Utilizing static and dynamic software analysis to aid cost estimation, software visualization, and test quality management

Gergő Balogh
Department of Software Engineering
University of Szeged

Szeged, 2020

Supervisor:
Dr. Árpád Beszédes

Summary of the Ph.D. thesis submitted for the degree of
Doctor of Philosophy of the University of Szeged

University of Szeged
PhD School in Computer Science

Introduction

Software development utilizes several abstract concepts. There are several properties of these which are well-known for general audiences. They are usually strongly related to their end-user features. Some of them describe the product itself, while others capture the implementation process. For example, users able to asses the number of available features and the amount of time (and eventually money) required to implement them. The layout of the GUI (and other similar attributes) is usually just the publicly visible surface of the enormous set of software system properties.

All the stakeholders strive for the common goal of creating or improving the software. However, their objectives may differ from one another's. For example, one of the primary motivations of managers is to deliver new features as soon as possible, while developers would like to enforce certain technical (de facto) standards, which usually increase development time, but later decrease maintenance costs.

The responsibility of researchers is more pronounced because stakeholders should rely on methodologies devised by these scientists to address the above-detailed issues. The new and improved methods (and the tools based on them) should take into account the complexity of the software systems and their rapid changes. In practice, it means we have to find an efficient way to navigate in the system and locate those parts that could yield some (unexpected) failures. Automatic or semi-automatic corrective and preventive techniques (like refactoring) can presumably improve the development time by reducing the amount of manual labor required to inspect and fix these error-prone structures. Finally, the efficiency of resource management and task assignment should be improved by examining adequate measurements of the development process.

Challenges

My responsibility as a researcher is to aid the stakeholders in achieving their common goals without hindering their objectives: by improving their processes and tools, and by helping juniors achieve their full potential. Keeping the above mentioned duties in mind, I was faced with some general challenges during my research, which I had to address. These challenges are unfolded in the list below.

Challenge 1: Software comprehension. *Students and newcomers have to get familiar with the large, previously created code base and understand*

abstract concepts of software development, while senior developers and testers have to navigate efficiently in a usually highly complex software structure.

Challenge 2: Fault localization. *Developers and testers have to locate those parts of the software and test suite which could cause failures, i.e. those parts that violate well-established principles.*

Challenge 3: Cost estimation. *The manager has to monitor the properties of the development process to improve it, while quantitative measurements are not able to capture intellectual and creative work, like software development.*

Challenge 4: Program structure analysis. *Software development research often utilizes a comparison of various interconnected entities. For example, software analysis frequently relies on test-code connections, which are not always noted explicitly.*

Thesis Points

The thesis result statements have been grouped into three major thesis points, where the author's contributions are clearly shown. The relation between thesis points and supporting publications are shown in table 1, where I list each relevant sub-thesis points per paper.

	Thesis point 1	**Thesis point 2**	**Thesis point 3**
[7]	1.1, 1.2		
[10]	1.3		
[1]	1.4		
[5]		2.1	
[4]		2.1, 2.3	
[3]		2.2	
[6]		2.3	
[8]		2.4	
[9]			3.1, 3.2
[11]			3.2, 3.3
[12]			3.2, 3.3
[2][1]			3.3

Table 1: Thesis contributions and supporting publications

T1: Measuring, predicting, and comparing the productivity of developer teams. My main goal was to address the two major issues during

[1]This paper was submitted for publication, but it was not accepted yet

software development from the manager's point of view: cost prediction and wasted effort handling.An overview of our research phases are shown on fig. 1.

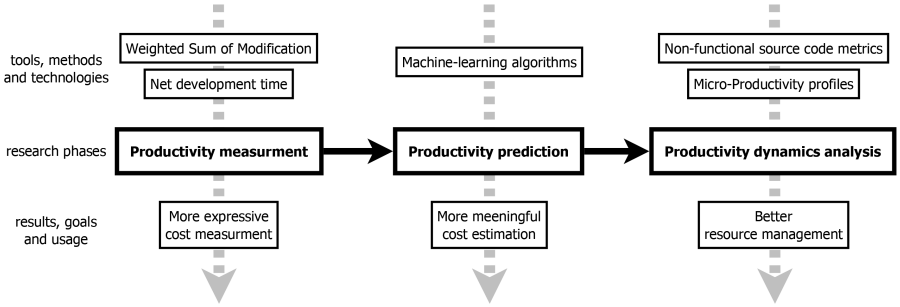


Figure 1: Phases of our productivity related researches

I am used the definition of productivity, where it is expressed as the ratio of the aggregate output to a single input, i.e., output per unit of input. More precisely, I used the ratio of the weighted count of modification (TMod) with the net development time (dt) to express productivity with the newly introduced Modification Effortmetric.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{TMod}}{\text{DT}} = \text{MEff} \tag{1}$$

T1.1: ***Productivity metrics that incorporate types of modifications possess more expressive power.*** I present two new metrics [7] for productivity measurement, namely Typed Modification (TMod) and Modification Effort (MEff). These highly customizable source code metrics are more expressive than the number of changed source code lines, which is commonly used to measure the productivity of developer teams or individual developers. We chose to use the modification effort because, during the implementation, developers consider methods and classes as logical units and not individual lines of source code.

MEff is a number that expresses the average amount of performed modification during a unit of time. The code example in listing 1 will be used to illustrate the measures for expressing programmer productivity. The modified code in listing 2 includes two changes over the previous version, occurring in three separate lines. The first change refers to a "return type change" in

line 2, while the second one is a "method implementation change" in line 4 and 6. For illustration purposes, let us assume that it takes 8 minutes for the programmer to implement both modifications together. Based on these values, the modification effort can be calculated by taking the ratio of the sum of the modification and the net development time: (1 return type ch. + 1 method imp. ch.)/8 min = 0.25. Notice that it is different from the naive method, which only counts the changed lines: 3 changened lines/8 min = 0.375. Furthermore, we could also use weights for different modification types to express the difference in effort to produce them.

Listing 1: Original version

```
class IntSet {
  protected double Find(double limit) {
    for (int i=0; i<Count(); i++) {
      double current=Items[i];
      if (current>limit) {
        return current;
      }
    }
  }
}
```

Listing 2: Modified version

```
class IntSet {
  protected int ✍ Find(double limit) {
    for (int i=0; i<Count(); i++) {
      int ✍ current = Items[i];
      if (current>limit)    {
        return i; ✍
      }
    }
  }
}
```

T1.2: ***The effectiveness of productivity prediction can be increased by taking account of different types of modifications.*** I was able to increase the efficiency of the previous modification cost prediction method based on product and process metrics using the previously mentioned novel approach to productivity measurement. I found that my productivity estimation model [7] can achieve a significant improvement in the overall efficiency of the prediction, from around 50% to 70% (F-measure).

We used decision trees (a machine-learning algorithm) to construct the prediction model. We also utilized a genetic algorithm to fine tune the base

model. The fitness value was calculated for each individual by evaluating the underlying prediction model with different weights.

The initial data was collected during the experiment from about 800 revisions, in an approximately 75 days long period. Both R&D and industrial projects were analyzed. The majority of their codebases were written in Java language using the Java EE 6 virtual machine and the Seam 2 framework.

T1.3: ***Measurement of wasted effort via developer interaction data helps managers to improve the development process.*** In a field study [10], I analyzed the aspect of productivity dynamics in a medium-sized J2EE project with 17 developers for seven months (fig. 2). Based on the experiments, project stakeholders identified several points to improve the development process.

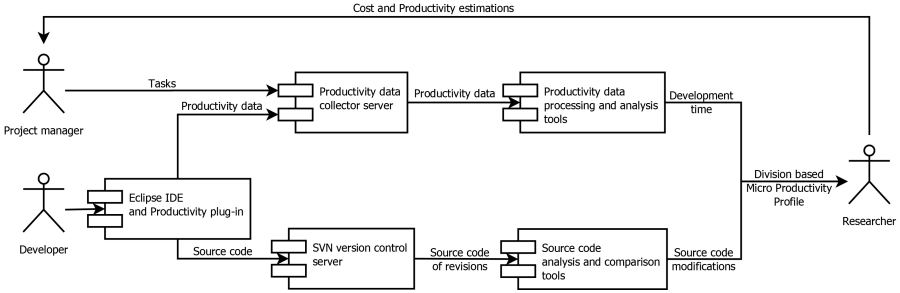


Figure 2: Measurement architecture

The central concept during this phase of the research is the Division based Micro-Productivity Profile (MPPD for short), which measures the frequency of changes in productivity at various granularity levels (fig. 3). A MPPD curve (fig. 3) shows the superfluous effort spent by developers during the implementation. In an ideal case, these would be zero, and the MPPD would be a flat line. In real life, these values are affected by incomplete specifications and requirements, which are changing over time. The steepness of the MPPD curve can be interpreted as the ratio at which the developers re-modify the same code again.

T1.4: ***The average of software quality (a factor of productivity) of the students and the professional developers' work does not show significant differences in classroom exercise.*** I conducted a case study [1] where several student's works were compared to works created by professional

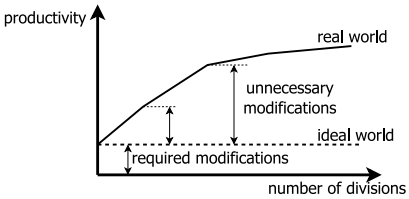


Figure 3: The underlying concepts of MPPD

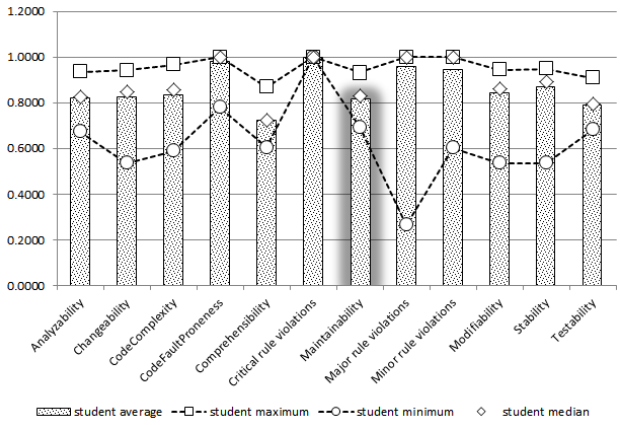
developers by using non-functional properties like software quality. The results suggest that there are not any significant differences between the average performance of the two groups. Although the quality of source code produced by experts had less fluctuation (fig. 4).

T2: Providing immersive methods for software and unit test visualization. This thesis point is related to the visualization of software system architectures and their connected entities.

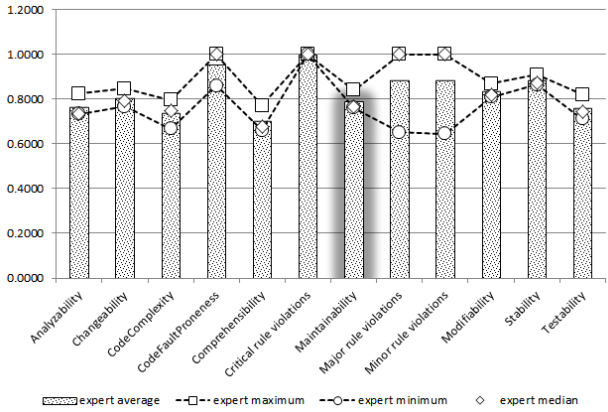
T2.1: ***Sandbox game-based techniques can enhance the visualization of software as a virtual city.*** My main contribution was to connect data visualization with the end-user graphics capabilities of games. My enhanced visualization method (and its supporting toolset) aided the developers and students to comprehend complex software systems by constructing a virtual city, which represents abstract, software development related concepts like source code metrics.

We used two main levels to represent data and entities of our visualization process. On the data level, each item has its own property set, for example, metrics. These information are displayed on the metaphor level, which all buildings in the metropolis belong to. The buildings and the world (city) itself has a couple of attributes that control its visual appearance. The properties are mapped to the attributes in order to visualize the data.

T2.2: ***The degree of realism for the city metaphor in software visualization can be measured automatically.*** I presented three low- and one high-level metric that expresses various features of a virtual city used to visualize software systems to capture the differences between a realistic and an unrealistic city. Both high- and low-level metrics were validated by a user survey [3]. The results show that it is possible to construct methods that can estimate the degree of realism of a generated city.



(a) Code quality metrics of students



(b) Code quality metrics of experts

Figure 4: High level code quality metrics

The three low-level metrics were: compactness, which expresses the density of the buildings in the city (fig. 6a); homogeneity, which represents the smoothness of the small scale scenery (fig. 6b); and connectivity, which describes the spatial coherence among buildings (fig. 6c).

I used the aggregated results of 51 complete and 20 partial surveys to

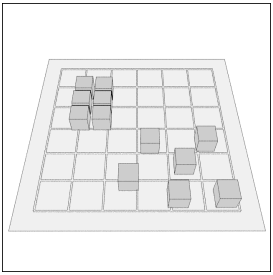
(a) Elevated grounds to group items

(b) Gardens with various flower-ratio

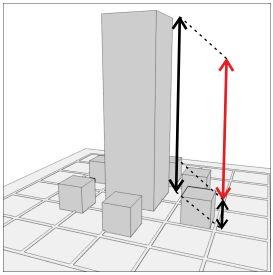
(c) Buildings surrounded with gardens

(d) Floors with various materials

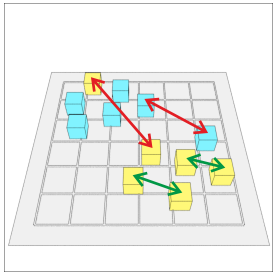
Figure 5: Items of the metaphor level



(a) Compactness



(b) Homogeneity



(c) Connectivity

Figure 6: Low-level metrics

construct the high-level metric. I asked the users to rank the cities according to their degree of realism and they had to decide which of the two given cities could be used as an example from a specific point of view. I utilized the Kendall tau correlation coefficient and community detection algorithm to

select a ranking, which reflects the opinions of most of the users. Then I solved a relaxed version of the inequality-system representing this particular opinion to calculate the weights to construct the high-level metric.

T2.3: ***Integration of integrated development environment and software visualization can aid developers to understand software systems.*** I present an approach to integrate our visualization tool, CodeMetropolis, into the Eclipse IDE. A set of plug-ins were implemented that were able to connect these two pieces of software (fig. 7). Hence, we became capable of integrating an elaborated visualization technique without disturbing the daily routine of developers.

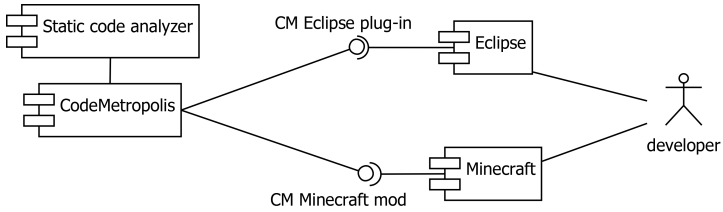


Figure 7: Overview of integration

T2.4: ***The city metaphor is able to visualize test-related metrics and test-code connection.*** I extended the metaphor to include properties of the tests related to the program code using a novel concept [8]. This allowed the combining of two previous approaches: a method to express test quality in terms of metrics, and visualization of code related metrics in the CodeMetropolis framework.

This way, code will become, for example, *houses* and tests will turn to *outposts* "defending the code." Physical attributes of the outposts such as height and material will indicate, for example, how thoroughly the associated code is tested (covered) or how specialized the tests are to this code or whether they test other objects as well.

To visualize the test-related metrics, we are using the aforementioned outpost objects. What we want to see is how well the code elements are tested along with the features. Outposts are placed inside the gardens of classes. Each outpost has a central watchtower and a surrounding fence, as shown in Figure 8. The various height attributes are used to represent the value of metrics. Also, one of its two building materials reflects the assigned feature. In addition, the outposts are equipped with explanatory signs.

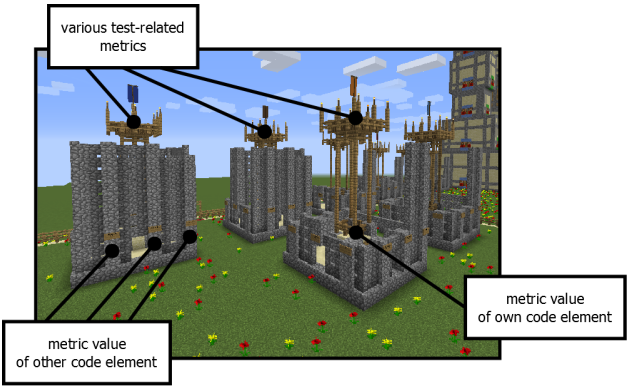


Figure 8: Parts of outpost of test-related metrics

T3: Spotting the structures in the package hierarchy that required attention using test coverage data. In this thesis point, I summarize my results considering test coverage analysis and its usage to improve the quality of (unit) tests and their subjects.

T3.1: ***Community detection is able to cluster test cases and code elements simultaneously.*** To automate various test and code analysis tasks, I employ a clustering algorithm that can group test and code items. This method allowed the simultaneous inspection of tests and their subjects and aided researchers in conducting further analyses.

We employ two clustering algorithms that can group together test and code items. The first one is based on code coverage and captures dynamic relations between the test suite and the system under test. This is then compared to the other, trivial clustering, that works from static (package hierarchy) information and captures the structural properties of the tests and program code. In order to determine the clustering of the tests and code based on the dynamic behavior of the test suite, we will apply community detection on the detailed code coverage information. The concept of clustering based on dynamic behavior used in this work can be illustrated by investigating different regions in the coverage matrix. Groups of tests and methods that form "dense regions" in the matrix may be grouped, indicating that there is a close correspondence between them from a dynamic point of view.

T3.2: ***Classification of structural discrepancies of tests and code elements helps developers to improve test and code quality by provid-***

ing contextual data to restore test-code traceability links. This work addressed the quality of unit test suites from a novel angle. My approach was to compare the physical organization of tests and tested code in the package hierarchy to what can be observed from the dynamic behavior of the tests.

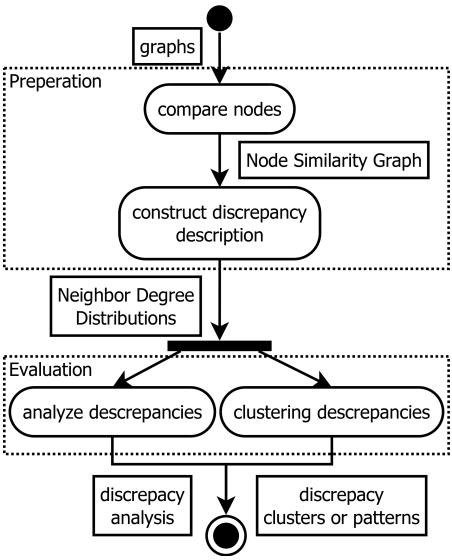
Guidelines through examples for refactoring the problematic tests were provided based on measurements of large open-source systems with notable test suites. Our subject systems were medium to large sized open-source Java programs which have their unit tests implemented using the JUnit test automation framework. We chose these systems because they had a reasonable number of test cases compared to the system size. The collected data was interpreted as contextual information for a semi-automatic method for recovering test-to-code traceability links. It is based on computing connections using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the given information.

The discrepancies found in the results of two clusterings can be seen as some sort of smell, which indicates potential problems in the structural organization of tests and code. In the first phase of our research, we used our experiences to define the three coarse-grained discrepancy patterns. Then, in the second phase, these patterns were refined, to help the user, who will determine the final traceability links based on the differences and contextual information. Our research utilized the Discrete Neighbor Degree Distribution and Node Similarity Graph, which are key concepts of the more general UniGDA methodology, to find and identify these patterns. These were discussed in the next sub-thesis point.

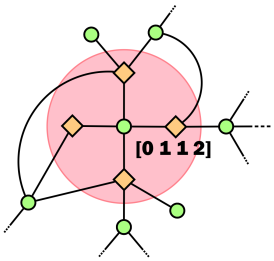
T3.3: ***Providing a methodology for unified graph's discrepancy analysis.*** I presented a methodology for a unified graph's discrepancy analysis, named UniGDA. It is based on the previously defined domain-specific discrepancy detection techniques, which were extended to arbitrary graphs by providing several domain-independent similarity functions and patterns.

UniGDA contains two main phases (fig. 9a). During the first one, we calculate the similarity between each pair of the inspected graphs and construct a generic description of the discrepancies among them. We use this information to classify or further analyze the discrepancies during the evaluation phase.

We construct a graph-based representation (Node Similarity Graph, or NSG for short), which contains all the required information to locate and process the discrepancies. To enable further analysis of the discrepancies, we require a unified way to characterize them, i.e. tools, which could be used to



(a) Overview of the Process



(b) Example graph for DNDD calculation

Figure 9: Methodology for Unified Graph's Discrepancy Analysis

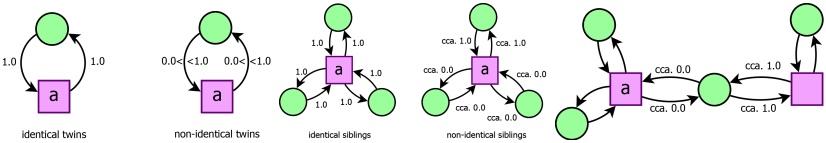
answer the questions: *How do similar entities relate to other entities?* To address this issue, we introduce a special feature vector and function to describe the inspected vertex and its neighbors in the NSG, namely the *Neighbor Degree Distribution (NDD) vector and curve*. As their name suggests, they encode the distribution of the inspected node's adjacent vertices according to their degree. When applied to the NSG, they provide an easy-to-use tool to describe the local similarity relations.

For example, $d = (0, 1, 1, 2, 0, 0, \dots)$ is the DNDD vector of the middle vertex in fig. 9b. The red circle represents the scope of the DNDD vector since it is only capable of encoding information in a 2-edge wide context. This DNDD vector means that the inspected vertex in the middle has the following neighbors.

$d_1 = 0$ There is not any adjacent vertex with only one connection.
$d_2 = 1$ It has one neighbor with two adjacent vertices, i.e. the left one.
$d_3 = 1$ There is one adjacent vertex with three connections, i.e. the right one.
$d_4 = 2$ It has two neighbors with the degree of 4, the top and bottom ones.

The usage of NDD vectors and curves made it possible to conduct an in-

depth analysis of node similarity. We could use these descriptors to define *similarity patterns*. These are sub-graphs of the NSG, which describe the relation of the inspected node in respect of its similarity to other entities. Figure 10 shows several examples for these patterns. It is exceedingly difficult to give a domain-independent meaning to more complex patterns, but, for instance, the fostering patterns could be an indicator of a poorly chosen similarity function. For example, if we try two pair source code items by using all of their attributes, including accessibility (such as private, public, and protected), there should be many methods that are slightly similar to others.



(a) Twins (b) Siblings (c) Fostering

Figure 10: Similarity Patterns

In the case of UniGDA, both the definition of meaningful similarity function and the interpretation of similarity patterns are challenging tasks. These aspects require in-depth knowledge of the field. My experience suggests that these can be done by manually analyzing several sample cases. It is a well-known fact, that the gathering of expert opinion is a time consuming, hence costly phase. But these tasks only have to be completed once, and subsequent analyses could use these data, thanks to my Methodology for Unified Graph's Discrepancy Analysis.

Bibliography

[1] Gergő Balogh. "Comparison of Software Quality in the Work of Children and Professional Developers Based on Their Classroom Exercises". In: *International Conference on Computational Science and Its Applications*. Springer, Cham. 2015, pp. 36–46.

[2] Gergő Balogh. "First Steps towards a Methodology for Unified Graph's Discrepancy Analysis". submittted for review to 13th International Conference of Graph Transformation, (part of STAF 2020).

[3] Gergő Balogh. “Validation of the city metaphor in software visualization”. In: *International Conference on Computational Science and Its Applications*. Springer, Cham. 2015, pp. 73–85.

[4] Gergo Balogh and Arpad Beszedes. “CodeMetrpolis—A minecraft based collaboration tool for developers”. In: *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE. 2013, pp. 1–4.

[5] Gergő Balogh and Arpad Beszedes. “CodeMetropolis-code visualisation in MineCraft”. In: *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE. 2013, pp. 136–141.

[6] Gergő Balogh, Attila Szabolics, and Arpád Beszédes. “CodeMetropolis: Eclipse over the city of source code”. In: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. IEEE. 2015, pp. 271–276.

[7] Gergő Balogh, Ádám Zoltán Végh, and Árpád Beszédes. “Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm”. In: *SSBSE Fast Abstract track* (2012), pp. 1–6.

[8] Gergo Balogh et al. “Using the City Metaphor for Visualizing Test-Related Metrics”. In: *1st International Workshop on Validating Software Tests*. 2016.

[9] Gergő Balogh et al. “Are My Unit Tests in the Right Package?” In: *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE. 2016, pp. 137–146.

[10] Gergő Balogh et al. “Identifying wasted effort in the field via developer interaction data”. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE. 2015, pp. 391–400.

[11] Tamás Gergely et al. “Analysis of Static and Dynamic Test-to-code Traceability Information”. In: *Acta Cybernetica* 23.3 (2018), pp. 903–919.

[12] Tamás Gergely et al. “Differences between a static and a dynamic test-to-code traceability recovery method”. In: *Software Quality Journal* (2018), pp. 1–26.