2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Using the City Metaphor for Visualizing Test-Related Metrics

Gergő Balogh
University of Szeged, Hungary
Software Engineering Department
geryxyz@inf.u-szeged.hu

Tamás Gergely
University of Szeged, Hungary
Software Engineering Department
gertom@inf.u-szeged.hu

Árpád Beszédes
University of Szeged, Hungary
Software Engineering Department
beszedes@inf.u-szeged.hu

Tibor Gyimóthy
University of Szeged, Hungary
Software Engineering Department
gyimothy@inf.u-szeged.hu

***Abstract*—Software visualization techniques and tools play an important role in system comprehension efforts of software developers in the era of increasing code size and complexity. They enable the developer to have a global perception on various software attributes with the aid of different visualization metaphors and tools. One such tool is CodeMetropolis which is built on top of the game engine Minecraft and which uses the city metaphor to show the structure of the source code as a virtual city. In it, different physical properties of the city and the buildings are related to various code metrics. Up to now, it was limited to represent only code related artifacts. In this work, we extend the metaphor to include properties of the *tests* related to the program code using a novel concept. The test suite and the test cases are also associated with a set of metrics that characterize their quality (such as coverage and specialization), but also reveal new properties of the system itself. In a new version of CodeMetropolis, gardens representing code elements will give rise to *outposts* that characterize properties of the tests and show how they contribute to the quality of the code.**

I. Introduction

Visualization of software artifacts has been the focus of researchers for a long time. However, its use in the daily work of software developers is still limited. Many research prototypes deal with a limited set of artifacts, for example only structural code information, programmer activity, or perhaps some process data is visualized. Although there have been countless, very creative approaches proposed, they are not always easily adoptable by software developers due to the sub-optimal metaphor employed by the approach, or cumbersome application of the visualization tool within the developer's well-established workflow.

The present work is a step towards reducing this gap because we introduce an approach to jointly visualize attributes of two related software artifacts side by side, namely code and tests, thus enabling their common investigation. There are numerous approaches in which various properties of the source code are visualized, however visualizing attributes of the associated tests (test cases or test suites) is rarely a concern of researchers.

In this work we employ the city metaphor [1]. It uses the concept of a landscape of buildings and related physical objects, where their physical attributes correspond to the software attributes. This enables the user navigating through software artifacts in a virtual 3D space, which is very intuitive from our daily lives. Our approach, CodeMetropolis [2], [3] differs from related methods in that we employ an existing 3D visualization engine borrowed from the Minecraft game [4].

Previously, only code related metrics were available in CodeMetropolis as the basic mapping from code artifacts to the virtual objects' properties in the Minecraft world. The present work extends the metaphor by using additional metrics computed from test related artifacts of the same system. Our approach to combine code and test metrics in CodeMetropolis is to build separate objects corresponding to the code and the associated tests on a physically close proximity. In addition, suitable mapping is used between the metrics and the physical properties such as building dimensions and build materials. This way, code will become *houses* and test will turn to *outposts* "defending the code." Physical attributes of the outposts such as height, density and material will indicate, for example, how thoroughly the associated code is tested (covered) or how specialized are the tests to this code or do they test other objects as well.

II. Motivation and related Work

Information can be represented in many different ways, but most of us like to see some visual representations where various colors, shapes, and sizes are assigned to different data attributes. This is not different in the field of software engineering, where various techniques and tools were designed to represent the software, its parts, and their different attributes in a visual form. The goal is usually to help the understanding and the analysis of the large amount of available data.

Data visualization techniques are often used in software engineering, but the relatively new concept of *gamification* is still far from this field [5]. In general, gamification employs gaming elements in non-game contexts in order to ease the learning curve of new technologies, improve user engagement and organizational productivity. In software engineering, there are some examples of using this paradigm, mostly in the co-texts of learning and enhancing motivation [6]. Our approach for combined test and code visualization could be a first step to utilize gamification for improving motivation and productivity, because testing and code inspection are often tedious tasks.

CodeCity [7] and EvoSpace [8] are two visualization solutions that are very close to our approach. They use the analogy of buildings in a city. CodeCity simplifies the buildings to boxes, and assigns source code properties to width, height,

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.48

and color attributes of these buildings. The buildings represent classes, and districts are formed from buildings whose classes are in the same namespaces. EvoSpace extends this mapping, and allows the exploration of lower level structures of the classes by "opening" the houses.

Visualization of test related phenomena is usually limited to code coverage or execution result metrics and simple diagrams. Sosnówka proposed the Test City Metaphor [9] to visualize test entities and properties in order to support regression test selection for low level test cases. This metaphor does not concern the code itself, however. The authors of the present paper used the city metaphor to visualize source code metrics in Minecraft [2], which is now extended to test-related metrics, while maintaining the visual representation of code-related ones.

III. Background

A. Measuring test-related metrics

Similarly to code metrics, test-related metrics can be defined for the different code and test artifacts. A popular test-related metric is *code coverage*, which expresses the percentage of how well the code elements are covered by the test cases. Code coverage can be computed at different levels: a single global value can express to what extent all the test cases are able to check the whole code base; a value can be assigned to method-test case pairs to show detailed coverage; or it can be assigned to *functional units* formed from pairs of code and test groups [10]. In a functional unit, a code group implements some functionality and the associated test group is intended to verify it. Analyzing the system and the tests with this kind of division can be an aid in test selection, prioritization, and test suite reduction activities [10], [11].

We defined several concrete metrics for the above mentioned functional units. For example, the *partition* metric characterizes how well a set of test cases can differentiate between the program elements based on their coverage information. This is an important aspect for fault localization applications, where the differences in execution profiles of passed and failed test case executions are observed to find code elements that caused the tests to fail. The *specialization* metric shows how specialized a test group is to a code group in terms of the ratio of other test groups. A small value shows that other test groups intensively test the code group in question, while a high value reflects better specialization. A related metric is the *uniqueness* metric, which measures the portion of the elements that are covered only (uniquely) by a particular test group.

These metrics are applicable to cross-functional code and test groups, not only to functional units. For example, we can compute how the test group of functional unit *A* covers the code group of functional unit *B*. These additional measurements can reveal properties of the test suite and its parts, and hence they may contribute to e. g. the changeability or maintainability of the test suite.

We used the *SoDA library and toolset* [12] to compute different test-related metrics. SoDA uses detailed coverage information and other metadata (e. g. functionalities tested or implemented by a group of items) to compute the above mentioned metric values as well as others such as the tests to code element ratio.

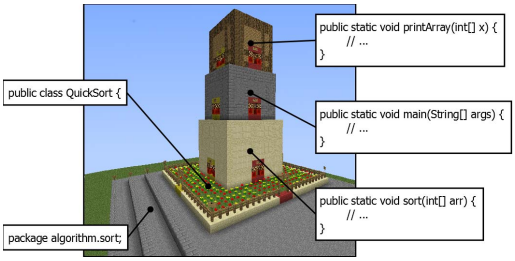

Fig. 1: Source code in the city metaphor

B. The city metaphor

In software visualization, there have been various mappings proposed between code elements and visualization space objects. As mentioned above, these mappings are usually based on metaphors such as forests, landscapes or solar systems. The metaphor determines what objects can be used in the visualization space and what attributes of these objects can potentially be used to capture the properties of the represented entities. A popular metaphor in software visualization is the city metaphor [1], where source code elements are assigned to buildings, and the whole software is presented as a city.

On a high level, the city metaphor usually assigns classes to buildings and attributes to width, length, height, or color of the buildings. Sometimes additional elements like the roof size, shape, or color are used to express other metrics. Buildings may also represent methods, which are grouped into districts according to their relationsip in the code; districts are usually formed from namespaces or packages. The hierarchical structure of packages are usually represented by flat platforms that are elevated from their context. Figure 1 shows a possible implementation of the city metaphor visualization using the CodeMetropolis tool [13], [2] (here, static source code metrics are presented only).

IV. Test Visualization in CodeMetropolis

A. Overview

CodeMetropolis assigns gardens to classes, cellars to attributes, rooms to methods, buildings to method sets, and uses elevated platforms to denote namespaces (or packages) and express inclusion. Different metrics can be assigned to properties of various components of the virtual city. For example, the physical dimensions of cellars and rooms, the amount of the flowers, trees or mushrooms in a garden can represent various metrics like complexity, size, coupling, code style issues, etc. The assignment between the metrics and the visualization attributes is easily configurable.

The main goal of this work was to extend this metaphor and include the visualization of functional and cross-functional units, and test-related metrics, but also preserve visibility of existing static code attributes. Former methods presented either

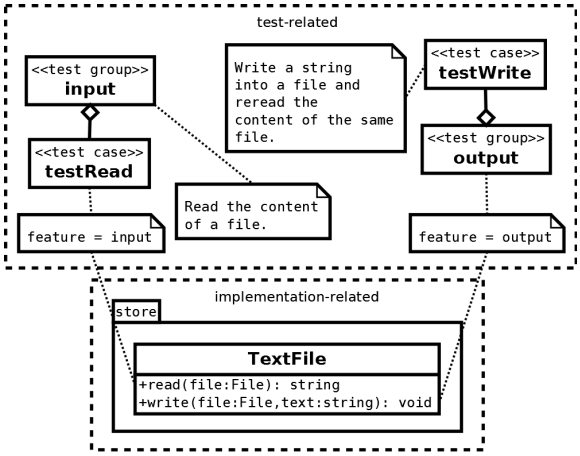

Fig. 2: An example of functional units

code or test related objects individually, but not both in a common space.

Data preparation: The first step in our method is source code analysis and code metrics computation. Next, functional units are formed by assigning code and test case groups to the different feature sets of the program. This process can be performed in various manual, semi-automatic or automatic ways, but this is not a topic of the present paper. Finally, tests are executed, which generates the detailed code coverage from which test related metrics are calculated. More details of the process can be found in other work [10].

Visualization: The first step in visualization is to map code and test entities and their properties to architectural or landscape objects, to their attributes, and other visually observable phenomena. Then, these objects are constructed from the building blocks of the Minecraft world and placed in it according to their relations. Finally, the game loads the created world and the developers can get around the city and examine different objects.

B. Program elements to be visualized

The existing visualization concepts in CodeMetropolis remained the same, thus all source code elements (namespaces, classes, methods, attributes) and their properties (source code metrics) are visualized as before. The additions are *functional* and *cross-functional units* and their metrics. Functional units are organized around the functionalities (features) of the software. For each feature there are test cases created to test the given functionality, we call these the *test groups*. Similarly, the features were implemented in certain classes and methods, which constitute the *code group*. A *functional unit* consist of the code group and the test group of the same feature, while a *cross-functional unit* consists of a code group and a test group of two different funcionalities [10].

As an example, consider a class called `TextFile` with only two methods, `read()` and `write()` implementing *input* and *output* features, which are tested by the test cases **testRead** and **testWrite**, respectively (Figure 2). This enables us to define two functional units: *input-input* with **testRead** and `read()`; *output-output* with **testWrite** and `write()`; as well as two cross-functional units: *input-output* with **testRead** and `write()`; *output-input* with **testWrite** and `read()`.

We could not directly visualize functional units as simple objects in the virtual space, as this combination of code and test groups does not fit in the existing hierarchical approach based on the source code. So, we decided to represent a functional unit as some visible properties of the corresponding objects. Similarly, test cases do not appear in the visualization space as individual objects, rather our metrics were computed for the *code item–test case* relations. Since the granularity of the metrics is not individual pairs of these items but functional code and test groups, the base of visualization will be *code group–test group* relations. More concretely, test related metrics computed for a given functional or cross-functional unit will appear as objects, and their visual properties will reflect the corresponding metrics. We call these objects the *outposts*.

However, if we placed outposts directly in the visualization space, we would loose the connection between them and the source code. Therefore, we place an outpost for each (test metric, code group, class) triple. This decision implied an additional property to be visualized: the *completeness* of a feature regarding a class. This metric expresses the ratio of the concerned code elements (methods in the same class that belong to the assigned code group) compared to all code elements that are assigned to the feature implemented by the given code group. For example, if the class `TextFile` has a `read()` method assigned to the *input* feature, which has two other assigned methods in addition from other classes, then `TextFile` provides $1/3$ feature completeness for the *input* feature.

C. Side by side visualization of code and tests

Two objects whose source code elements are close to each other in the code structure (and hence appear also close in the virtual city) may implement different features. Similarly, the same feature may be assigned with distant objects, so mapping functionalities to object placement would raise several problems. Therefore, instead of representing features as objects they will be mapped to object properties. Fortunately, in Minecraft we can use many kinds of building blocks, so we assigned different blocks to different features. Then, the outlook (color and texture) of the objects represents the assigned feature.

To visualize the concept of (cross-) functional units and their test related metrics we are using the mentioned outpost objects. What we want to see is how well the code elements are tested along the features, so we had to find a way to visualize the metric values of functional and cross-functional units. Outposts are placed inside the gardens of classes, and each outpost is assigned to a (test metric, unit, class) triplet. Each outpost has a central watch tower and a surrounding fence as shown in Figure 3. The height attribute is used to represent the metric value. Also one of its two building materials reflects

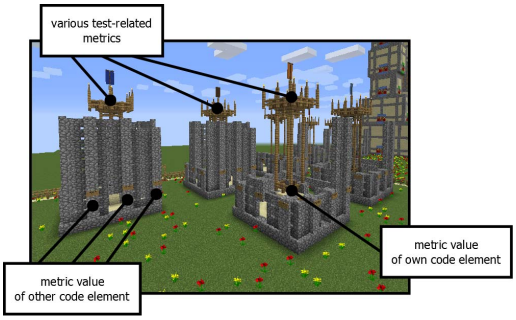

Fig. 3: Parts of outpost of test-related metrics

the assigned feature. In addition, the outposts are equipped with explanatory signs and a colored flag on the top.

The basic concept behind using outposts is that tests are "guarding" the code. Tests of a certain functional unit are created to check the quality of the code of the same unit, but they are not intended to test code from cross-functional units. Based on this consideration, each outpost of the class is assigned to a *metric–code group* pair and represents a single metric of multiple (cross-) functional units. The central tower of the outpost is assigned to the test group of the same feature the code group of the outpost was assigned to, while segments of the surrounding fence of the outpost represent the other test groups. Thus, the central tower represents the metric value of the functional unit. The completeness of the central tower shows the actual feature completeness regarding the given class and functionality. The tower also has a scaffolding up to the top which represents the actual metric value. The name of the functional unit is shown on a wall sign, but also encoded into the building material of the tower walls and outpost ground. This provides a strong visual connection between the methods and the outposts of their test related data.

The surrounding fence of the tower is divided into segments and each segment is assigned to a cross-functional unit (cross-functional test group of the given code group). The height of each fence segment represents the metric value of the same metric (which was assigned to the outpost) for the cross-functional unit assigned to the segment. For example, if the outpost stands for coverage and is assigned to the *input* code group, the fence will represent the *output* test group, i. e. the *output-input* cross-functional unit. Note, that in the example the whole fence represents the same unit as this is the only cross-functional unit.

This construction of the outpost lets us visualize some common shortcomings of the tests. For example, a high fence around a low tower in an outpost (like the leftmost outpost in Figure 3) assigned to the coverage metric (and some feature) shows that tests intended to check the implementation of a feature are not performing well, while other tests (not intentionally created to do so) will do the job instead; which violates the modularity of the system.

V. Conclusions

Understanding the structure of large test suites and the relation of its constituent test cases to the code of a system is hard, and there are not much tools to aid this activity. This work combines two previous approaches: a method to express test quality in terms of metrics, and visualization of code related metrics in the CodeMetropolis framework. The city metaphor employed by CodeMetropolis seems to be useful for test metrics as well, and we believe that the side by side presentation of code and tests will enable for the developer to obtain a more global picture of her software.

Currently, the approach has been tried on systems that we developed, about which we have in depth knowledge. In the near future we plan to perform additional experiments, possibly involving human evaluation, on other software. Our long term goal is to enhance the metaphor to include additional information sources (such as defects or process data) because we believe that a successful visualization needs to feed from multiple sources.

References

[1] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 921–922. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370188

[2] G. Balogh and Á. Beszédes, "CodeMetropolis - code visualisation in Minecraft," in *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13), Tool Track*, Sep. 2013, pp. 127–132.

[3] "Homepage of codemetropolis project." [Online]. Available: http://www.sed.inf.u-szeged.hu/codemetropolis

[4] "Minecraft Official Website." [Online]. Available: http://minecraft.net/

[5] D. J. Dubois and G. Tamburrelli, "Understanding gamification mechanisms for software development," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 659–662.

[6] S. Deterding, "Gamification: Designing for motivation," *ACM Interactions*, vol. 19, no. 4, pp. 14–17, 2012.

[7] R. Wettel and M. Lanza, "Codecity," in *Proceedings of 1st International Workshop on Academic Software Development Tools and Techniques 2008*, ser. WAS-DeTT 2008, 2008, pp. 1–13.

[8] D. Lalanne and J. Kohlas, *Human machine interaction: research results of the MMI program*. Springer Science & Business Media, 2009, vol. 5440.

[9] A. Sosnówka, "Test city metaphor for low level tests restructuration in test database," in *Evaluation of Novel Approaches to Software Engineering*. Springer, 2013, pp. 141–150.

[10] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, "Beyond code coverage – an approach for test suite assessment and improvement," in *Proceedings of the Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART 2015)*. IEEE Computer Society, Apr. 2015, pp. 1–7.

[11] F. Horváth, B. Vancsics, L. Vidács, Á. Beszédes, D. Tengeri, T. Gergely, and T. Gyimóthy, "Test suite evaluation using code coverage based metrics," in *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15)*, Oct. 2015, pp. 46–60.

[12] D. Tengeri, Á. Beszédes, D. Havas, and T. Gyimóthy, "Toolset and program repository for code coverage-based test suite analysis and manipulation," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, Sep. 2014, pp. 47–52.

[13] G. Balogh, A. Szabolics, and A. 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.