various approaches to solve this problem.

Some work has focused on requirements in respect to their precision, both by defining new languages for this purpose [17], as well as properly using the existing ones [18]. The disadvantage of these approaches is that they do not propose further code generation. Some approaches can be distinguished by their use of use case scenarios for requirements specification. Giganto et al. [19] propose an algorithm to identify use case sentences from requirements specifications written in controlled natural language and as a result – automatically obtain classes from use cases. Whereas Mustafiz et al. [20] propose transformation rules for creating different types of behavioral diagrams from use case scenarios. Deeptimahanti et al. [21] suggest to analyze requirements specifications presented in natural language, using Natural Language Processing techniques and generate use case diagrams and class models.

Most of the solutions focused on code generation use graphical notations like UML [22] to specify static and dynamic aspects of systems. One example is the open source AndroMDA [23] code generation framework which supports the Model Driven Architecture [24] paradigm. As input, AndroMDA takes UML models from various CASE tools and provides generation of deployable applications and software components. In turn, textual specifications used as input, turn out to be often too formalized and thus difficult to understand by the end-users where the purpose is specifying requirements [25].

There are also number of solution focusing directly on graphical user interface generation. Many of them, however, use notations designed specifically for this purpose e.g. the one proposed by Falb et al. [26]. Some other solution use the existing software development notations. However, these notations operate at significantly lower level of abstraction than requirements specification, as e.g. propopsed by Janssen et al. [27]. There also exist a solution based on requirements scenarios [28], but it only generates graphical user interface mockups.

3. Syntax for Domain Elements in RSL

RSL is based on scenarios consisting of sentences that describe interactions between the actors and the system, written in constrained natural language. Scenarios are also grouped into

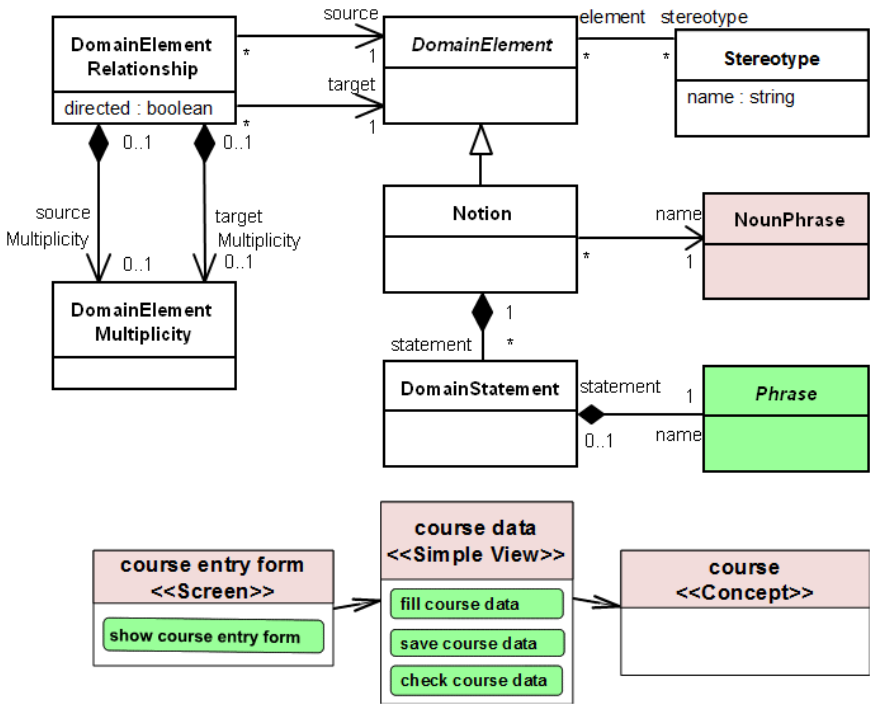


Figure 1. Abstract and concrete syntax for domain elements

RSL-specific "Use Cases", which are similar to the widely-known UML use cases. Additionally, every specification in RSL contains a domain model created from the notions used in use case scenarios. Each noun phrase in a scenario sentence should have a corresponding domain element.

The RSL's domain models are based on the metamodel, where its simplified version is presented in Figure 1 (upper part). The high-level elements in the RSL's metamodel can be compared to those of UML and to represent them we could use simply a profile of UML. However, RSL defines a very detailed notation for requirements representations which are precisely linked to domain elements. This unique feature of RSL allows for capturing precise models of the software system's essence [10].

In concrete syntax, domain elements resemble UML classes with associations, as presented in the lower part of Figure 1. For our considerations we will concentrate on "Domain Elements" of type "Notion". "Notions" represent business entities, buttons, windows and other elements that occur in the problem and system domain. Each "Notion" has a name represented as a "Noun Phrase" and contains "Domain Statements" with "Phrases" coming from use case scenarios. "Domain Elements" can be structured through specifying relationships and generalizations between them. Some "Notions" can be defined as attributes of other "Notions". Different types of notions are distinguished through their "Stereotypes".

An important part of the RSL meta-model is centred around "Phrases", which is presented in Figure 2. Phrases occur in scenario sentences and in domain statements. Phrases are divided into "Noun Phrases" and "Verb Phrases", where the second type can be further divided into "Simple Verb Phrases" and "Complex Verb Phrases". A "Noun Phrase" contains a "Noun" and an optional "Modifier", which can describe the "Noun" more precisely. A "Simple Verb Phrase" can be used as a sentence predicate and consists of a verb and a noun e.g. *adds* **selected student**. "Complex Verb Phrases" extend "Simple Verb Phrases" with a preposition and an additional "Noun Phrase" representing the indirect object. More detailed description of RSL syntax and its role in code generations can be found in [29] and [11].

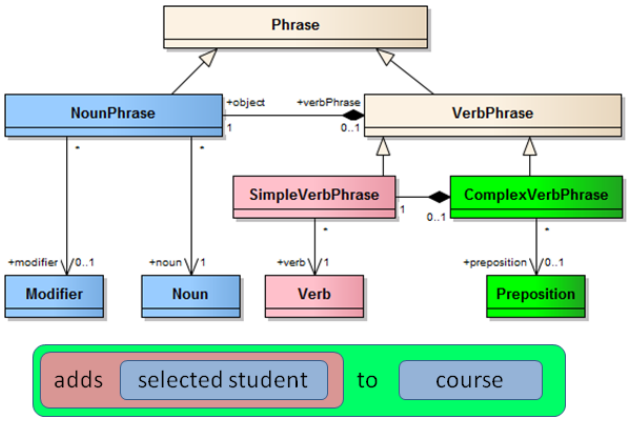


Figure 2. Abstract and concrete syntax for phrases

For the purpose of generating user interface code, the above meta-model of domain elements in RSL needed some additions. The fundamental distinction takes the form of a "notion type" which stems from the "Stereotype" attached to a "Notion". Possible notion types are: Concept, Attribute, Simple View, List View, Screen, Message and Trigger. These types of notions can be used as sentence objects in use case scenarios. For user interface generation it is also important to distinguish verbs associated with these notion types, as part of verb phrases. The verbs like 'show', 'refresh' and 'close' are associated in phrases with Screens and form so-called System-to-Actor sentences (e.g. "**System** *shows* **new student window**"). The verbs like 'select' are associated with Triggers and form so-called Actor-to-System sentences (e.g. "**User** *selects* **save button**").

4. Semantics for Domain Elements in RSL

The notion types described at the end of the previous section have specific meaning, which determines correct generation of the user interface code. A **Concept** is a representation of a business entity stored and processed by the system. It has no direct impact on generating user interface code, however it groups atomic data attributes for the purpose of their further processing. An **Attribute** describes one of the Concept's properties just like class attributes in UML. It should be noted that in RSL, Attributes are separate model entities. Each Attribute should be connected with at least one Concept. Moreover, each Attribute has its specific type which describes specific kind of data it represents like Text, Number, Date etc. A **Screen** is a representation of a window or a web-page depending on the transformation's target technology (for web-based technologies like JavaFX, Echo3, a Screen will be transformed into a web-page and for desktop technologies like Swing – into a window). A variant of Screen is a **Message** that represents a simple modal window used to show some error or confirmation message to the user. It causes generation of a proper pop-up message window.

A **Simple View** represents a set of data made available to the user during some interaction with the system. It can point to the attributes of many different Concepts, but should have defined a main Concept. If the Simple View is connected with a Screen, its attributes will be used as the basis for the window content. For every Attribute connected to a Simple View, an appropriate user interface widget will be created. Its type will depend on the Attribute type; for example it will be a text field for a text or a number Attribute or a check-box for a true/false attribute. Attributes typed as Date should generate not only a labeled text field, but also a button to call a calendar pop-up with the possibility to select the date.

A **List View** is similar to a Simple View, however presents many instances of given data

set in an ordered form. It causes generation of a table or a list. Attributes connected to a List View are used in the creation of its fields in the way analogical to that of a Simple View. Both Simple Views and List Views can be called Data Views. Data Views are usually connected to Screens. The direction of this relationship indicates type of access to data. Connection from a Screen to a Data View indicates that data will be entered, and connection from a Data View to a Screen indicates that some existing data will be presented. In case that there is no direction – both access types are assumed (modification of existing data).

A **Trigger** is a representation of a link, button or any other element of user interaction. Just like the Screen - it is a platform-independent term and its final form depends on a platform-specific transformation. Additionally, some Trigger invoking an operation, can be connected to a Data View that determines the data involved in that operation. There is no need need to define relations between Triggers and Screens. The transformation generates them based on scenarios assuring that there won't be any Trigger without functionality described in a scenario and there will be no Trigger described in a scenario which does not have a related Screen.

In addition to these domain elements, user interface elements are generated based on certain use case scenario configurations. This involves two types of sentences. A System-to-Actor sentence refers to a Screen or Message. It denotes an interaction of the system with an actor through displaying a window or message. These kind of sentences result in generating code that contains invocations of methods to display, refresh or close some user interface element. An Actor-to-System sentence refers to a Trigger. It denotes an interaction of an actor with the system through selecting some active element (button, hyperlink) in a window. An Actor-to-System sentence generates an appropriate event handler code. This code is generated in the code of the user interface element that was referred by a previous System-to-Actor sentence.

5. Code Generation Process

Figure 3 presents an overview of the software development process using the ReDSeeDS tool. The first step is to formulate and write requirements in RSL according to the rules described in the previous sections. The tool supports this process by offering a specialized scenario editor, automatic notion creation, notion editor with type assignments and much more.

The next step in the process is to execute a model transformation and generate detailed-design-level UML models with embedded code. The appropriate transformation program for generating the user interface elements was developed in the language MOLA (MOdel transformation LAnguage) [30]. MOLA is a graphical language which uses pattern matching algorithms on meta-model level to transform one model into another. In our case, this will be an RSL model translated into a UML class model with inserted code fragments. MOLA contains both declarative and imperative constructs. The declarative elements include rules which represents queries on the model, connected with indications which elements should be created or deleted. MOLA declarative rules are presented as gray rectangles with rounded corners, containing objects from the meta-model. Query elements have solid black borders, whereas create elements have thick red dashed borders. Imperative elements include control flows between the rules which are denoted by dashed arrows in a notation similar to UML's activity diagrams. Also, loops are possible, which are denoted by thick black boxes with rules that are to be iterated, contained inside them. The first rule inside a loop is the loop's iterator rule with one element being a loop-head and denoted with a thicker border. MOLA is also a procedural language, where procedure calls are denoted with special actions with procedure names and parameters. Procedure definitions declare these parameters as large arrow-shaped boxes. Procedures also declare variables as white rectangular boxes.

To present the idea of the user interface generation program, we provide three of its frag-

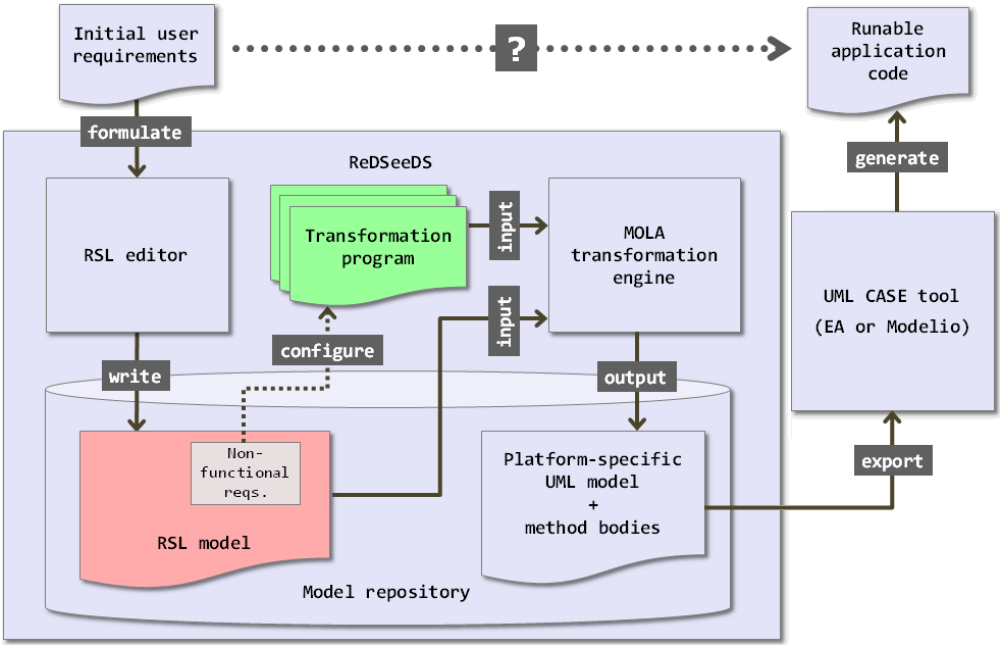


Figure 3. Process overview

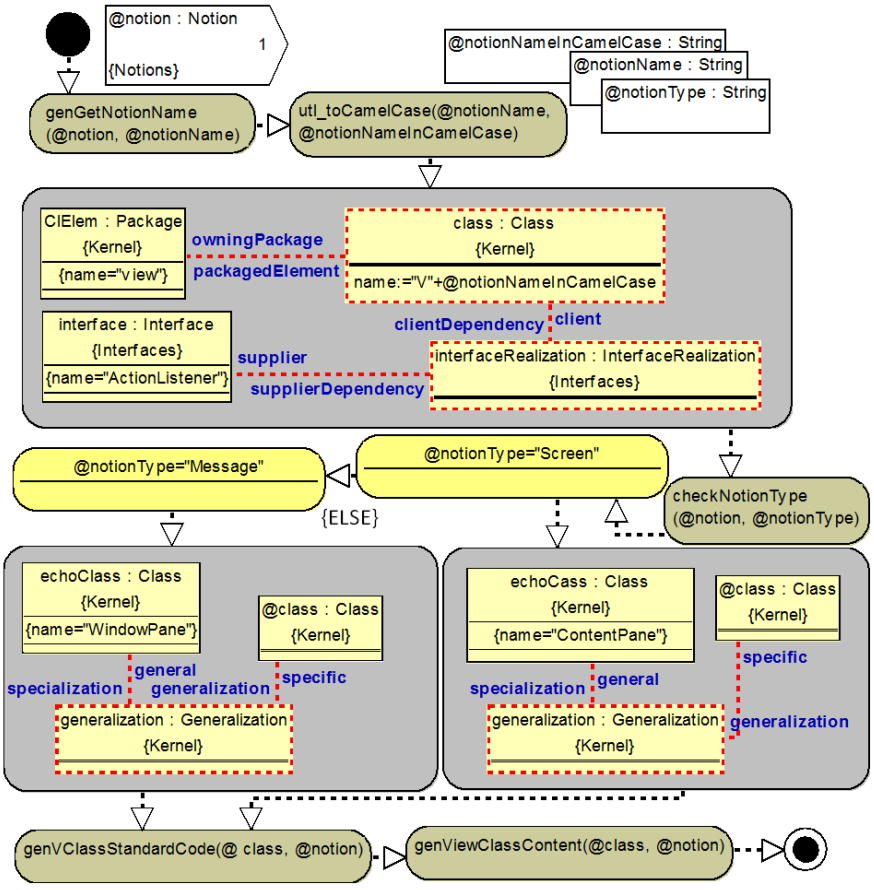


Figure 4. Procedure ('genViewClass') for generating classes from Screens and Messages

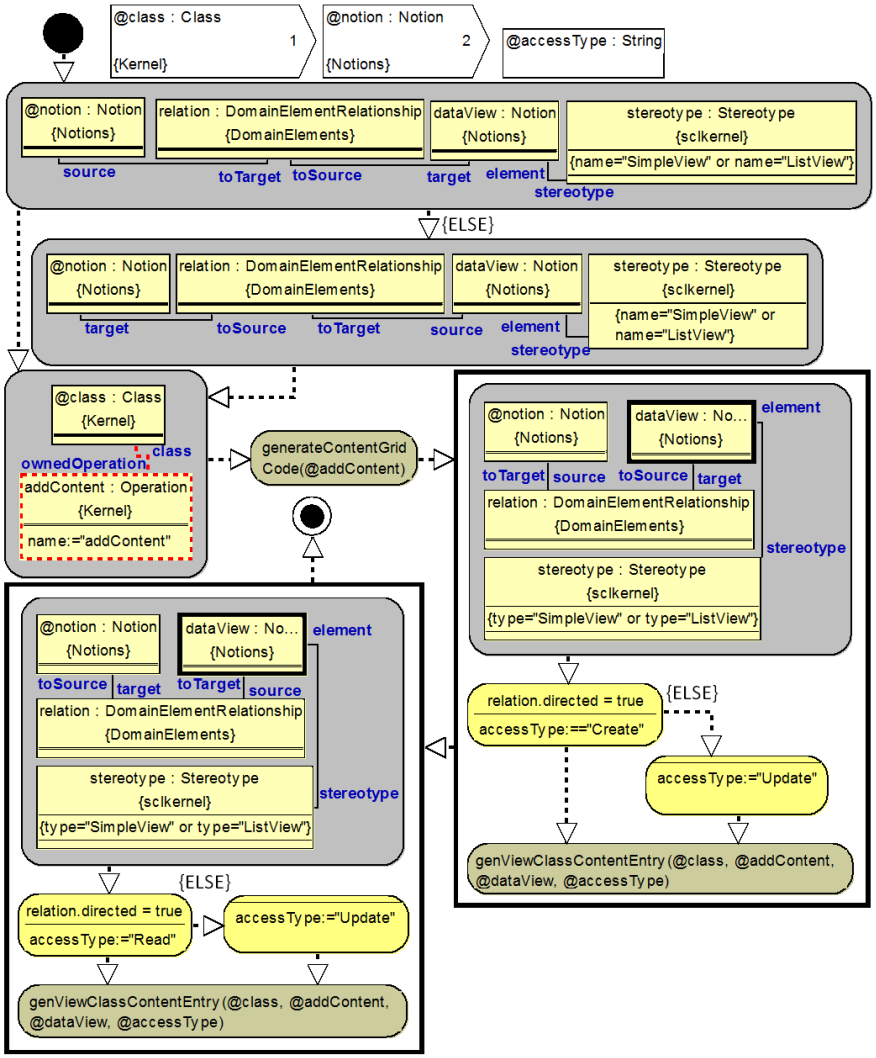


Figure 5. Procedure ('genViewClassContent') for generating the 'addContent' operation

ments. The actual transformation is much more elaborate and thus had to be simplified and abridged. Figure 4 shows the basic procedure ('genViewClass') for the creation of classes that handle widow-related code. These classes stem from the Screens or Messages. The appropriate Notion is given as the parameter to this procedure. After retrieving the Notion's name and converting it to camel case format, the procedure creates a properly named class (prefixed with 'V'). This class realises (see 'InterfaceRealisation') the standard 'ActionListener' interface. Then, the notion type is checked and depending on this, appropriate generalisation is created with either the standard 'WindowPane' or 'ContentPane' class. After this, the transformation calls the procedure to generate common code for all such classes ('genVClassStandardCode') and code individual for each class ('genViewClassContent').

Figure 5 shows this second procedure, which is more interesting. It generates the contents of the previously generated class, based on the features of the appropriate Notion and the associated elements. Firstly, the transformation checks the direction of the relation between the given Notion (Screen or Message) and another Notion which is a Simple View or a List View. Then depending on this determined direction, it assigns the type of access to window elements, to be provided by the generated controls. After that, the transformation generates an oper-

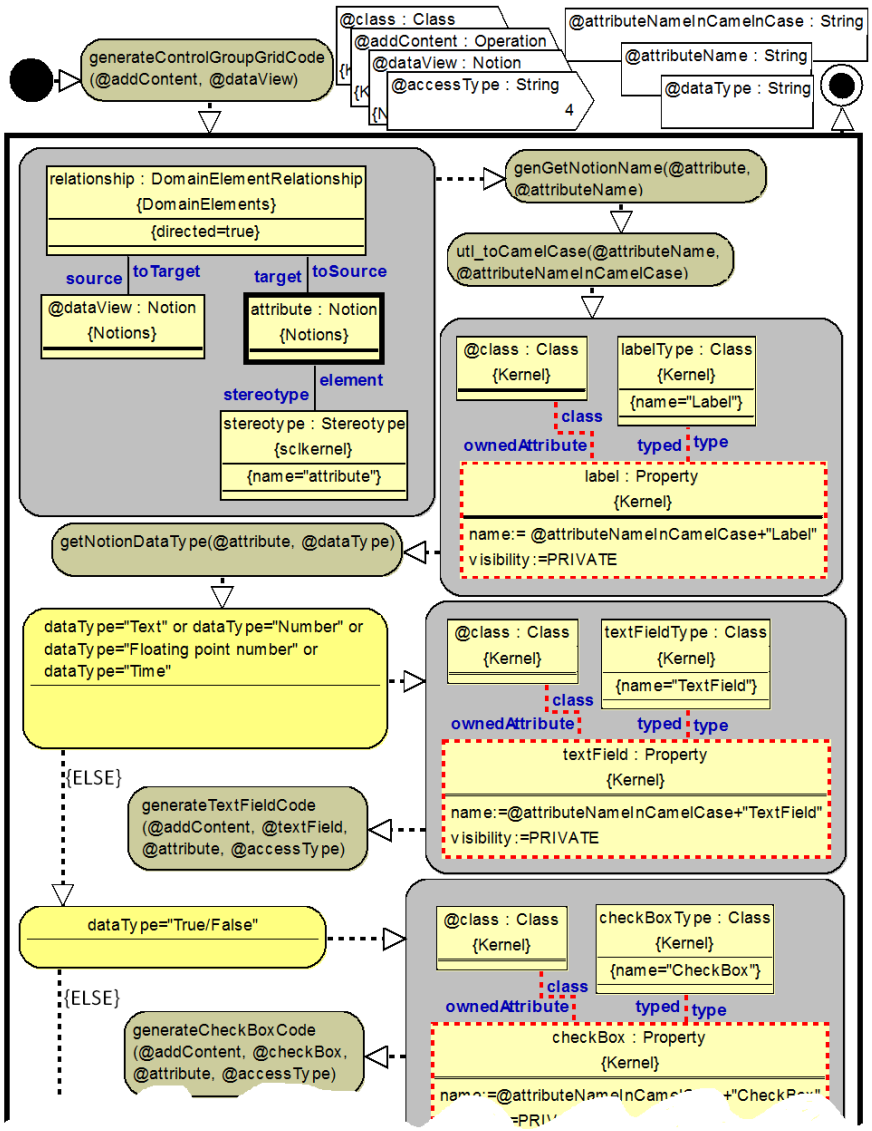


Figure 6. Procedure ('genViewClassContentEntry') for generating field initiation code

ation ('addContent') to fill the window content and fills it with standard code ('generateContentGridCode'). The last part of the procedure contains two loops (for two possible directions between the Notion and its related Data Views). In each iteration, an appropriate Data View and its Attributes are processed and appropriate field initiation code is created and inserted into the 'addContent' method.

Figure 6 shows fragment of the procedure that generates the actual field initiation code. Firstly, the standard initial part of code for the control group is generated through a call to an appropriate procedure. Then, a loop is performed for each Attribute pointed-to from the Data View which is the procedure's parameter. Inside the loop, firstly, the notion name is retrieved and converted to camel case format. After that, a private class property (attribute) is generated to hold the label field for the given Attribute. Then, the Attribute's data type is retrieved and depending on it, a property (attribute) for holding the actual control type is generated. For simplicity, the Figure shows fully only the fragment associated with the generation of Text Fields. The last part of the loop contains a call to the procedure that generates the proper code that initialises the just generated attributes.

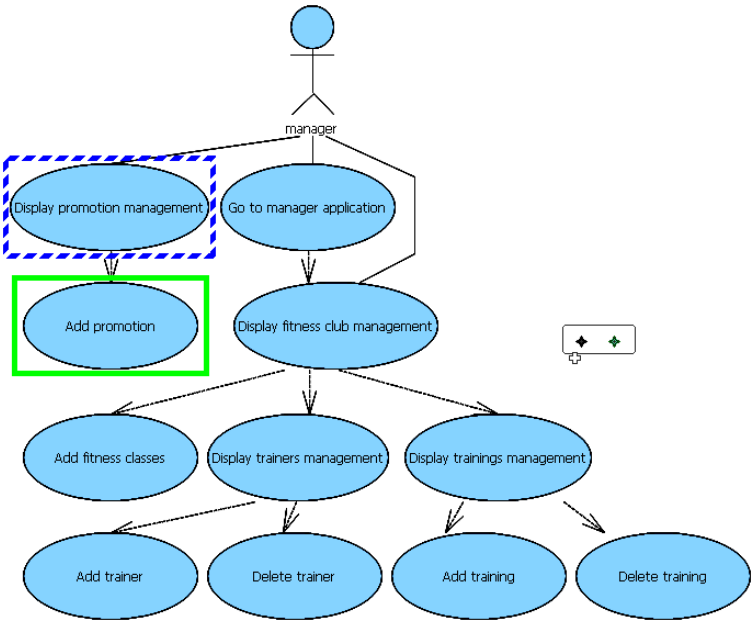


Figure 7. Use case model fragment for the case study

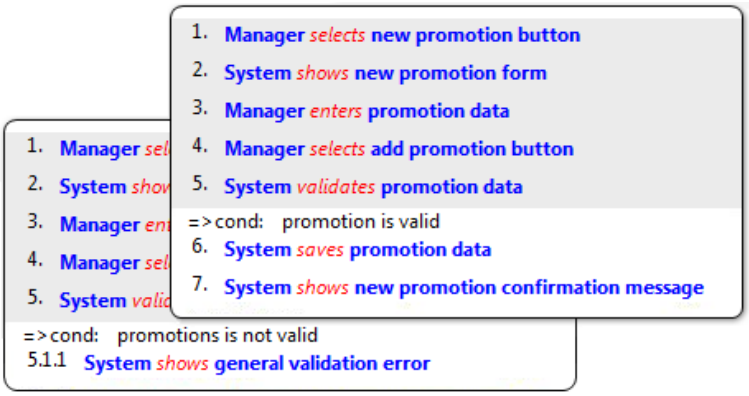


Figure 8. Scenarios of the "Add promotion" use case

As we can see, the output of the presented transformation is a UML model consisting of classes with attributes, operations and code embedded in these operations. The next step is to export this UML model and generate code with a UML tool providing an appropriate code generator (see Fig. 3). The code generator is invoked automatically and thus from the user perspective is seen as part of the overall transformation process. ReDSeeDS currently supports export and code generation using Enterprise Architect [31] and Modelio [32]. The full generated code complies with the Model-View-Presenter pattern [33] and is also based on the Echo framework [34].

6. Illustrative Example

The presented approach has been validated during a case study which was to implement a sports centre management system. This involved about 30 use cases, of which some are presented in Figure 7. In this brief example, we will show mainly the code generated around the domain models for the use case surrounded by the green thick frame ("Add promotion"). This use case has two scenarios, presented in Figure 8. In addition, we will show the user interface generated from the use case surrounded by a dashed blue frame ("Display Promotion management"). This will allow to present support for generating lists.

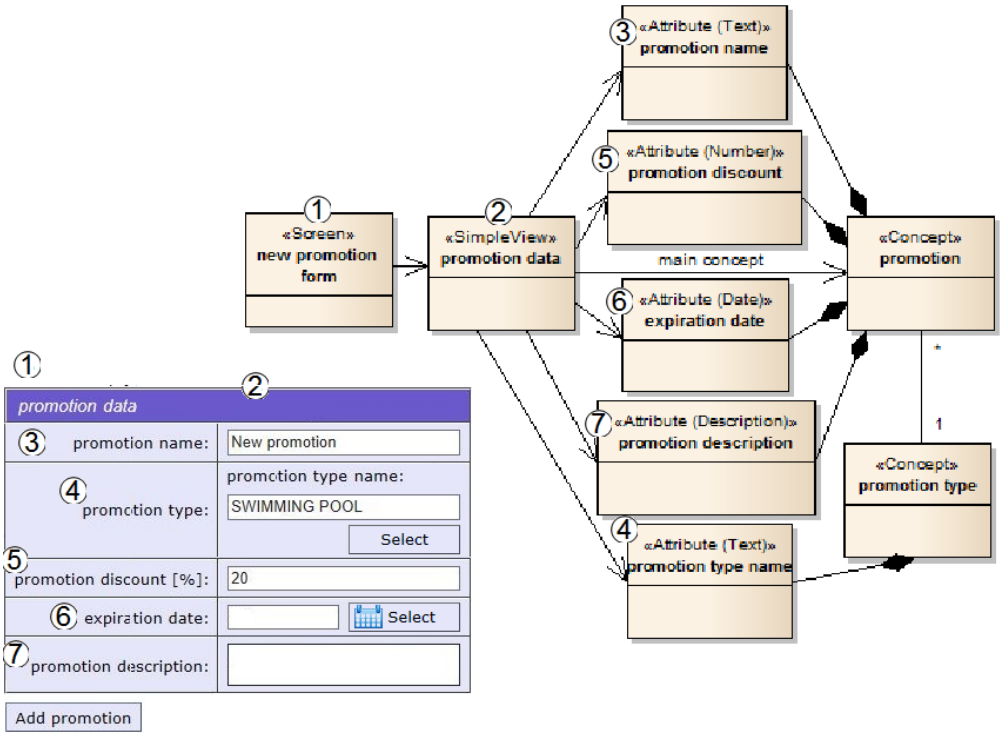


Figure 9. Domain model for the "new promotion form"

Figure 9 presents the domain model that complements the scenarios of "Add promotion", together with the actually generated user interface for the "new promotion form" window. We can observe that "new promotion form" is a Screen which points at "promotion data" which is a Simple View. This results in generating the "new promotion form" window (1) with the appropriate section corresponding to "promotion data" (2). The connection is directed from the Screen to the Simple View, which means that the window will serve entering data. The main Concept associated with "promotion data" is the "promotion". The Simple View points to several Attributes contained in the "promotion" and in the associated "promotion type" Concept. This set of relations to Attributes means that the section corresponding to "promotion data" will be filled with controls to input data related to the mentioned Attributes.

The types of these controls depends on the data types of the given attributes. We can see the equivalence in Figure 9. For instance, "Promotion name" (3) typed as Text is created as a Text field, and "Expiration date" (6) typed as Date is generated as a Text field with a button to open the date chooser.

A special case is the "Promotion type" (4) which is part of a Concept that is not the main Concept. It is generated into an separate embedded group of labelled controls. In this particular case, only one Text field ("Promotion type name") is generated from the appropriate Attribute. We can also notice an additional button ("Select") which was not covered by the semantic rules in the previous sections and can be used to select one value from a pop-up list. "Promotion type" takes the form of an embedded group of controls, not a list, because of the singular multiplicity of the relationship between the promotion (main Concept) and the promotion type (associated Concept).

Code for creating these controls as the content of "new promotion form" is shown in Figure 10. Apart from generating the fields, code contains creation of the "Add promotion" button. This is based on sentence 4, in relation to sentence 2 of the use case scenario shown in Figure 8. The code generator produces also an event handler associated with this button, presented in Fig-

```
private void addContent(){
    (...)
    promotionNameLabel = new Label("promotion name:  ");
    gridLayout = new GridLayoutData();
    gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
    promotionNameLabel.setLayoutData(gridLayout);
    promotionDataGrid.add(promotionNameLabel);
    promotionNameTextField = new TextField();
    promotionNameTextField.setWidth(new Extent(75, Extent.PERCENT));
    promotionDataGrid.add(promotionNameTextField);

    typeLabel = new Label("promotion type:  ");
    (...)
    discountLabel = new Label("promotion discount [%]:  ");
    (...)
    expirationDateLabel = new Label("expiration date:  ");
    (...)
    descriptionLabel = new Label("promotion description:  ");
    (...)
    addPromotionButton = new Button("Add promotion");
    addPromotionButton.setStyle(buttonStyle);
    addPromotionButton.setActionCommand("addPromotionButton");
    addPromotionButton.addActionListener(this);
    column.add(addPromotionButton);
}
```

Figure 10. Fragment of code for creating content of "new promotion form"

```
public void actionPerformed(ActionEvent e){

    (...)

    if (e.getActionCommand().equals("addPromotionButton")) {
        XPromotionData promotion = new XPromotionData();
        promotion.setPromotionName(promotionNameTextField.getText());
        promotion.setType(typeList.getSelectedValue().toString());
        promotion.setDiscount(Integer.parseInt(discountTextField.getText()));
        promotion.setDescription(descriptionTextArea.getText());
        promotion.setExpirationDate(DAOUtils.toSQLDate(expirationDateTextField.getText()));
        presenter.SelectsAddPromotionButton(promotion);
    }

    (...)

}
```

Figure 11. Handler code for the "add promotion" button

ure 11. This is presented to show completeness and coherence of the generated code but more detailed discussion is out of scope of this paper.

In addition to generating simple forms, the code generator can produce lists from List View elements. This is illustrated in Figure 12. The situation is in most part similar to the previous case, but data is represented in a collection form because a List View is used instead of a Simple View. Moreover, only some of the Attributes of the "promotion" are presented on the screen, because not all are connected to the List View.

7. Conclusion and Future Work

The presented approach aims to give the requirements model the feature of executability. The functional requirements are represented using the Requirements Specification Language in which emphasis is placed on both readability and precision. Using the presented transformation program in combination with a precise RSL specification, we obtain a typical business application, with simple, but fully functional graphical user interface, ready for deployment. Still, we can find

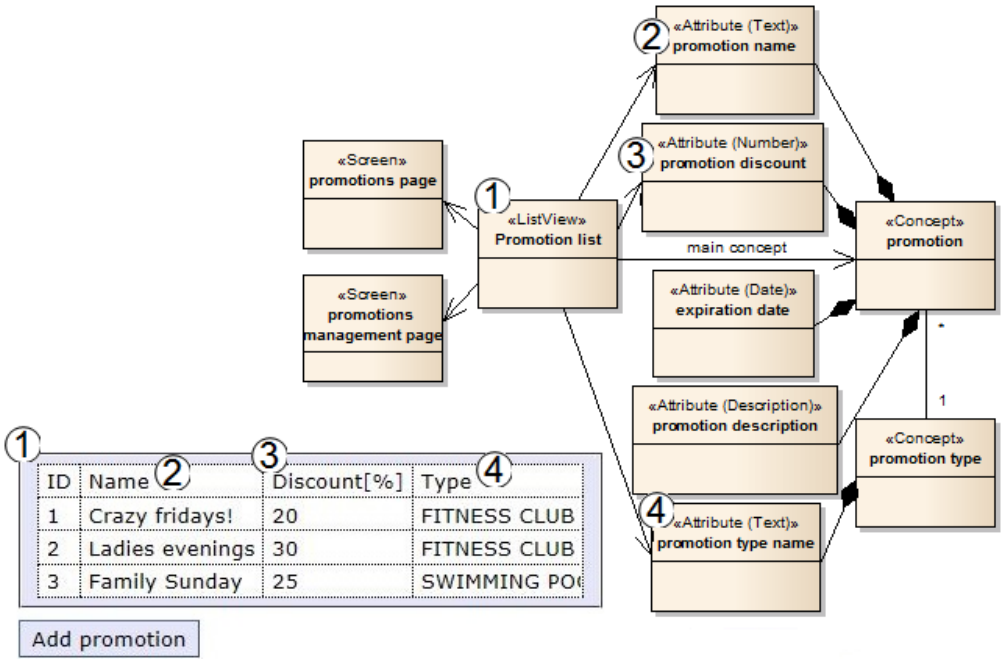


Figure 12. Domain model for the "promotion list"

some limitations of the presented transformations due to limitations of the current RSL syntax for domain elements. However, we plan to remove these limitations by extending the RSL notation and refining its semantics.

Currently, this approach can be used with success for fast prototyping. furthermore, through refinement of the graphical interface arrangement and use of appropriate outlook styles, it can also be brought to the condition of the final product. The presented solution is being validated on a much larger case study based on a legacy corporate banking system. Furthermore there are plans to conduct experiments with university students. Their goal will be to compare productivity and quality when the presented solution is used versus traditional approaches.

Future development work will include extending the ReDSeeDS tool with an editor to enable management of user interface element arrangement. There are also plans to further develop the overall transformation, taking into account various new technologies and platforms. There is ongoing work on developing new transformations which will provide high-level separation of concerns and thereby high reusability. In the future, the transformations are planned to offer several technology options to build the presentation layer such as Google Web Toolkit, Apache Wicket, JavaFX, Adobe Flex.

Acknowledgment

Part of this research has been carried out in the REMICS project and partially funded by the EU (contract number ICT-257793 under the 7th Framework Programme), see www.remics.eu.

References

[1] K. El Emam, "A Replicated Survey of IT Software Project Failures," *IEEE Software*, Vol. 25, No. 5, 2008, pp. 84–90.

[2] B. H. C. Cheng and J. Atlee, "Research Directions in Requirements Engineering," in *Future of Software Engineering, FOSE '07*, 2007, pp. 285–303.

[3] B. Berenbach, "A 25 year retrospective on model-driven requirements engineering," in *Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE*, 2012, pp. 87–91.

[4] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[5] D. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, Vol. 39, No. 2, 2006, pp. 25–31.

[6] S. Greenspan, J. Mylopoulos, and A. a. Borgida, "Capturing More World Knowledge in the Requirements Specification," in *Proc. 6th International Conference on Software Engineering*. IEEE Computer Society Press, 1982, pp. 225–234.

[7] S. Greenspan, J. Mylopoulos, and A. Borgida, "On formal requirements modeling languages: RML revisited," in *ICSE '94: Proc. 16th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 135–147.

[8] H. Kaindl, M. Śmiałek, P. Wagner, D. Svetinovic, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, H. Schwarz, D. Bildhauer, J. P. Brogan, K. S. Mukasa, K. Wolter, and T. Krebs, "Requirements Specification Language Definition," ReDSeeDS Project, Project Deliverable D2.4.2, 2009. [Online]. www.redseeds.eu

[9] J. Helming, M. Koegel, F. Schneider, M. Haeger, C. Kaminski, B. Bruegge, and B. Berenbach, "Towards a unified Requirements Modeling Language," in *Requirements Engineering Visualization (REV), 2010 Fifth International Workshop on*, Sept 2010, pp. 53–57.

[10] W. Nowakowski, M. Śmiałek, A. Ambroziewicz, and T. Straszak, "Requirements-Level Language and Tools for Capturing Software System Essence," *Computer Science and Information Systems*, Vol. 10, No. 4, 2013, pp. 1499–1524.

[11] M. Śmiałek, N. Jarzebowski, and W. Nowakowski, "Translation of Use Case Scenarios to Java Code," *Computer Science*, Vol. 13, No. 4, 2012, pp. 35–52.

[12] M. Śmiałek, W. Nowakowski, N. Jarzebowski, and A. Ambroziewicz, "From Use Cases and Their Relationships to Code," in *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012*. IEEE, 2012, pp. 9–18.

[13] T. Straszak and M. Śmiałek, *Advances in Software Development*. Polish Information Processing Society, 2013, ch. Acceptance test generation based on detailed use case models, pp. 116–126.

[14] "ReDSeeDS project home page." [Online]. http://redseeds.eu/

[15] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski, and T. Straszak, "Introducing a unified Requirements Specification Language," in *Proc. CEE-SET'2007, Software Engineering in Progress*. Nakom, 2007, pp. 172–183.

[16] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10–19.

[17] P. Shaker, J. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*, 2012, pp. 151–160.

[18] M. El-Attar and J. Miller, "AGADUC: Towards a More Precise Presentation of Functional Requirement in Use Case Mod," in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, 2006, pp. 346–353.

[19] R. Giganto and T. Smith, "Derivation of Classes from Use Cases Automatically Generated by a Three-Level Sentence Processing Algorithm," in *Systems, 2008. ICONS 08. Third International Conference on*, 2008, pp. 75–80.

[20] S. Mustafiz, J. Kienzle, and H. Vangheluwe, "Model transformation of dependability-focused requirements models," in *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, 2009, pp. 50–55.

[21] D. K. Deeptimahanti and R. Sanyal, "Semi-automatic generation of UML models from natural language requirements," in *Proceedings of the 4th India Software Engineering Conference*, ser. ISEC '11, 2011, pp. 165–174. [Online]. http://doi.acm.org/10.1145/1953355.1953378

[22] *Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02*, Object Management Group, 2009.

[23] "AndroMDA project home page." [Online]. http://andromda.org/

[24] "MDA website." [Online]. http://omg.org/mda/

[25] Y. Wang and M. Wu, "Case studies on translation of RTPA specifications into Java programs," in *Canadian Conference on Electrical and Computer Engineering*, Vol. 2, 2002, pp. 675–680.

[26] J. Falb, S. Kavaldjian, R. Popp, D. Raneburger, E. Arnautovic, and H. Kaindl, "Fully Automatic User Interface Generation from Discourse Models," in *Proceedings of the 14th International Conference on Intelligent User Interfaces*, ser. IUI '09. New York, NY, USA: ACM, 2009, pp. 475–476. [Online]. http://doi.acm.org/10.1145/1502650.1502722

[27] C. Janssen, A. Weisbecker, and J. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93. New York, NY, USA: ACM, 1993, pp.

418–423. [Online]. http://doi.acm.org/10.1145/169059.169335

[28] M. ElKoutbi, I. Khriss, and R. Keller, "Generating user interface prototypes from scenarios," in *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, 1999, pp. 150–158.

[29] M. Śmiałek, N. Jarzebowski, and W. Nowakowski, "Runtime semantics of use case stories," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, Sept 2012, pp. 159–162.

[30] A. Kalnins, J. Barzdins, and E. Celms, "Model Transformation Language MOLA,"*Lecture Notes in Computer Science*, Vol. 3599, 2004, pp. 14–28, MDAFA'04.

[31] "Enterprise Architect Website." [Online]. http://www.sparxsystems.com/products/ea/

[32] "Modelio Website." [Online]. http://www.modelio.org/

[33] M. Potel, "MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java," Taligent Inc., Tech. Rep., 1996. [Online]. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

[34] "Echo Framework Home Page." [Online]. http://echo.nextapp.com/

e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 53–64, DOI 10.5277/e-Inf140104

Supporting Analogy-based Effort Estimation with the Use of Ontologies

Joanna Kowalska*, Mirosław Ochodek*
*Faculty of Computing, Institute of Computing Science, Poznan University of Technology
miroslaw.ochodek@cs.put.poznan.pl

Abstract

The paper concerns effort estimation of software development projects, in particular, at the level of product delivery stages. It proposes a new approach to model project data to support expert-supervised analogy-based effort estimation. The data is modeled using Semantic Web technologies, such as Resource Description Framework (RDF) and Ontology Language for the Web (OWL). Moreover, in the paper, we define a method of supervised case-based reasoning. The method enables to search for similar projects' tasks at different levels of abstraction. For instance, instead of searching for a task performed by a specific person, one could look for tasks performed by people with similar capabilities. The proposed method relies on ontology that defines the core concepts and relationships. However, it is possible to introduce new classes and relationships, without the need of altering the search mechanisms. Finally, we implemented a prototype tool that was used to preliminary validate the proposed approach. We observed that the proposed approach could potentially help experts in estimating non-trivial tasks that are often underestimated.

1. Introduction

Accurate effort estimate is invaluable at every stage of software development. At early stages, it helps to assess feasibility of a project and negotiate the contract, whereas during product delivery stages, it helps to establish achievable deadlines and to reasonably allocate project resources.

Unfortunately, the unique nature of effort estimation at different stages of software development makes it difficult to establish a single, coherent method of collecting data for the purpose of effort prediction. The main reason of that is because the required level of details visibly differs between the levels of tasks. At the level of software development project we usually collect some of its general properties. For instance, in the ISBSG database [1] one can find information such as customer's domain, type of application, level of programming language, etc. This data is usually sufficient to identify and indicate the values of so-called cost drivers used in most of the model-based effort estimation methods (e.g., a well-known COCOMO II [2] defines 22 such factors – 17 cost-drivers and 5 scale-drivers) or to use analogy-based methods such as ACE [3], ANGEL [4], Estor [5]. However, such general data becomes less usable if one would like to estimate smaller tasks performed within short development cycles advocated by agile software development methods, like Scrum [6] or eXtreme Programming [7]. This is mainly because the contexts of such small tasks are more diverse, what makes definition of a universal set of cost drivers a cumbersome task. For instance, let us consider how contextually different could be these two tasks: conducting a meeting with a customer and implementing a login function in a web application.

This at least partially explains why estimation of low-level tasks is usually performed with the use of expert-judgment methods (e.g., group methods such as Planning Poker [8–11]) and why there are almost no model-based methods to estimate effort of such tasks. However, it is impor-

tant to mention that the expert-based judgment methods are far from being perfect, because they frequently involve a high degree of wishful thinking and inconsistency. In addition, their results could be biased by business pressure [12, 13]. According to Jørgensen [12] the organizations that have had the most success at meeting cost and schedule commitments use a mix of model-based and expert-judgment methods.

Therefore, the question arises whether it is possible to collect and store project data in such a way that it would enable to combine expert-based and model-based methods of project tasks estimation.

In the paper, we address this question by proposing a new approach to model information regarding projects tasks. Our ultimate goal is to combine expert-based and analogy-based effort estimation methods. The proposed approach is based on Semantic Web technologies, such as Resource Description Framework (RDF) and Ontology Language for the Web (OWL) and has the following features:

- it enables to model and store information regarding project tasks and allows to dynamically extend the ontology by introducing new concepts and relationships (Section 2),
- it supports supervised case-based reasoning – allows to dynamically change the abstraction level of search criteria (Section 3),
- it can be potentially applied to support expert-based effort estimation at the level of product delivery stage (Section 4).

2. Modeling Projects Tasks

Semantic Web technologies in their simplest form offer means to express and store facts in the form of triples (subject, predicate, object) using Resource Description Framework (RDF). Each piece of information is uniquely identified by its Uniform Resource Identifier (URI). This representation of information can be augmented with ontologies expressed in one of the variants of Ontology Language for the Web (OWL). It is also possible to use reasoners and rules engines.

The *ontology* forms an information domain model. It uses a predefined, reserved vocabulary of terms to define *concepts* and the *relationships* between them for a specific area of interest, or domain [14]. Although ontologies are developed and studied for many years, we have recently observed rapid evolution of technologies that support ontology modeling.

An example of a simple knowledge base in a form of semantic network is presented in Figure 1. It states that there are two *individuals*: John and Simon. Each of them is uniquely identified by its URI, e.g., my_data:John[1]. Both John and Simon belong to the class my_onto:Person (Person has a type of owl:Class). Because they are people, they have property my_onto:hasName, which represents person's name. In addition, there is a relationship between both of them stating that John knows Simon.

The great advantage of using Semantic Web technologies to store information is that the data model can be easily extended. It is easy to introduce new individuals, classes, properties and constraints – usually, without the need of modifying the source code of a computer program.

2.1. Projects Tasks Ontology

Assuming that the contexts can differ visibly between project tasks, we would like to propose an ontology that defines the most important concepts and relationships to enable modeling project tasks for the purpose of effort estimation. We also assume that the ontology can be extended by definitions of new classes and relationships that are characteristic for a specific context.

The proposed knowledge model will focus on modeling five types of facts regarding project task:

- **Who?** – it represents information about the one that performed the task. It could be either an individual or a group of people.
- **Did what?** – it corresponds to both the type of activity and inputs/outputs of the task.
- **How?** – it regards any tools, methods, technologies that were used to complete the task.

[1] We are going to omit the namespace part of URI (e.g., my_data:) unless there is a collision between names.

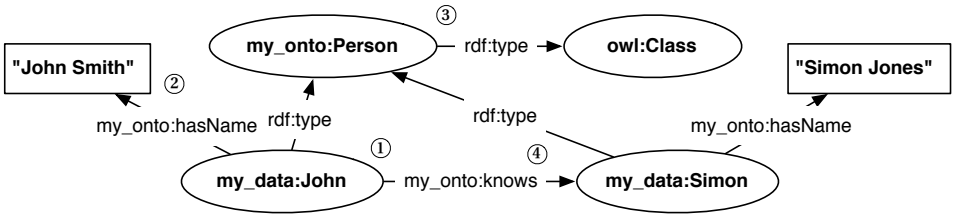


Figure 1. An example of knowledge representation in RDF and OWL (① an individual that belongs to the class Person; ② a data property stating that John has a name John Smith; ③ Person is an OWL class; ④ an object property stating that John knows Simon)

– **When?** – it relates to the actual effort and timespan of the task completion.

An exemplary knowledge base storing information about a project task called Task1 is presented in Figure 2. It shows the usage of classes (ellipses with dashed lines) and relationships defined in the proposed ontology. a project task is represented by an individual that belongs to the class *Task*. Each task can have a number of properties corresponding to the aforementioned questions – Who?, Did what?, How? and When?:

– *hasPerformer* – it relates to individuals belonging to the class *Performer* (or its subclasses) that were involved in the completion of the task. Performers can have different capabilities indicated by the property *hasCapability*. a capability has its *level* and the property *in* referring to the subject the capability concerns. In the example, Task1 was performed by John Smith, who is highly skilled Java developer.
– *hasInput* – this property describes all the prerequisites of the task, e.g., requirements, constraints. In the example, Task1 has a single input. It is a use case (UC1) describing user functional requirements to be implemented. We do not restrict the types of inputs to any classes. However, an input can poses a property *hasSize* that is recognized and interpreted by the case-based reasoning algorithm. For instance, the size of the use case UC1 is expressed using the number-of-steps measure. We would also like to emphasize that the presented ontology could be dynamically extended or merged with existing domain ontologies to precisely model the inputs. For instance, UC1 belongs to the class *Creation Use Case* that is not a part of the proposed ontology, however, it still can be used to support effort estimation.
– *hasType* – it represents the type of activity being performed. The taxonomy of types has a hierarchical structure.
– *hasMeans* – the property determines all the means that were used to complete the task. In the example, Java was used to implement UC1.
– *hasOutput* – it represents the artifacts that need to be produced.
– *hasSource* – it provides information about the entity that proposed the task. For instance, it could be a person or company.
– *actualEffortInHours* and *estimatedEffortInHours* – the properties correspond to the actual effort of the tasks, and if available, its estimated effort.
– *from* and *to* – properties defining a timespan when the task was performed.

Tasks can be composed into hierarchies using the *subTaskOf* relationship. This relationship is transitive, which means that if a task has sub-tasks defined, it automatically poses all their features. For instance, in the showed example, Task1 is a sub-task of Task2. This means that Task2 poses all the properties of Task1. For instance, one could conclude that John Smith also participated in completion of Task2. In addition, one of the goals of Task2 was to implement UC1. The composition of tasks enables to compare tasks at different levels.

3. Supervised Case-based Reasoning

In order to perform case-based reasoning using the proposed ontology we need a method to nav-

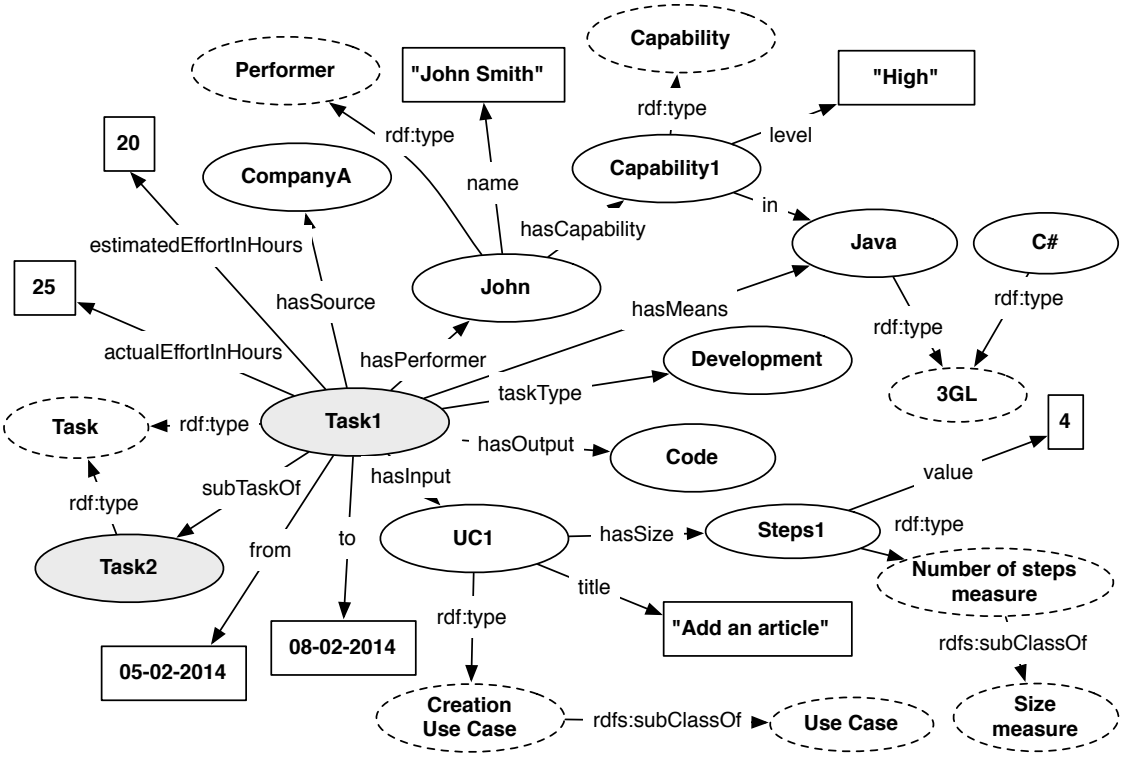


Figure 2. An example of project-task model using the proposed ontology

igate through semantic network. As the individuals in the ontology form a complicated graph of relationships, we decided to introduce *x-level* notation to indicate the depth of graph exploration. For instance, 1-level of navigation means that the exploration starts at the given node (RDF resource or OWL class) and finishes at the node's direct neighbors. The 2-level navigation implies traversing through all nodes available on 1-level and recursive invocation of 1-level navigation for each of them.

The main goal of the proposed case-based reasoning method is to give the expert possibility to dynamically adjust the demanded level of similarity between tasks. We defined five levels of similarity:

– *Near-exact similarity* – only tasks which have exactly the same values of all properties at 1-level would be classified as similar. For instance, if two tasks are being compared that have almost the same values of all properties, but the sets of performers are different, then, these tasks will not meet conditions to classify them as similar.
– *Similarity after generalization to a given class* – generalization can be defined as navigating up in the hierarchical taxonomy of classes. If two tasks were connected to individuals belonging to the same, given class, then these two tasks would be classified as similar. For instance, let us assume that there are two tasks: the first one was implemented in Java and the second one was implemented in C#. If one considers their similarity after generalizing them to the class *3GL programming language*, then the tasks would be considered similar.
– *Similarity after generalization to classes on a given level* – this approach is more general than the generalization to a given class, because the process of navigating up in class hierarchy is not based on a single class, but it is performed for all classes on a given level. For instance, if a task has individuals that *directly* belong to both 3GL programming language and Web Framework classes, a similar task will also have to be connected to individuals that belong to these classes.

- *Similarity when values of a given property are equal* – tasks in the project ontology are not only connected with individuals, but also with plain values. Generalizations work only for class instances, so there is a need to introduce a mechanism of comparing tasks based on so-called *datatype properties* (e.g., integers, strings, etc.). Tasks are considered similar if they have the same values of a given property.
- *Similarity when values of properties on a given level are equal* – it is a more general version of similarity based on equality of properties. This time, all datatype properties at a given level need to be the same to conclude that the tasks are similar.

The proposed approach makes it possible to give the expert opportunity to select which levels of similarity should be selected in a given context. The decision is made by invoking one of the following commands:

- ExactSearch() – it performs a search using *near-exact similarity* comparison,
- Generalize(Relation, Class) – it alters search criteria by introducing the similarity after generalization to the *Class*.
- Generalize(Relation, Level) – it alters search criteria by introducing the similarity after generalization to the classes on the given *Level*.
- SameProperties(Relation, Property) – it alters search criteria by introducing the similarity when values of the given *Property* are equal.
- SameProperties(Relation, Level) – it alters search criteria by introducing the similarity when values of the properties on the given *Level* are equal.

The *Relation* parameter shows in which direction the mechanism should work. It is important to emphasize, that if an expert decides to execute command on the specific relation, then the remaining relations still have to match the previously defined criteria. In addition, all the previously applied commands might be reverted.

The method always starts from the *ExactSearch* command, because it finds the tasks that are the most similar. Afterwards, an expert has possibility to execute different commands and observe the results. The results could be any means supporting expert-based effort estimation, e.g., cumulative density function plots, regression-based models, description of similar tasks.

An example of supervised search session is presented in Figure 3. It presents how the similarity assessment of Task1 and Task2 changes due to execution of commands. Initially, the tasks cannot be classified as similar, because on the 1-level only the *Development* and *Java* is connected to both of them. When the expert executes the SameProperties(hasPerformer, 2) command, *John* and *Anna* become similar, because they both share the same node *HighJava* on the 2-level. Finally, the expert executes Generalize(hasInput, 1) that generalizes UC1 and UC3 to the same class *Creation Use Case*. As a result, Task1 and Task2 are classified as similar.

4. Preliminary Empirical Evaluation

In order to preliminary evaluate the potential usefulness of the proposed approach, we decided to perform a post-mortem analysis of a software development project. In particular, we wanted to investigate whether estimates provided by the proposed method could potentially prevent experts from making the most significant estimation errors (especially prevent them from underestimating effort of tasks).

For the purpose of the method evaluation we implemented a prototype tool on the top of Apache Jena Framework. At current stage of development, the tool cannot be used on-line by an expert, because it lacks easy-to-use user interface. Therefore, instead of conducting the action research study, we decided to perform analysis of existing data. This, however, visibly limits the conclusions we could draw from the study.

4.1. The Project under Study

The selected project (eProto3) was an in-house software development project conducted at Poznan University of Technology (PUT) in

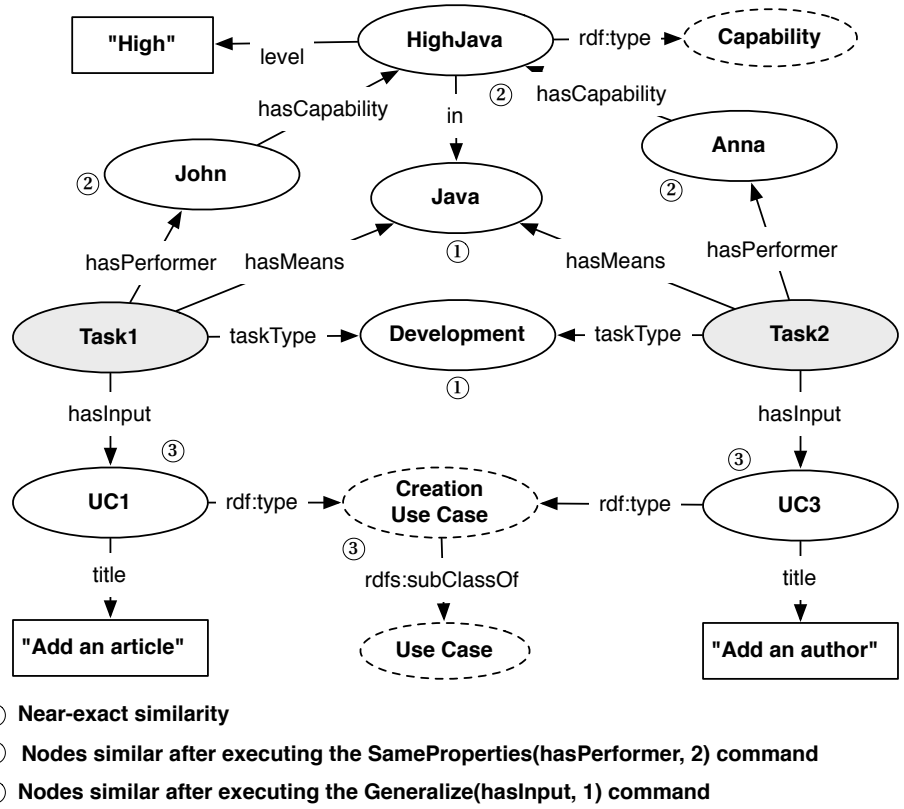


Figure 3. An example of supervised search

2011-2012. Its main goal was to enhance the existing system used to collect students' final grades. The development of the new version of the system was one of the steps taken by the University to fully eliminate the need of paper students' record books.

The project was conducted according to the XPrince methodology [15], which combines PRINCE2 [16] at organization level and eXtreme Programming [7] at the product delivery level.

The project team consisted of PUT employees, 3rd and 4th year students. The total reported effort in the project was around 1600 man-hours.

The lifecycle of the project was convergent with PRINCE2 recommendations. During the Initiating a Project stage (IP) non-functional and functional requirements in form of use cases were elicited. The prepared software requirements specification (SRS) served as a product backlog. XPrince assumes that delivery stages are organized similarly to releases in most of agile software development methods. The scope of each delivery stage was agreed during a Planning Game session [7]. Therefore, it was possible that the project would not deliver whole functionality that was defined in SRS.

The analysis was performed based on the tasks recorded in the project's issue tracker (Redmine) during the IP stage, and three delivery stages (the distributions of the tasks' actual effort are presented in Figure 4). Tasks contained information about the estimated effort by the project team members (we would refer to them as expert estimates) and actual effort. The recorded tasks related to large variety of activities, e.g., meetings, requirements engineering, implementation, testing, etc. Many of these tasks had hierarchical structure, especially ones defined during the delivery stages. For instance, each delivery stage had a corresponding task, which was decomposed into set of smaller tasks. For example, some of the sub-tasks were concerning implementation of use cases. These tasks were further decomposed to tasks which goals were to implement use-case steps.

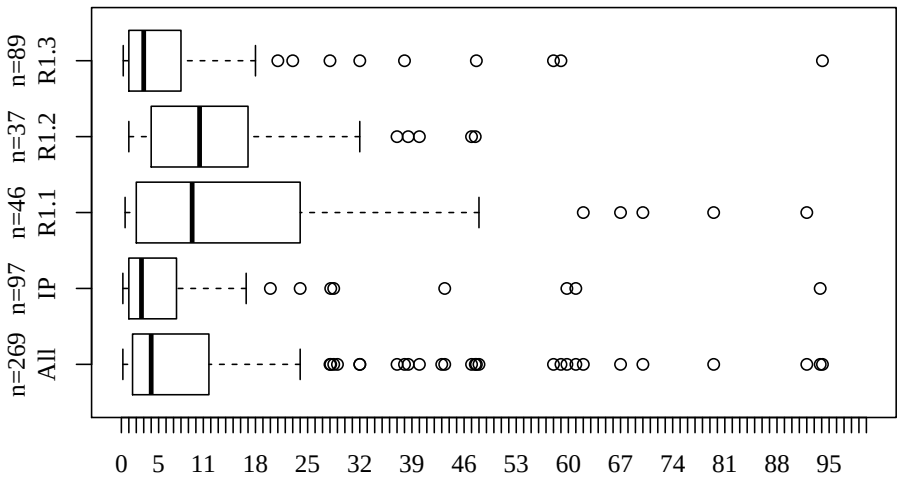


Figure 4. Box-plots presenting actual effort of the tasks in man-hours for all stages (All), Initiating a Project stage (IP), and three releases (R1.1-3)

We were able to automatically retrieve most of the data from the Redmine instance and missing information, e.g., performers capabilities, was added manually (in this particular case, we surveyed team members about their capabilities during the project).

4.2. Evaluation Methodology

We wanted to compare the accuracy of tasks' estimates obtained from three sources:

- (ES) analogy-based effort estimates based on the results of *exact search*,
- (SS) analogy-based effort estimates based on *supervised search*,
- (Exp) project team members' estimates (*experts' estimates*).

In order to compare the accuracy of estimates we used a prediction error metric called balanced measure of relative error (BRE)[2], which is calculated as:

$$BRE = \frac{|actual\ effort - estimated\ effort|}{min(actual\ effort, estimated\ effort)}$$

We also wanted to investigate if the prediction method provided unbiased results. For this purpose, we used slightly modified version of BRE measure, called BRE_bias that is defined in the following way:

$$BRE\_bias = \frac{actual\ effort - estimated\ effort}{min(actual\ effort, estimated\ effort)}$$

If the value of BRE_bias measure is greater than zero it means that the effort was underestimated. Negative value indicates overestimation.

We decided to analyze the accuracy of effort prediction approaches using the k-fold cross-validation method. In the first step we randomly divided the set of tasks into $k = 10$ exclusive subsets with possibly equal cardinality. The validation process took k iterations. During each of the iterations, a single set T became a testing set while the remaining $k-1$ sets were treated as historical database. Each task from the set T was estimated using a given effort estimation approach. The obtained estimate was compared with the actual effort to calculate prediction error measures.

By definition, the supervised search approach should be used by an expert, who executes the commands in order to search for similar tasks. The choice of command that expert executes is determined by the results obtained in the previous step. Therefore, the expert search strategy can differ depending on the task being estimated

[2] We decided to use the BRE error measure instead of MRE (Magnitude of Relative Error), which was more frequently used in the past, because the latter one was recently criticized by many researchers, mainly for being unbalanced [17–20].

and content of historical database. Unfortunately, we were not able to simulate such complex behavior. Thus, in the analysis we defined a simple strategy that our virtual expert used to supervise the search. The will of the expert to refine the search was based on the number of similar tasks found in the previous iteration. If the previous steps did not provide a single similar task, expert performed the following steps (after each step verifying if there are any similar tasks found)[3]:

1. ExactSearch – the search started from finding nearly-the-same tasks.
2. Generalize(hasInput, 1) – we decided that the first refinement should concern making inputs more abstract (e.g., instead of UC1 we could have a use case with the main theme of creating an object in the system).
3. SameProperties(hasPerformer, 2) – instead of finding exactly the same performers, we would like to find performers with the same capabilities (e.g., highly skillful Java programmers instead of John Smith)[4].
4. Generalize(hasMeans, 1) – we searched for similar tasks that were performed with the use of similar tools, programming language or technologies.
5. SameProperties(hasType, 1) – finally, we try to look for the tasks similar tasks that have little bit more general type.

The second stage of the case-base reasoning is to predict effort of the task based on found similar tasks. In the study, we used the following strategy to estimate effort. If size was available both estimated and similar historical tasks, we constructed a linear regression model. If the size was not measured for the inputs, we selected mean actual effort of similar tasks as the task estimate.

4.3. Data Analysis

During the analysis of the eProto3 project data it turned out that the experts' estimates were provided for 132 out of 269 tasks (Exp). In addition, the exact search approach was able to estimate 100 tasks (ES) and the supervised search provided estimates for 199 tasks (SS). Therefore, in order to compare the prediction accuracy of approaches we decided to analyze the following sets of tasks: a = Exp ∩ ES ∩ SS (51 tasks), B = Exp ∩ SS (100 tasks) and C = ES ∩ SS (100 tasks).

The first observation was that when all tasks are considered, expert-based estimated are the most accurate (error measures are presented in Table 1). The average values of BRE ranged from 0.33 to 1.06 (depending on the measure of central tendency and set of tasks). They also seemed to be median-unbiased, while for mean-bias we observed a tendency to underestimate. The exact and supervised search approaches on average performed visibly worse than experts – average BRE ranged from 0.76 to 2.53. The estimates seemed to be median-unbiased and contrary to experts' estimates we observe a tendency to overestimate for mean-bias (which from practical point of view is favorable).

The second, not surprising, observation was that the experts performed almost perfect when it comes to small tasks (e.g., 1 man-hour or less). Therefore, it seems that for such tasks no support is necessary. However, taking into account how short iteration-cycles are planned in agile software development, it is rarely observed that tasks are decomposed to such a level. In eProto3 project, during the Planning Game sessions the negotiation between the customer representative and development team was usually at the level of use cases (and rarely at the level of use-case steps). Team members often added the estimates of smaller tasks during the development.

As a result, we decided to filter out tasks having actual effort lesser than 1 man-day (8 man-hours) and repeat the analysis. This time the on average values of BRE for experts' estimates ranged from 0.70 to 2.56. We also observed a visible tendency to underestimate effort of bigger tasks by the experts. The exact and supervised search approaches on average performed

[3] As it was presented in Section 4, the execution of commands Generalize and SameProperties does not redefine the search criteria, but refine the existing ones.

[4] During the analysis, it turned out that eProto3 team members had exclusive sets of capabilities, therefore, this step did not have any effect on the results.

Table 1. Effort estimation errors (BRE and BRE_bias)

		BRE			BRE_bias		
	Tasks set	median	mean	SD	median	mean	SD
All tasks:							
Experts	A = 51	0.33	1.06	2.17	0.00	0.43	2.38
Exact Search	A = 51	1.00	1.91	3.85	0.00	−0.33	4.29
Supervised Search	A = 51	1.00	1.91	3.85	0.00	−0.33	4.29
Experts	B = 100	0.45	1.05	1.80	0.00	0.39	2.05
Supervised Search	B = 100	1.19	2.53	3.96	−0.06	−1.25	4.53
Exact Search	C = 100	0.76	1.78	3.05	−0.01	−0.52	3.49
Supervised Search	C = 100	0.76	1.78	3.05	−0.01	−0.52	3.49
Actual effort >= 8h:							
Experts	A' = 8	0.85	2.56	4.25	0.79	2.40	4.35
Exact Search	A' = 8	0.00	0.38	0.63	0.00	0.03	0.75
Supervised Search	A" = 8	0.00	0.38	0.63	0.00	0.03	0.75
Experts	B' = 31	0.70	1.49	2.43	0.60	1.29	2.55
Supervised Search	B' = 31	0.70	0.95	0.93	0.00	−0.23	1.32
Exact Search	C' = 10	0.00	0.33	0.57	0.00	0.00	0.67
Supervised Search	C' = 10	0.00	0.33	0.57	0.00	0.00	0.67
All tasks and BRE Experts > 2: (the results for C would be the same as for A)							
Experts	A" = 6	4.81	6.07	3.32	4.06	2.91	6.72
Exact Search	A" = 6	1.69	2.82	3.88	0.22	−1.24	4.77
Supervised Search	A" = 6	1.69	2.82	3.88	0.22	−1.24	4.77
Experts	B" = 14	4.00	4.75	2.47	3.65	2.39	4.92
Supervised Search	B" = 14	1.35	3.16	3.84	−0.64	−2.27	4.47

a little bit better than experts – the average BRE ranged from 0.00 to 0.95 (the most important comparison, based on the set B' indicated difference in median BRE between experts and the supervised search approach at the level of 0.00 and for mean BRE at the level of 0.54). Again the proposed approaches seemed almost unbiased (in one case a minor tendency to overestimate was observed).

The goal of the proposed analogy-based effort estimation method is to support, not eliminate, expert in effort estimation. Therefore, we decided to investigate if the proposed approaches could potentially prevent experts from making most harmful errors in their estimations. The idea was to select tasks that had BRE for expert-based effort estimates greater than 2.00 and observe their corresponding estimates suggested by the tool. The first observation was that both experts and proposed approaches were not able to provide accurate estimates. The average BRE for experts ranged from 4.00 to 6.07 with major tendency to underestimate. The proposed approaches performed better – the average BRE ranged from 1.35 to 3.16. With a single exception, the proposed approaches had tendency to overestimate.

4.4. Discussion of the Results

First of all, we want to emphasize that the goal of the study was not to prove that the method provides estimates with higher accuracy than experts. Instead we treated it as a preliminary study that would show us further directions for improvements.

To sum up the results, we have to admit that generally team members were able to provide accurate effort estimates. The estimates provided by tool, especially for small tasks, were less accurate. However, when the size of the task increased,

the accuracy of the tool was comparable, or taking into account its tendency to overestimate even practically favorable.

We observed that the main reason of poor performance of the proposed approaches was the lack of quantitative complexity measured for most of the tasks. We observed that accuracy of the tool could be increased either by providing these kind of measures (e.g., even simple measure such as number of pages of documentation to be produced, etc.) or to precisely describe tasks using the ontology. We believe that the problem could be mitigated if a true human expert was supervising the tool. From our investigation many of the tasks were correctly classified as similar, taking into account available information, however, after reading their titles, the difference between them became obvious.

We also observed that the tool was able to provide better estimates for the tasks that were poorly estimated by experts, which in our opinion is a promising finding. However, still the question arises if the feedback provided by the system would have strong-enough impact on experts' decisions to prevent them from making significant mistakes in their estimates.

4.5. Limitations and Threats to Validity of the Study

There are limitations and threats to validity of the study that needs to discussed. The main threat to construct validity relates to the fact that the proposed study assessed only some aspects of the proposed approaches. We believe that the approaches should support expert-based effort estimation, e.g., as an external voice in group-based effort estimation methods like Planning Poker. However, in the study we simulated behavior of an expert, who always performed in the same way (even if it was unreasonable in a given context).

The main threats to internal validity relate mainly to the project data we obtained from the Redmine system. We suspect that for smaller tasks experts could record actual effort in such a way that it fit the estimated effort (e.g., if one completes a task that was estimated for 30 minutes in 20 minutes he/she very often will just copy the estimated effort as actual). From the practical point of view the difference is not so visible, but when it comes to calculating BRE measure its impact becomes visible.

The threats to external validity regard the ability to generalize the findings. The goal of the study was to collect first observations regarding the method. Therefore, this group of threats does not affect the results too much. The most important threats in this category refer to the size of the sample and software development methodology that was used. For instance, the requirements were documented in the form of use cases rather than in the form of user stories.

5. Related Work

There are three categories of related works that we would like to discuss, namely, analogy-based effort estimation, supporting effort estimation during release planning in agile software development and usage of ontologies for effort estimation.

Analogy-based effort estimation has been developed for many years. Probably the most recognized methods of this type are ACE [3], ANGEL [4], Estor [5].

The main challenge of analogy-based effort estimation is the construction of a mechanism that will enable us to find similar cases (projects) to the target one that is estimated. Most of the methods tackle with this problem by representing software projects in vector spaces (each feature is represented by a single dimension). Then various techniques are used to find similar projects, e.g., based on different similarity distance measures, e.g., Euclidean, Manhattan, Minkowski.

Another, important problem is that the accuracy of analogy-based methods strongly dependent on the precision of historical data. Recently, Azzeh et al. [21] proposed to use Fuzzy numbers to mitigate this problem. The advantage of this solution is that it can be applied when not all requirements are known. The main drawback is that it is only usable on the project level.

Our approached differs visibly from the previous works in the area, because it enhances case-based reasoning process with semantics. Giving the analogy to the approaches using vector spaces, we could say that we are able to dynamically transform the vector space that is used to describe the projects and to find similarities between them.

Still, the main aim of our approach is to support effort estimation during release planning activities (especially, in agile software development). Majority of related works in this area focus on expert-based (and particularly grouped-based) effort estimation methods. For instance, the Planning Poker method has been recently frequently studied [8–11]. However, there are some works concerning usage of model-based effort estimation methods at the release level. For instance, Hearty et al. [22] proposed the method to predict Project Velocity using Bayesian Nets (BNs); Miranda et al. [23] proposed an approach to support sizing of user stories based on paired comparison.

The usage of ontologies to effort estimation was considered by Hamadan et al. [24]. They identified the importance of organizational and cultural factors and project leadership for improving effort estimates by analogy. The authors created a project ontology, which focuses on the environmental factors. Distance between projects was calculated and used to assess their similarity. However, this approach could be used only at the project level and requires a large number of similar projects in the database.

6. Conclusions

We proposed a new approach to model project data to support expert-supervised analogy-based effort estimation. The data is modeled using Semantic Web technologies, such as Resource Description Framework (RDF) and Ontology Language for the Web (OWL).

In addition, we defined a method of supervised case-based reasoning. The method enables to search for similar project tasks at different levels of abstraction. For instance, instead of searching for a task performed by a specific person, one could look for tasks performed by people with similar capabilities.

The proposed method relies on ontology that defines the core concepts and relationships. However, it is possible to introduce new classes and relationships, without the need of altering the search mechanisms.

Finally, we implemented a prototype tool that was used to preliminary validate the proposed approach. We observed that the proposed approach could potentially help experts in estimating non-trivial tasks that are often underestimated.

References

[1] P. R. Hill, *Practical Software Project Estimation: A Toolkit for Estimating Software Development Effort & Duration*. McGraw-Hill, 2011.

[2] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, Vol. 1, No. 1, 1995, pp. 57–94.

[3] F. Walkerden and R. Jeffery, "An empirical study of analogy-based software effort estimation," *Empirical Software Engineering*, Vol. 4, No. 2, 1999, pp. 135–158.

[4] M. Shepperd, C. Schofield, and B. Kitchenham, "Effort estimation using analogy," in *Proceedings of the 18th International Conference on Software Engineering, Berlin, 1996*. IEEE, 1996, pp. 170–178.

[5] T. Mukhopadhyay, S. Vicinanza, and M. Prietula, "Examining the feasibility of a case-based reasoning model for software effort estimation," *MIS Quarterly*, Vol. 16, No. 2, 1992, pp. 155–171.

[6] K. Schwaber and M. Beedle, *Agile software development with Scrum*. Prentice Hall, 2002.

[7] K. Beck and C. Andres, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[8] J. Grenning, "Planning poker or how to avoid analysis paralysis while release planning," 2002.

[9] K. Moløkken-Østvold, N. C. Haugen, and H. C. Benestad, "Using planning poker for combining expert estimates in software projects," *Journal of Systems and Software*, Vol. 81, No. 12, 2008, pp. 2106–2117.

[10] V. Mahnič, "A case study on agile estimating and planning using scrum," *Electronics and Electrical Engineering*, Vol. 111, No. 5, 2011, pp. 123–128.

[11] V. Mahnič and T. Hovelja, "On using planning poker for estimating user stories," *Journal of Systems and Software*, Vol. 85, No. 9, 2012, pp. 2086–2095.

[12] M. Jorgensen, B. Boehm, and S. Rifkin, "Software development effort estimation: Formal models or expert judgment?" *Software, IEEE*, Vol. 26, No. 2, March 2009, pp. 14–19.

[13] R. Popli and N. Chauhan, "Cost and effort estimation in agile software development," in *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*. IEEE, 2014, pp. 57–61.

[14] J. Hebeler, M. Fisher, R. Blace, and A. Perez-Lopez, *Semantic web programming*. John Wiley & Sons, 2011.

[15] J. Nawrocki, L. Olek, M. Jasinski, B. Paliświat, B. Walter, B. Pietrzak, and P. Godek, "Balancing agility and discipline with xprince," in *Rapid integration of software engineering techniques*. Springer, 2006, pp. 266–277.

[16] A. Murray, *Managing Successful Projects with PRINCE2*. TSO, 2009.

[17] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion mmre," *IEEE Transactions on Software Engineering*, Vol. 29, No. 11, 2003, pp. 985–995.

[18] M. Jørgensen, "A critique of how we measure and interpret the accuracy of software development effort estimation," in *First International Workshop on Software Productivity Analysis and Cost Estimation*. Nagoya: Information Processing Society of Japan, 2007.

[19] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd, "What accuracy statistics really measure [software estimation]," in *Software, IEE Proceedings*, Vol. 148, No. 3. IET, 2001, pp. 81–85.

[20] M. Shepperd, M. Cartwright, and G. Kadoda, "On building prediction systems for software engineers," *Empirical Software Engineering*, Vol. 5, No. 3, 2000, pp. 175–182.

[21] M. Azzeh, D. Neagu, and P. I. Cowling, "Analogy-based software effort estimation using fuzzy numbers," *Journal of Systems and Software*, Vol. 84, No. 2, 2011, pp. 270–284.

[22] P. Hearty, N. Fenton, D. Marquez, and M. Neil, "Predicting project velocity in XP using a learning dynamic bayesian network model," *IEEE Transactions on Software Engineering*, Vol. 35, No. 1, 2009, pp. 124–137.

[23] E. Miranda, P. Bourque, and A. Abran, "Sizing user stories using paired comparisons," *Information and Software Technology*, Vol. 51, No. 9, 2009, pp. 1327–1337.

[24] K. Hamdan, H. El Khatib, J. Moses, and P. Smith, "A software cost ontology system for assisting estimation of software project effort for use with case-based reasoning," in *Innovations in Information Technology, 2006*. IEEE, 2006, pp. 1–5.

e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 65–78, DOI 10.5277/e-Inf140105

Malicious JavaScript Detection by Features Extraction

Gerardo Canfora*, Francesco Mercaldo*, Corrado Aaron Visaggio*
*Department of Engineering, University of Sannio
canfora@unisannio.it, fmercaldo@unisannio.it, visaggio@unisannio.it

Abstract

In recent years, JavaScript-based attacks have become one of the most common and successful types of attack. Existing techniques for detecting malicious JavaScripts could fail for different reasons. Some techniques are tailored on specific kinds of attacks, and are ineffective for others. Some other techniques require costly computational resources to be implemented. Other techniques could be circumvented with evasion methods. This paper proposes a method for detecting malicious JavaScript code based on five features that capture different characteristics of a script: execution time, external referenced domains and calls to JavaScript functions. Mixing different types of features could result in a more effective detection technique, and overcome the limitations of existing tools created for identifying malicious JavaScript. The experimentation carried out suggests that a combination of these features is able to successfully detect malicious JavaScript code (in the best cases we obtained a precision of 0.979 and a recall of 0.978).

1. Introduction

JavaScript [1] is a scripting language usually embedded in web pages with the aim of creating interactive HTML pages. When a browser downloads a page, it parses, compiles, and executes the script. As with other mobile code schemes, malicious JavaScript programs can take advantage of the fact that they are executed in a foreign environment that contains private and valuable information. As an example, a U.K. researcher developed a technique based on JavaScript timing attacks for stealing information from the victim machine and from the sites the victim visits during the attack [2]. JavaScript code is used by attackers for exploiting vulnerabilities in the user's browser, browser's plugins, or for tricking the victim into clicking on a link hosted by a malicious host. One of the most widespread attacks accomplished with malicious JavaScript is drive-by-download [3, 4], consisting of downloading (and running) malware on the victim's machine. Another example of JavaScript-based attack is represented by scripts that abuse systems resources, such as opening windows that never close or creating a large number of pop-up windows [5].

JavaScript can be exploited for accomplishing web based attacks also with emerging web technologies and standards. As an example, this is happening with Web Workers [6], a technology recently introduced in HTML 5. A Web Worker is a JavaScript process that can perform computational tasks, and send and receive messages to the main process or to other workers. A Web Worker differs from a worker thread in Java or Python in a fundamental aspect of the design: there is no sharing of the state. Web Workers were designed to execute portions of JavaScript code asynchronously, without affecting the performance of the web page. The operations performed by Web Workers are therefore transparent from the point of view of the user who remains unaware of what is happening in the background.

The literature offers many techniques to detect malicious JavaScripts, but all of them show some limitations. Some existing detection solutions leverage previous knowledge about malware, so they could be very effective against well-known attacks, but they are ineffective against zero-day attacks [7]. Another limitation of many detectors of malicious JavaScript code is that they are designed for recognizing specific kinds of attack, thus for circumventing them, attackers usually mix up different attack's types [7]. This paper proposes a method to detect malicious JavaScript that consists of extracting five features from the web page under analysis (WUA in the remaining of the paper), and using them for building a classifier. The main contribution of this method is that the proposed features are independent of the technology used and the attack implemented. So it should be robust against zero-day attacks and JavaScripts which combine different types of attacks.

1.1. Assumptions and Research Questions

The features have been defined on the basis of three assumptions. One assumption is that a malicious website could require more resources than a trusted one. This could be due to the need to iterate several attempts of attacks until at least one succeeds, to executing botnets functions, or to examining and scanning machine resources. Based on this assumption, two features have been identified. The first feature (*avgExecTime*) computes the average execution time of a JavaScript function. As discussed in [8, 9], the malware is expected to be more resource-consuming than a trusted application. The second feature (*maxExecTime*) computes the maximum execution time of JavaScript function.

The second assumption is that a malicious web page generally calls a limited number of JavaScript functions to perform an attack. This could have different justifications, i.e. a malicious code could perform the same type of attacks over and over again with the aim of maximizing the probability of success: this may mean that a reduced number of functions is called many times. Conversely, a benign JavaScript usually exploits more functions to implement the business logic of a web application [10]. One feature has been defined on this assumption (*funcCalls*) that counts the number of function calls done by each JavaScript.

The third assumption is that a JavaScript function can make use of malicious URLs for many purposes, i.e. performing drive-by download attacks or sending data stolen from the victim's machine. The fourth feature (*totalUrl*) counts the total number of the URLs into a JavaScript function, while the fifth feature (*extUrl*) computes the percentage of URLs outside the domain of the WUA.

We build a classifier by using these five features in order to distinguish malicious web applications from trusted ones; the classifier runs six classification algorithms.

The paper poses two research questions:

- RQ1: can the five features be used for discriminating malicious from trusted web pages?
- RQ2: does a combination of the features exist that is more effective than a single feature to distinguish malicious web pages from trusted ones?

The paper proceeds as follows: next section discusses related work; the following section illustrates the proposed method; the fourth section discusses the evaluation, and, finally, conclusions are drawn in the last section.

2. Related Work

A number of approaches have been proposed in the literature to detect malicious web pages. Traditional anti-virus tools use static signatures to match patterns that are commonly found in malicious scripts [11]. As a countermeasure, complex obfuscation techniques have been devised in order to hide malicious code to detectors that scan the code for extracting the signature. Blacklisting of malicious URLs and IPs [7] requires that the user trusts the blacklist provider and entails high costs for management of database, especially for guaranteeing the dependability of the information provided. Malicious websites, in

fact, change frequently the IP addresses especially when they are blacklisted.

Others approaches have been proposed for observing, analysing, and detecting JavaScript attacks in the wild, for example, using high-interaction honeypots [12–14] and low-interaction honeypots [15–17]. High-interaction honey-clients assess the system integrity, by searching for changes to the registry entries, and to the network connections, alteration of the file system, and suspect usage of physical resources. This category of honey-clients is effective, but entails high computational costs: they have to load and run the web application for analysing it, and nowadays websites contain a large number of heavy components. Furthermore, high-interaction honey-clients are ineffective with time-based attacks, and most honey-clients' IPs are blacklisted in the deep web, or they can be identified by an attacker employing CAPTCHAs [7].

Low-interaction honey-clients reproduce automatically the interaction of a human user with the website, within a sandbox. These tools compare the execution trace of the WUA with a sample of signatures: this makes this technique to fail against zero-day attacks.

Different systems have been proposed for off-line analysis of JavaScript code [3, 18–20]. While all these approaches are successful with regard to the malicious code detection, they suffer from a serious weakness: they require a significant time to perform the analysis, which makes them inadequate for protecting users at run-time. Dewald [21] proposes an approach based on a sandbox to analyse JavaScript code by merging different approaches: static analysis of source code, searching forbidden IFrames and dynamic analysis of JavaScript code's behaviour.

Concurrently to these offline approaches, several authors focused on the detection of specific attack types, such as heap-spraying attacks [22, 23] and drive-by downloads [24]. These approaches search for symptoms of certain attacks, for example the presence of shell-code in JavaScript strings. Of course, the main limitation is that such approaches cannot be used for all the threats.

Recent work has combined JavaScript analysis with machine learning techniques for deriving automatic defences. Most notably are the learning-based detection systems Cujo [25], Zozzle [26], and IceShield [27]. They classified malware by using different features, respectively: q-grams from the execution of Javascript, context's attributes obtained from AST and some DOM tree's characteristics. Revolver [28] aims at finding high similarity between the WUA and a sample of known signatures. The authors extract and compare the AST structures of the two JavaScripts. Blanc et al. [29] make use of AST fingerprints for characterizing obfuscating transformations found in malicious JavaScripts. The main limitation of this technique is the high false negatives rate due to the quasi similar subtrees.

Clone detection is a direction explored by some researchers [2, 30], consisting on finding similarity among WUA and known JavaScript fragments. This technique can be effective in many cases but not all, because some attacks can be completely original.

Wang et al. [31] propose a method for blocking JavaScript extensions by intercepting Cross-Platform Component Object Model calls. This method is based on the recognition of patterns of malicious calls; misclassification could occur with this technique so innocent JavaScript extensions could be signaled as malicious. Barua et al. [32] also faced the problem of protecting browsers from JavaScript injections of malicious code by transforming the original and legitimate code with a key. By this way, the injected code is not recognized after the deciphering process and thus detected. This method is applicable only to the code injection attacks. Sayed et al. [33] deal with the problem of detecting sensitive information leakage performed by malicious JavaScript. Their approach relies on a dynamic taint analysis of the web page which identifies those parts of the information flow that could be indicators of a data theft. This method does not apply to those attacks which do not entail sensitive data exfiltration. Schutt et al. [34] propose a method for early identification of threats within javascripts at runtime, by building a classifier which uses the

events produced by the code as features. A relevant weakness of the method is represented by evasion techniques, described by authors in the paper, which are able to decrease the performance of the classification. Tripp et al. [35] substitute concrete values with some specific properties of the document object. This allows for a preliminary analysis of threats within the JavaScript. The method seems to not solve the problem of code injection. Xu and colleagues [36] propose a method which captures some essential characteristics of obfuscated malicious code, based on the analysis of function invocation. The method demonstrated to be effective, but the main limitation is its purpose: it just detects obfuscated (malicious) JavaScripts, but does not recognize other kinds of threats.

Cova et al. [3] make use of a set of features to identify malicious JavaScript including the number and target of redirections, the browser personality and history-based differences, the ratio of string definition and string uses, the number of dynamic code executions and the length of dynamically evaluated code. They proposed an approach based on an anomaly detection system; our approach is similar but different because uses the classification.

Wang et al. [37] combine static analysis and program execution to obtain a call graph using the abstract syntax tree. This could be very effective with attacks that reproduce other attacks (this practice is very common among inexperienced attackers, known also as "script-kiddies") but it is ineffective with zero-day attacks.

Yue et al. [38] focus on two types of insecure practices: insecure JavaScript inclusion and insecure JavaScript dynamic generation. Their work is a measurement study focusing on the counting of URLs, as well as on the counting of the eval() and the document.write() functions.

Techniques known as language-based sandboxing [33, 39–42] aimed at isolating the untrusted JavaScript content from the original webpage. BrowserShield [43], FBJS from Facebook [44], Caja from Google [45], and ADsafe which is widely used by Yahoo [39], are examples of this technique. It is very effective when coping with widget and mashup webpages, but it fails if the web page contains embedded malicious code. A relevant limitation of this technique is that third parties' developers are forced to use the Software Development Kits delivered by sandboxes' producers.

Ismail et al. [46] developed a method which detects XSS attacks with a proxy that analyses the HTTP traffic exchanged between the client (web browser) and the web application. This approach has two main limitations. Firstly, it only detects reflected XSS, also known as non-persistent XSS, where the attack is performed through a single request and response. Second, the proxy is a possible bottleneck for performance as it has to analysing all the requests and responses transmitted between the client and the server. In [47], Kirda et al. propose a web proxy that analyses dynamically generated links in web pages and compares those links with a set of filtering rules for deciding if they are trusted or not. The authors leverage a set of heuristics to generate filtering rules and then leave the user to allow or disallow suspicious links. A drawback of this approach if that involving users might negatively affect their browsing experience.

3. The Proposed Method

Our method extracts three classes of features from a web application: JavaScript execution time, calls to JavaScript functions and URLs referred by the WUA's JavaScript.

To gather the required information we use:

1. dynamic analysis, for collecting information about the execution time of JavaScript code within the WUA and the called functions;
2. static analysis, to identifying all the URLs referred in the WUA within and outside the scope of the JavaScript.

The first feature computes the average execution time required by JavaScript function:

$$avgExecTime = \frac{1}{n}\sum_{k=1}^{n} t_i$$

where: t_i is the execution time of the i-th JavaScript function, and n is the number of JavaScript functions in the WUA.

The second feature computes the maximum execution time of all JavaScript functions:

$$maxExecTime = \max(t_i)$$

where t_i is the execution time of the i-th JavaScript function in the WUA.

The third feature computes the number of functions calls made by the JavaScript code:

$$funcCalls = \sum_{i=1}^{n} c_i$$

where n is the is the number of JavaScript functions in the WUA, and c_i is the number of calls for the i-th function.

The fourth feature computes the total number of URLs retrieved in a web page:

$$totalUrl = \sum_{i=0}^{m} u_i$$

where: u_i is the number of times the i-th URL is called by a JavaScript function and m is the number of different URLs referenced within the WUA.

The fifth feature computes the percentage of external URLs referenced within the JavaScript:

$$extUrl = \frac{\sum_{k=0}^{j} u_k}{\sum_{i=0}^{m} u_i} * 100$$

where: u_k is the number of times the k-th URL is called by a JavaScript function, for j different external URLs referenced within the JavaScript, while u_i is the number of times the i-th URL is called by a JavaScript function, for m total URLs referenced within the JavaScript.

We used these features for building several classifiers. Specifically, six different algorithms were run for the classification, by using the Weka suite [48]: J48, LADTree, NBTree, RandomForest, RandomTree and RepTree.

3.1. Implementation

The features extracted from the WUA by dynamic analysis were:
- execution time;
- calls to javascript function;
- number of function calls made by the javascript code.

The features extracted from the WUA by static analysis were:
- number of URLs retrieved in the WUA;
- URLs referenced within the WUA.

The dynamic features were captured with Chrome developer [49], a publicly available tool for profiling Web Applications. Each WUA was opened with a Chrome browser for a fixed time of 50 seconds, and the Chrome developer tool performed a default exploration of the WUA, mimicking user interaction and collecting the data with the Mouse and Keyboard Recorder tool [50], a software able to record all mouse and keyboard actions, and then repeat all the actions accurately.

The static analysis aimed at capturing all the URLs referenced in the JavaScript files included in the WUA. URLs were recognized through regular expressions: when an URL was found, it was compared with the domain of the WUA: if the URL's domain was different from the WUA's domain, it was tagged as an external URL.

We have created a script to automate the data extraction process. The script takes as input a list of URLs to analyse and perform the following steps:
- step 1: start the Chrome browser;
- step 2: start the Chrome Dev Tools on the panel Profiles;
- step 3: start the tool for profiling;
- step 4: confirm the inserted URL as parameter to the browser and waiting for the time required to collect profiling data;
- step 5: stop profiling;
- step 6: save data profiling in the file system;
- step 7: close Chrome;
- step 8: start the Java program to parse profiling saved;
- step 9: extract the set of dynamic features;
- step 10: save source code of the WUA;

– step 11: extract the set of static features;
– step 12: save the values of the features extracted into a database.

The dynamic features of the WUA are extracted from the log obtained with the profiling.

4. Experimentation

The aim of the experimentation is to evaluate the effectiveness of the proposed features, expressed through the research questions RQ1 and RQ2.

The experimental sample included a set of 5000 websites classified as "malicious", while the control sample included a set of 5000 websites classified as "trusted".

The trusted samples includes URLs belonging to a number of categories, in order to make the results of experimentation independent of the type of web-site: Audio-video, Banking, Cooking, E-commerce, Education, Gardening, Government, Medical, Search Engines, News, Newspapers, Shopping, Sport News, Weather.

As done by other authors [33] the trusted URLs were retrieved from the repository "Alexa" [51], which is an index of the most visited websites. For the analysis, the top ranked websites for each category were selected, which were mostly official websites of well-known organizations. In order to have a stronger guarantee that the websites were not phishing websites or did not contain threats, we submitted the URLs to a web-based engine, VirusTotal [52], which checks the reliability of the web sites, by using anti-malware software and by searching the web site URLs and IPs in different blacklists of well-known antivirus companies.

The "malicious" sample was built from the repository hpHosts [53], which provides a classification of websites containing threats sorted by the type of the malicious attack they perform. Similarly to the trusted sample, websites belonging to different threat's type were chosen, in order to make the results of the analysis independent of the type of threat. We retrieved URLs from various categories: sites engaged in malware distribution, in selling fraudulent applications, in the use of misleading marketing tactics and browser hijacking, and sites engaged in the exploitation of browser and OS vulnerabilities. For each URL belonging to the two samples, we extracted the five features defined in section 3.

Two kinds of analysis were performed on data: hypothesis testing and classification. The test of hypothesis was aimed at understanding whether the two samples show a statistically significant difference for the five features. The features that yield the most relevant differences between the two samples were then used for the classification.

We tested the following null hypothesis:

$\mathbf{H_0}$: malware and trusted websites have similar values of the proposed features.

The $\mathbf{H_0}$ states that, given the i-th feature f_i, if f_{iT} denotes the value of the feature f_i measured on a trusted web site, and f_{iM} denoted the value of the same feature measured on a malicious web site:

$$\sigma(f_{iT}) = \sigma(f_{iM}) \text{ for } i = 1, \ldots, 5$$

being $\sigma(f_i)$ the means of the (control or experimental) sample for the feature f_i.

The null hypothesis was tested with Mann-Whitney (with the p-level fixed to 0.05) and with Kolmogorov-Smirnov Test (with the p-level fixed to 0.05). Two different tests of hypotheses were performed in order to have a stronger internal validity since the purpose is to establish that the two samples (trusted and malicious websites) do not belong to the same distribution.

The classification analysis was aimed at assessing whether the features where able to correctly classify malicious and trusted WUA. Six algorithms of classification were used: J48, LadTree, NBTree, RandomForest, RandomTree, RepTree. Similarly to hypothesis testing, different algorithms for classification were used for strengthening the internal validity.

These algorithms were first applied to each of the five features and then to the groups of features. As a matter of fact, in many cases a classification is more effective if based on groups of features rather than a single feature.

4.1. Analysis of Data

Figure 1 illustrates the boxplots of each feature. Features *avgExecTime*, *maxExecTime* and *funcCalls* exhibit a greater gap between the distributions of the two samples.

Features *totalUrl*, and *extUrl* do not exhibit an evident difference between trusted and malicious samples. We recall here that *totalUrl* counts the total number of URLs in the JavaScript, while *extUrl* is the percentage of URLs outside the WUA domain contained in the script. A possible reason why these two features are similar for both the samples is that trusted websites may include external URLs due to external banners or to external legal functions and components that the JavaScript needs for execution (images, flash animation, functions of other websites that the author of the WUA needs to recall). Using external resources in a malicious JavaScript is not so uncommon: examples are drive by download and session hijacking. External resources can be used when the attacker injects a malicious web page into a benign website and needs to lead the website user to click on a malicious link (which can not be part of the benign injected website). We expect that extending this analysis to the complete WUA (not limited to JavaScript code) could produce different results: this goal will be included in the future work.

On the contrary, features *avgExecTime*, *maxExecTime* and *funcCalls* seem to be more effective in distinguishing malicious from trusted websites, which supports our assumptions.

Malware requires more execution time than trusted script code because of many reasons (*avgExecTime*, *maxExecTime*). Malware may require more computational time for performing many attempts of the attack till it succeeds. Examples may be: complete memory scanning, alteration of parameters, and resources occupation.

Some kinds of malware aim at obtaining the control of the victim machine and the command centre, once infected the victim, could occupy computational resources of the victim for sending and executing remote commands. Furthermore, some other kinds of malware could require time because they activate secondary tasks like downloading and running additional malware, as in the case of drive-by-download.

The feature *funcCalls* suggests that trusted websites have a larger number of functions called

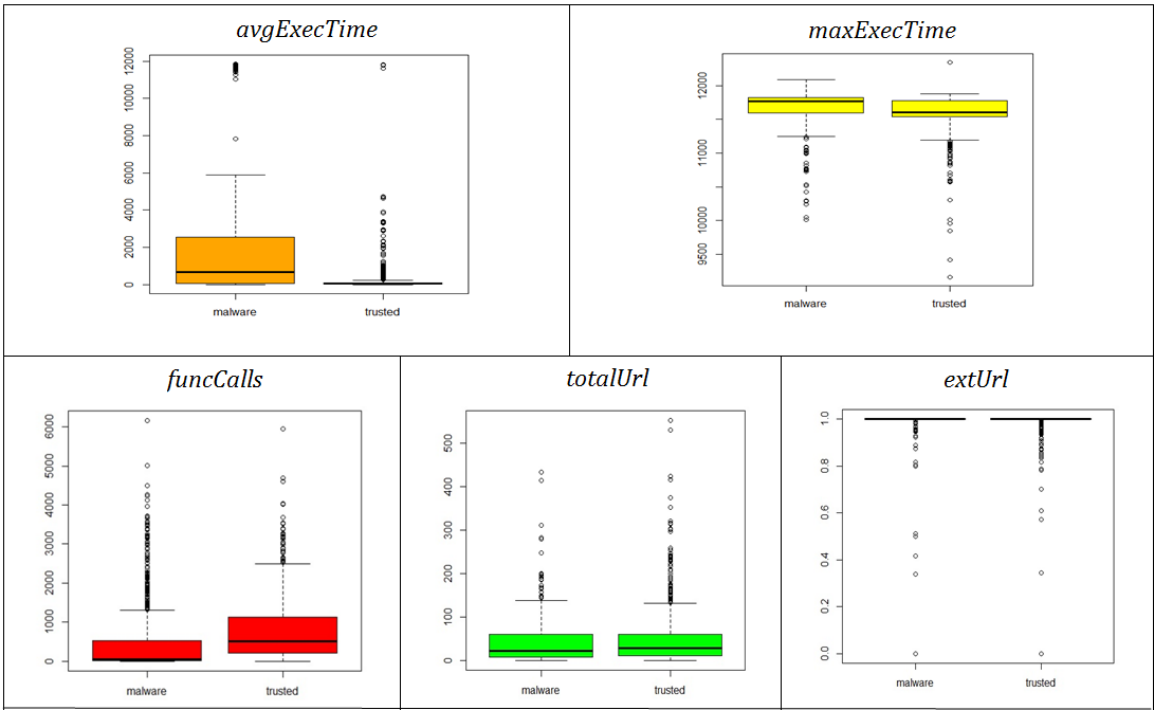


Figure 1. Boxplots of features

or function-calls. Our hypothesis for explaining this finding is that trusted websites need calling many functions for executing the business logic of the website, like data field controls, third party functions such as digital payment, elaborations of user inputs, and so on. On the contrary, malicious websites have the only goal to perform the attack, so they are poor of business functions with the only exception for the payload to execute. Instead, they perform their attack at regular intervals; for this reason malicious WUAs show a higher value of *avgExecTime* and *maxExecTime* with respect to the trusted ones.

In order to optimize the client-server interaction, the trusted website could have many functions but usually with a low computational time, in order to avoid impacting on the website usability. This allows, for example, performing controls, such as data input validation, on the client side and sending to the server only valid data.

The hypothesis test produced evidence that the features have different distributions in the control and experimental sample, as shown in Table 1.

Summing up, the null hypothesis can be rejected for the features *avgExecTime*, *maxExecTime*, *funcCalls*, *totalUrl* and *extUrl*.

With regard to classification, the training set T consisted of a set of labelled web applications (WUA, l) where the label $l \in \{trusted, malicious\}$. For each WUA we built a feature vector $F \in R^y$, where y is the number of the features used in training phase ($1 \leq y \leq 5$). To answer to RQ1 we performed five different classifications each with a single feature ($y = 1$), while for RQ2 we performed three classifications with $2 \leq y \leq 5$).

We used k-fold cross-validation: the dataset was randomly partitioned into k subsets of data. A single subsets of data was retained as the validation data for testing the model, while the remaining $k - 1$ subsets was used as training data. We repeated the process k-times, each of the k subsets of data was used once as validation data. To obtain a single estimate we computed the average of the k results from the folds.

Specifically, we performed a 10-fold cross validation. Results are shown in Table 2. The rows represent the features, while the columns represent the values of the three metrics used to evaluate the classification results (precision, recall and roc-area) for the recognition of malware and trusted samples. The *Recall* has been computed as the proportion of examples that were assigned to class X, among all examples that truly belong to the class, i.e. how much part of the class was captured. The *Recall* is defined as:

$$Recall = \frac{tp}{tp + fn}$$

where tp indicates the number of true positives and fn is the number of false negatives.

The Precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class, i.e.:

$$Precision = \frac{tp}{tp + fp}$$

where fp indicates the number of false positives.

The Roc Area is the area under the ROC curve (AUC), it is defined as the probability that a randomly chosen positive instance is ranked above randomly chosen negative one. The classification analysis with the single features suggests several considerations.

With regards to the recall:

– Generally the classification of malicious websites is more precise than the classification of trusted websites.

Table 1. Results of the test of the null hypothesis H_0

Variable	Mann–Whitney	Kolmogorov–Smirnov
avgExecTime	0.000000	$p < .001$
maxExecTime	0.000000	$p < .001$
funcCalls	0.000000	$p < .001$
totalUrl	0.000000	$p < .001$
extUrl	0.002233	$p < .001$

Table 2. Precision, Recall and RocArea obtained by classifying Malicious and Trusted dataset, using the single features of the model, with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree

Features	**Algorithm**	**Precision**		**Recall**		**RocArea**	
		Malware	Trusted	Malware	Trusted	Malware	Trusted
avgExecTime	J48	0.872	0.688	0.597	0.898	0.741	0.741
	LADTree	0.836	0.691	0.606	0.881	0.789	0.789
	NBTree	0.872	0.686	0.59	0.895	0.771	0.771
	RandomForest	0.744	0.758	0.759	0.744	0.762	0.762
	RandomTree	0.773	0.762	0.762	0.763	0.766	0.766
	RepTree	0.971	0.735	0.704	0.819	0.725	0.725
maxExecTime	J48	0.657	0.635	0.606	0.684	0.675	0.675
	LADTree	0.638	0.663	0.69	0.606	0.691	0.691
	NBTree	0.672	0.634	0.587	0.713	0.654	0.654
	RandomForest	0.683	0.703	0.718	0.667	0.775	0.775
	RandomTree	0.678	0.708	0.731	0.653	0.782	0.782
	RepTree	0.663	0.686	0.706	0.641	0.724	0.724
funcCalls	J48	0.928	0.677	0.582	0.876	0.722	0.722
	LADTree	0.816	0.678	0.587	0.868	0.75	0.75
	NBTree	0.824	0.677	0.582	0.896	0.727	0.727
	RandomForest	0.784	0.719	0.629	0.772	0.672	0.672
	RandomTree	0.782	0.683	0.646	0.765	0.696	0.696
	RepTree	0.763	0.675	0.686	0.788	0.787	0.787
totalUrl	J48	0.615	0.552	0.381	0.762	0.603	0.603
	LADTree	0.566	0.555	0.511	0.609	0.6	0.6
	NBTree	0.607	0.533	0.284	0.717	0.565	0.565
	RandomForest	0.624	0.653	0.689	0.585	0.691	0.691
	RandomTree	0.619	0.655	0.7	0.57	0.691	0.691
	RepTree	0.617	0.609	0.595	0.631	0.66	0.66
extUrl	J48	0.514	0.7	0.993	0.061	0.527	0.527
	LADTree	0.51	0.704	0.972	0.066	0.512	0.512
	NBTree	0.514	0.716	0.992	0.062	0.527	0.527
	RandomForest	0.513	0.513	0.992	0.061	0.532	0.532
	RandomTree	0.514	0.787	0.993	0.061	0.527	0.527
	RepTree	0.514	0.597	0.593	0.561	0.527	0.527

This could be due to the fact that some trusted websites have values for the features comparable with the ones measured for malicious ones. This is evident by looking at the boxplots (figure 1), which show an area of overlapping between the boxplots of the trusted and malicious websites. The problem is that some trusted websites could have values comparable with the malware while others do not. As a matter of fact, some trusted WUAs can contain more business functions than other ones, and require more client machine resources, and so on. This depends on the specific business goals of each trusted website. And, consequently, on the type and numbers of the functions that must be implemented for supporting the business goals. Except for *funcCalls*, the trusted websites' sample include a greater number of outliers than the malware sample, which is the main cause of the misclassifications of trusted websites and supports our explanation.

– The feature *extUrl* is the best in terms of recall regarding the malicious websites; in fact, its value is 0.993 using the algo-

rithms of classification J48 and RandomTree. This feature is able to reduce the false negatives in malicious detection because external URLs are commonly used by malicious websites, for the reasons previously discussed.

– Regarding the recall inherent the recognition of the trusted websites, the best feature is *avgExecTime* (recall is 0.898 with the J48 classification algorithm). This confirms the conjecture that malicious scripts tend to be more resource demanding than trusted ones.

With regards to the precision:

– The features *avgExecTime* and *funcCalls* are the best for the detection of the malicious JavaScript, with values, respectively of 0.971 (with the algorithm RepTree) and 0.928 (with the algorithm J48). This strengthens the conjecture that trusted websites make use of less computational time and a larger number of functions than malicious websites.
– The precision in the classification of sites categorized as trusted shows the maximum value 0.787 (classification of the feature *extUrl* with the algorithm RandomTree). This value is largely unsatisfactory, and it will be improved by using combinations of features, as discussed later in this section.

With regards to the roc area:

– The performances of all the algorithms are pretty the same for malware and trusted applications.
– The feature *avgExecTime* presents the maximum roc-area value equal to 0.789 with LADTree algorithm. Reasons have been discussed previously, even if it cannot be considered a good value.

In order to make the classification more effective, we run the classification algorithms by using groups of features. The first group includes the features *avgExecTime* and *funcCalls*, while the second includes *avgExecTime*, *funcCalls*, and *extUrl*. Finally, the last group is made up of all the five features extracted. The groups were made on the basis of the classification results of individual features, in order to improve both the precision and the recall of the classification.

avgExecTime and *funcCalls* were the best in class, so we grouped them together. In particular, these features were grouped together in order to obtain the maximum precision value for detecting malicious web applications.

avgExecTime, *funcCalls*, and *extUrl* were grouped together in order to obtain the maximum precision value in the detection of trusted applications. We excluded *maxExecTime* and *totalUrl* from the second phase of classification, because they produced the worst results in the first phase of classification.

The classification of the groups of features confirms (shown in Table 3) our expectations. The first set of features, *avgExecTime* and *funcCalls*, presents the maximum precision regarding malicious websites, corresponding to 0.982 with the classification algorithm J48, while in the detection of the trusted web sites the precision is 0.841 with the classification algorithm REPTree. Compared to the individual features we have therefore an improvement, in fact *avgExecTime* had a precision of 0.971 while *funcCalls* showed a precision 0.928 in the recognition of malware websites. The recall for malicious websites is 0.873 with the classification algorithm J48, while for trusted sites it is 0.897 with the classification algorithm NBTree. With respect to the recognition of malicious websites we have registered an improvement, as with individual features the obtained values were respectively 0.762 (*avgExecTime*) and 0.686 (*funcCalls*). With respect to the trusted websites, the situation is pretty similar, as the values of single features were 0.898 (*avgExecTime*) and 0.896 (*funcCalls*), i.e. slightly greater.

The second group (*avgExecTime*, *funcCalls*, *extUrl*) is very close to the first group (two values are slightly higher and two are slightly lower), but precision and recall are higher than the second group.

We can conclude that the best classification is based on

– *avgExecTime*, *funcCalls*, i.e. the average execution time (*avgExecTime*) and the cumulative number of function calls done by each portion of JavaScript code (*funcCalls*);

Table 3. Precision, Recall and RocArea obtained by classifying Malicious and Trusted dataset, using the three groups of features of the model, with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree

Features	**Algorithm**	**Precision**		**Recall**		**RocArea**	
		Malware	Trusted	Malware	Trusted	Malware	Trusted
	J48	0.982	0.823	0.873	0.88	0.888	0.888
	LADTree	0.87	0.801	0.784	0.873	0.872	0.857
avgExecTime	NBTree	0.848	0.686	0.59	0.897	0.779	0.779
funcCalls	RandomForest	0.84	0.825	0.815	0.797	0.985	0.985
	RandomTree	0.824	0.818	0.779	0.768	0.977	0.977
	RepTree	0.871	0.841	0.824	0.856	0.913	0.913
	J48	0.873	0.842	0.835	0.879	0.885	0.885
	LADTree	0.86	0.801	0.784	0.873	0.857	0.857
avgExecTime	NBTree	0.848	0.69	0.599	0.893	0.789	0.789
funcCalls	RandomForest	0.969	0.978	0.978	0.969	0.985	0.985
extUrl	RandomTree	0.97	0.979	0.98	0.97	0.979	0.979
	RepTree	0.867	0.852	0.849	0.87	0.918	0.918
	J48	0.875	0.878	0.879	0.874	0.922	0.922
avgExecTime	LADTree	0.858	0.804	0.788	0.87	0.858	0.858
maxExecTime	NBTree	0.847	0.72	0.657	0.881	0.827	0.827
funcCalls	RandomForest	0.979	0.984	0.985	0.979	0.992	0.992
totalUrl	RandomTree	0.982	0.978	0.978	0.982	0.98	0.98
extUrl	RepTree	0.877	0.871	0.87	0.879	0.927	0.927

– *avgExecTime*, *funcCalls*, *extUrl*, i.e. the set of the features of the first group classified along with the percentage of external domain URLs that do not belong to the Web Application's domain (*extUrl*).

Although the proposed features show to be effective in detecting malicious javascript, misclassification occurs however. The explanation maybe the following: each feature represents an indicator of the possibility that the JavaScript is malicious, rather than offering the certainty. The fact that in average a malicious JavaScript requires a longer execution time (as shown by boxplots) does not mean that all the benign JavaScripts require a small execution time (as outliers in boxplots show). Many payloads contained in malicious javascript entail a long time to be executed, but also some business logic of benign javascripts may require long time to be executed. For instance, a benign javascript may contain a multimedia file. The same explanation applies to justify the presence of misclassification for all the other features. Benign files could have a smaller fragmentation because they have a simpler business logic or because of the style of the programmer who has written the code.

Finally, the number of the external URLs may be high in a benign websites for several reasons: the benign websites make use of many resources or services hosted in other websites, or it has many advertisement links in its pages.

5. Conclusion and Future Work

In this paper we propose a method for detecting malicious websites that uses a classification based on five features.

Current detection's techniques usually fail against zero-day attacks and websites that merge several techniques. The proposed method should overcome these limitations, since its independent of the implementation of the attack and the type of the attack.

The selected features, combining static and dynamic analysis, respectively compute the average and maximum execution time of a JavaScript function, the number of functions invoked by the JavaScript code, and finally the number and

the percentage of the URLs contained in the JavaScript code, but that are outside the domain of the WUA.

The analysis of data collected by analysing a sample of 5000 trusted and 5000 untrusted websites demonstrated that considering groups of features for classifications, rather than single features, produces better performances. As matter of fact the group (*avgExecTime* and *funcCalls*) and the group (*avgExecTime*, *funcCalls*, *extUrl*) produce high values of precision and recall, both for the recognition of malicious websites, and for trusted websites. Regarding to the second group (*avgExecTime*, *funcCalls*, *extUrl*), the precision is 0.979 for malware websites and 0.969 for trusted ones. The recall is 0.978 for malware websites and 0.969 for trusted ones.

In summary, the two groups of features seem effective for detecting current malicious JavaScripts.

Possible evasion techniques that attackers can assume against this detection method are the following.

Concerning *avgExecTime* and *funcCalls*, the attacker should reduce the time of scripts execution and improve the fragmentation of the code. The first workaround is very difficult to implement, because the large amount of time is often a needed condition of the attacks performed. Improving the fragmentation of code is possible, but as the attacker should produce a number of functions similar to a typical trusted website, the required effort could make very expensive the development of the malicious website, and this could be discouraging. As shown in the boxplot (figure 1), the gap to fill is rather large. Our opinion is that *extUrl* is the weakest feature and so the easiest to evade, but it must be considered that in the group of features (*avgExecTime*, *funcCalls*, *extUrl*) the strength of the other ones may compensate its weakness.

Obfuscation is an evasion technique that could be effective especially with regards to *funcCalls*, *totalUrl* and *extUrl* features; future works will address this problem by studying: i) the impact of obfuscated JavaScript on the classification performances of our method; and ii) de-obfuscation methods to precisely calculate these features. Many benign websites may make use of external libraries which are highly time-consuming. In a future work we will investigate the possibility to recognize these time-consuming external libraries in order to exclude them from the computation of the feature. Additionally we plan to enforce the reliability of our findings by extending the experimentation to a larger sample, in order to enforce the external validity. Another improvement of our method consists of extending the search of URLs to the complete WUA, and not limiting it to the JavaScript scope.

References

[1] D. Flanagan, *JavaScript: The Definitive Guide*, 4th ed., O'Reilly Media, 2001. [Online]. http://shop.oreilly.com/product/9780596000486.do

[2] "Javascript and timing attacks used to steal browser data," Blackhat 2013, last visit 19th June 2014. [Online]. http://threatpost.com/JavaScript-and-timing-attacks-used-to-steal-browser-data/101559

[3] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proc. of the International World Wide Web Conference (WWW)*, 2010, pp. 281–290.

[4] C. Eilers, *HTML5 Security*. Developer Press, 2013.

[5] O. Hallaraker and G. Vigna, "Detecting malicious JavaScript code in Mozilla," in *Proceedings of the 10th IEEE International Conference of Engineering of Complex Computer System*, 2005, pp. 85–94.

[6] "Web workers, W3C candidate recommendation," 2012, last visit 19th June 2014. [Online]. http://www.w3.org/TR/workers/

[7] B. Eshete, "Effective analysis, characterization, and detection of malicious web page," in *Proceedings of the 22nd International Conference on World Wide Web companion*. International World Wide Web Conferences Steering Committee, 2013, pp. 355–360.

[8] L. Martignoni, R. Paleari, and D. Bruschi, "A framework for behavior-based malware analysis in the cloud," in *Proceedings of the 5th International Conference on Information Systems Security*, 2009, pp. 178–192.

[9] M. F. Zolkipli and A. Jantan, "An approach for malware behavior identification and classifica-

tion," in *Proceedings of International Conference of Computer Research and Development*, 2011.

[10] C. Ardito, P. Buono, D. Caivano, M. Costabile, and R. Lanzilotti, "Investigating and promoting UX practice in industry: An experimental study," *International Journal of Human-Computer Studies*, Vol. 72, No. 6, 2014, pp. 542–551.

[11] "ClamAV. Clam AntiVirus," last visit 19th June 2014. [Online]. http://clamav.net

[12] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proc. of USENIX Security Symposium*, 2008.

[13] C. Seifert and R. Steenson, "Capture honeypot client (capture hpc)," Victoria University of Wellington, NZ, 2006. [Online]. https://projects.honeynet.org/capture-hpc

[14] Y. M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowsk, S. Chen, and S. T. King, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2006.

[15] A. Büscher, M. Meier, and R. Benzmüller, "Throwing a monkeywrench into web attackers plans," in *Proc. of Communications and Multimedia Security (CMS)*, 2010, pp. 28–39.

[16] A. Ikinci, T. Holz, and F. Freiling, "Monkey-Spider: Detecting malicious websites with low-interaction honeyclients," in *In Proceedings of Sicherheit, Schutz und Zuverlässigkeit*, 2008, pp. 891–898.

[17] J. Nazario, "A virtual client honeypot," in *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[18] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proc. of the International World Wide Web Conference (WWW)*, 2011, pp. 197–206.

[19] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin, "ZDVUE: prioritization of JavaScript attacks to discover new vulnerabilities," in *Proceedings of the Fourth ACM Workshop on Artificial Intelligence and Security (AISEC 2011)*, 2011, pp. 637–652.

[20] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," Microsoft Research, Tech. Rep. MSR-TR-2011-94, 2011. [Online]. http://research.microsoft.com/pubs/152601/rozzle-tr-10-25-2011.pdf

[21] A. Dewald, T. Holz, and F. Freiling, "ADSandbox: sandboxing JavaScript to fight malicious websites," in *Proceedingsof the 2010 ACM Symposium on Applied Computing (SAC '10)*, 2010, pp. 1859–1864.

[22] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *In Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009, pp. 88–106.

[23] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proc. of USENIX Security Symposium*, 2009.

[24] L. Lu, V. Yegneswaran, P. A. Porras, and W. Lee, "BLADE: an attack-agnostic approach for preventing drive-by malware infections," in *Proc. of Conference on Computer and Communications Security (CCS)*, 2010, pp. 440–450.

[25] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 31–39.

[26] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser JavaScript malware detection," in *Proc. of USENIX Security Symposium*, 2010, pp. 3–3.

[27] M. Heiderich, T. Frosch, and T. Holz, "IceShield: detection and mitigiation of malicious websites with a frozen dom," in *Proceedings of Recent Adances in Intrusion Detection (RAID)*, 2011, pp. 281–300.

[28] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegle, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proceedings of the 22nd USENIX conference on Security*, 2013, pp. 637–652.

[29] G. Blanc, D. Miyamoto, M. Akiyama, and Y. Kadobayashi, "Characterizing obfuscated JavaScript using abstract syntax trees: Experimenting with malicious scripts," in *Proceedings of International Conference of Advanced Information Networking and Applications Workshops*, 2012.

[30] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing Queen's University at Kingston, Ontario, TR 2007-541, 2007.

[31] P. Wang, L. Wang, J. Xiang, P. Liu, N. Gao, and J. Jing, "MJBlocker: A lightweight and run-time malicious JavaScript extensions blocker," in *Proceedings of International Conference on Software Security and Reliability*, 2013.

[32] A. Barua, M. Zulkernine, and K. Weldemariam, "Protecting web browser extension from JavaScript injection attacks," in *Proceedings of International Conference of Complex Computer Systems*, 2013.

[33] B. Sayed, I. Traore, and A. Abdelhalim, "Detection and mitigation of malicious JavaScript using information flow control," in *Proceedings of Twelfth Annual Conference on Privacy, Security and Trust (PST)*, 2014.

[34] K. Schutt, M. Kloft, A. Bikadorov, and K. Rieck, "Early detection of malicious behaviour in JavaScript code," in *Proceedings of AISec 2012*, 2012.

[35] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web JavaScript code via dynamic partial evaluation," in *Proceedings of International Symposium on Software Testing and Analysis*, 2014.

[36] W. Xu, F. Zhang, and S. Zhu, "JStill: Mostly static detection of obfuscated malicious JavaScript code," in *Proceedings of International Conference on Data and Application Security and Privacy*, 2013.

[37] Q. Wang, J. Zhou, Y. Chen, Y. Zhang, and J.Zhao, "Extracting URLs from JavaScript via program analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 627–630.

[38] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 961–970.

[39] J. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: type-based verification of JavaScript sasndboxing," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[40] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *ECOOP 2010-Object-Oriented*, 2011, pp. 1–25.

[41] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure JavaScript substes," in *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[42] A. Taly, U. Erlingsson, J. Mitchell, M. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript apis," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 363–379.

[43] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web*, Vol. 1, No. 3, 2007.

[44] "Facebook SDK for JavaScript," last visit 13th October 2014. [Online]. https://developers.facebook.com/docs/javascript

[45] "Google Caja," last visit 13th October 2014. [Online]. https://developers.google.com/caja/

[46] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, Vol. 2, 2014.

[47] E. Kirda, C.Kruegel, G. Vigna, and N. Jovanic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 330–337.

[48] "Weka 3: Data mining software in Java," last visit 19th June 2014. [Online]. http://www.cs.waikato.ac.nz/ml/weka/

[49] "Chrome DevTools overview," last visit 19th June 2014. [Online]. https://developers.google.com/chrome-developer-tools/

[50] "Robot Soft - mouse and keyboard recorder," last visit 13th October 2014. [Online]. http://www.robot-soft.com/

[51] "Actionable analytics for the web," last visit 19th June 2014. [Online]. http://www.alexa.com/

[52] "VirusTotal," last visit 19th June 2014. [Online]. https://www.virustotal.com/

[53] "hpHosts onliine," last visit 19th June 2014. [Online]. http://www.hosts-file.net/

e-Informatica Software Engineering Journal (EISEJ) is an international, open access, peer-reviewed journal that concerns theoretical and practical issues pertaining development of software systems. Our aim is to focus on experimentation and data mining in software engineering.

The purpose of **e-Informatica Software Engineering Journal** is to publish original and significant results in all areas of software engineering research.

The scope of **e-Informatica Software Engineering Journal** includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular stress is laid on empirical evaluation.

e-Informatica Software Engineering Journal is published online and in hard copy form. The online version (which is our primary version) is open access, which means it is available at no charge to the public.

Topics of interest include, but are not restricted to:

— Software requirements engineering and modeling
— Software architectures and design
— Software components and reuse
— Software testing, analysis and verification
— Agile software development methodologies and practices
— Model driven development
— Software quality
— Software measurement and metrics
— Reverse engineering and software maintenance
— Empirical and experimental studies in software engineering (incl. replications)
— Evidence based software engineering
— Systematic reviews and mapping studies
— Meta-analyses
— Object-oriented software development
— Aspect-oriented software development
— Software tools, containers, frameworks and development environments
— Formal methods in software engineering.
— Internet software systems development
— Dependability of software systems
— Human-computer interaction
— AI and knowledge based software engineering
— Data mining in software engineering
— Prediction models in software engineering
— Tools for software researchers or practitioners
— Project management
— Software products and process improvement and measurement programs
— Process maturity models
— Search-based software engineering

Papers can be rejected administratively without undergoing review for a variety reasons, such as being out of scope, being badly presented to such an extent as to prevent review, missing some fundamental components of research such as the articulation of a research problem, a clear statement of the contribution and research methods via structured abstract or the evaluation of the proposed solution (empirical evaluation is strongly suggested).

The submissions will be accepted for publication on the base of positive reviews done by international Editorial Board and external reviewers.

English is the only accepted publication language. To submit an article please enter our online paper submission site.

Subsequent issues of the journal will appear continuously according to the reviewed and accepted submissions.

http://www.e-informatyka.pl/wiki/e-Informatica

e-Informatica

ISSN 1897-7979