
Visualizing Software Systems and Team Activity

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Design

presented by
Francesco Rigotti

under the supervision of
Prof. Michele Lanza
co-supervised by
Alberto Bacchelli, Lile Hattori

September 2011

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Francesco Rigotti
Lugano, 7 September 2011

Abstract

Software evolution consists of the modifications made to a software system, throughout its life-cycle, to cope with changes in its requirements or dependencies. Modifying a system, without having a sufficient understanding of it, has a high probability of introducing defects. Program comprehension is therefore required for an evolution that avoids system decay.

Software visualization techniques can be used to ease program comprehension through visual metaphors that leverage the ability of the human eye to identify colors, shapes, patterns, and differences. Many powerful software visualization tools exist, but the majority of them consists of stand-alone systems that are not integrated with the development tools already in use.

We have developed *Manhattan*, an *Eclipse* plugin that visualizes projects in the workspace as cities. While working on a project, a developer can see a representation of it, which is updated in real-time according to the changes he performs. Moreover the developer is visually informed about the changes performed by his colleagues in order to improve his awareness of the activity of the team as a whole, thus providing some sort of implicit communication that eases collaboration among developers.

Acknowledgements

First of all I want to thank Michele Lanza for being a great teacher and a great supervisor. I was taught not just mere notions, but good style and passion. Thanks for always helping me to do my best and making me proud of my work.

I want to thank Alberto Bacchelli and Lile Hattori for all the help they have given me regardless of their duties and deadlines. This thesis would not exist without them.

I have to thank my parents for supporting me in every way since I was born and for always giving their all, and even more than they should, for our family.

I thank my brother for all the advice he has given me and for all the fun we had.

I want to thank my aunt for teaching me that if you always do your best regardless of the situation, then you won't have any regrets, even if you fail.

My grandparents were the best. Grandpa was just too fun to be with and grandma was a great woman and a great cook. Please look over me from up there.

I thank my nephews for brightening my days with their smiles and their astonishing growth. Your uncle is proud of having taken care of you in these years. I also thank my sisters for being there and for giving me four incredible nephews always willing to play pirates.

Thanks to Gio, Grazia, Nico, and Po for being more than friends since when I came to this country. I won't forget how Ando, Bock, Gillo, Indu, and Tet made these years at USI so cool and entertaining.

Thanks to Sangio for proving himself as one of my best friends in life, despite we have seen each other for only two years; I hope that one day I will be able to repay you.

Thanks to Cora for introducing me to great people and great things.
Thanks to Ivo and Asta, for all the fun: everything was **gerei!**

Teo, with your actions and growth you showed that anyone can walk that path, but the choice is his and, once he is in, he has to walk with his own legs. Nothing is more important than this.

Thanks to the other people of Luganega and Fonta for walking down the same path with me and for turning every lunch and dinner into a feast.

Contents

Contents	vii
List of Figures	ix
1 Introduction	1
1.1 Software Visualization	1
1.2 Collaboration Support	2
1.3 Manhattan	3
1.4 Structure of the Document	4
2 State of the Art	5
2.1 Software Visualization	5
2.2 Collaboration Support	6
3 Manhattan	9
3.1 Contributing to Commons Math - a Use-Case for Manhattan	10
3.2 Visualizing Software Systems	12
3.2.1 The Code-Model	12
3.2.2 The City Metahpor	14
3.2.3 The View-Model	16
3.2.4 The Layout Algorithm	17
3.2.5 Reacting to Changes	19
3.2.6 Caching	21
3.2.7 Rendering the Visualization	22
3.2.8 Interacting with the Visualization	23
3.3 Visualizing Team Activity	24
3.3.1 Syde	24
3.3.2 Awareness in the City	26
3.3.3 Visualizing Change Notifications	27
3.3.4 Visualizing Conflict Alerts	29
4 Applications and Evaluation	33
4.1 Exploring a few Case Studies	33
4.1.1 ActiveMQ	34
4.1.2 Ant	35
4.1.3 Cobertura	36

4.1.4	jEdit	37
4.1.5	Vuze	38
4.1.6	NetBeans	39
4.1.7	Exploration Wrap Up	40
4.2	Evaluating our Approach with an Exploratory Study	40
4.2.1	Study Description	40
4.2.2	Results	42
5	Conclusions	43
5.1	Future Work	44
5.1.1	Corner-Stitch Layout	44
5.1.2	Improve the Visualization of Changes	44
5.1.3	Improve Performance	45
5.1.4	Support other Programming Languages	45
A	A	47
	Bibliography	73

Figures

3.1	Manhattan running inside of Eclipse	10
3.2	The tooltip for class <i>LocalizedFormats</i>	11
3.3	The activity of contributors to Commons Math	11
3.4	The tooltip for the emerging conflict on class <i>GeneticAlgorithm</i>	12
3.5	The code model used in Manhattan	13
3.6	The city of Apache BCEL	14
3.7	The mappings defined by the city metaphor	15
3.8	The city of ArgoUML	15
3.9	The view model	16
3.10	The city for a simple project (a) and the corresponding glyph tree (b)	17
3.11	The execution steps for the Rectangular-Packing algorithm, provided by Richard Wettel	18
3.12	The glyphs affected by an insertion (a), a resizing (b) and a removal (c)	19
3.13	The growth tolerance thresholds used to decrease the number of re-layouts	20
3.14	The design we conceived to render the visualization	22
3.15	The selection step (a) and the resulting focused city (b)	23
3.16	An architectural view of <i>Syde</i>	25
3.17	The information flow defined by our unified view	26
3.18	An aerial view of Commons Math showing notifications of developers' activity	28
3.19	A bird's view of the city of ArgoUML, enriched with change notifications	28
3.20	The tooltip description for a class modified by various developers	29
3.21	The city of CommonsMath with conflict alerts made visible using conflict beacons	30
3.22	The tooltip description for a conflict sphere	30
3.23	Conflict inspection in Manhattan	31
4.1	The city of Apache ActiveMQ	34
4.2	The city of Apache Ant	35
4.3	The city of Cobertura	36
4.4	The city of jEdit	37
4.5	The city of Vuze	38
4.6	The metropolis of NetBeans	39
A.1	Participant1, pages 1-4	48
A.2	Participant1, pages 5-8	49
A.3	Participant1, pages 9-12	50
A.4	Participant1, pages 13-16	51

A.5 Participant1, pages 17-20	52
A.6 Participant1, pages 21-24	53
A.7 Participant1, pages 25-28	54
A.8 Participant1, pages 29-32	55
A.9 Participant1, pages 33-36	56
A.10 Participant1, pages 37-40	57
A.11 Participant1, pages 41-44	58
A.12 Participant1, pages 45	59
A.13 Participant2, pages 1-4	60
A.14 Participant2, pages 5-8	61
A.15 Participant2, pages 9-12	62
A.16 Participant2, pages 13-16	63
A.17 Participant2, pages 17-20	64
A.18 Participant2, pages 21-24	65
A.19 Participant2, pages 25-28	66
A.20 Participant2, pages 29-32	67
A.21 Participant2, pages 33-36	68
A.22 Participant2, pages 37-40	69
A.23 Participant2, pages 41-44	70
A.24 Participant2, pages 45	71

Chapter 1

Introduction

Software engineering is concerned with writing programs that satisfy a given set of requirements. Throughout the entire life of a software system, specifications, used technologies, and the operational environment continue to change. If no appropriate measures to cope with these changes are taken, the system keeps aging until it becomes useless [Par94]. Software evolution is the set of modifications performed on a system, throughout its life-cycle, to prevent it from aging.

Modifying a system without having a sufficient understanding of it has a high probability of introducing defects or even disrupting the system's design, thus accelerating the aging process. Therefore the first requirement for a system to evolve properly is program comprehension: One needs to know the functionality implemented in each module, how modules collaborate with each other, and which modules are affected by a particular change in a given module. Nevertheless, understanding a software system is not trivial, because software is extremely complex, is intangible, and is in constant evolution.

The growth of software systems in size and complexity consequently leads to larger development teams, thus requiring a high level of collaboration and coordination among developers, which is not simple to achieve. In this context, two obstacles hinder the successful evolution of a software system: The difficulty to understand the system and the collaboration issues introduced by large teams.

We devised an approach that helps developers to overcome both obstacles by combining the efforts of researchers in the areas of *software visualization* and *collaboration support*. In the next sections we first introduce these research areas, then we describe our approach.

1.1 Software Visualization

In the past decades, researchers in this field have eased reasoning about software systems by producing interactive visual representations that make software “more tangible”. Software visualization techniques can be used to ease the comprehension of the design, behavior, and evolution of software systems, as they allow one to perceive software through vision, the sense from which humans acquire most information [War04].

Visual metaphors can be used to describe a system's properties with shapes, colors, and patterns, all of which can be identified by the eye quickly and in great quantity (i.e., in a single gaze, humans can identify many different colors and shapes at once), thus making visualization scalable.

To best support developers in driving the evolution of a system, a software visualization tool needs to be tightly integrated in the development process and used throughout the entire life of the system. This is because software evolves from its inception to its end. There are many software visualization tools that provide a considerable support to program comprehension, design assessment, evolution analysis, and reverse engineering tasks. However, most of these tools are standalone products that require a new piece of software to be installed on developers' computers. Standalone tools force developers to move away from the development environment, thus discouraging their use.

Although the ability of these tools to aid program comprehension is not affected by the described integration issues, their usefulness in the context of software evolution decreases, as they are less likely to be used. Since programmers build software systems using an *IDE* of some sort, the easiest way to integrate a tool in the development process is having the tool reside in the *IDE* itself.

With these considerations in mind, we have created *Manhattan*, a software visualization tool, in the form of an *Eclipse* plugin, that produces interactive visual representations of projects in the workspace.

1.2 Collaboration Support

In multi-developer projects, successful development requires collaboration and coordination among many developers. To properly coordinate their activities, developers need a certain level of awareness of the team's activity: Each developer needs to know who is currently working on what, who is expert on a given artifact, and who has recently worked on a given piece of code [DISK07, DB92].

Awareness is mainly achieved via communication and inspection of code changes on software configuration management systems (SCMs). In co-located development teams, face-to-face meetings are the preferred communication means [LVD06]. However, development teams spread across diverse countries and timezones has become the standard for open-source projects and large companies. This setting is referred to as *global software development* (GSD). Because of distance and differences in organizational culture, face-to-face meetings become impossible in *GSD* and are usually replaced with text-based communication, which revealed to be an inadequate replacement [DISK07].

Researchers tried to address this communication issue by notifying developers, in real-time, about changes performed by others and about emerging conflicts [BCSR07, dSCdW⁺06, HP08, SRvdH08, SGPP04, Sch01]. However, these contributions either treat source code files as plain text, thus losing important structural information, or do not store the collected change information for future use, thus losing the benefits of evolution analyses (e.g., code ownership analyses).

In the context of her Ph.D. work, Hattori created *Syde*, a set of *Eclipse* plugins, providing developers with real-time change notifications and alerts of emerging and emerged conflicts [HL09b]. Hattori's approach does not suffer from the drawbacks illustrated above.

By including *Syde* information in our visualization, we propose a unified view focused on the structure of software systems and on the activity of their development teams.

1.3 Manhattan

Projects in the workspace are visualized using the 3D city metaphor conceived by Wetzel and Lanza [WL07, Wet10]. This metaphor is effective, due to the many similarities between software engineering and architecture, and scales well for large projects.

To visualize a project, we first extract a model from its source-code and we create the visualization based on the extracted information. Our model contains information about components and their relationships, with a few metrics computed for the classes in the system. Since we model only the basic concepts of object oriented languages, we can virtually visualize all systems written with this language paradigm. Nevertheless, at the time of this writing, only a model extractor for Java systems is available.

Every *Eclipse* plugin has access to projects' source-code, to a multitude of core APIs, and to the APIs offered by other plugins. By exploiting these APIs, *Manhattan* offers three main advantages over many software visualization tools:

Autonomous Simplified Model Extraction

Without the need of additional tools, system models can be extracted by exploiting APIs offered by language-integration plugins (e.g. JDT for Java). Using these APIs also allows one to implement high-level parsers, which do not have to deal with language-specific syntaxes;

Real-time update

Both the model and the visualization are updated in real-time according to changes performed by developers. This updates are also triggered when developers check out from a repository the changes performed by their colleagues;

Interaction with code

If a building attracts the interest of a developer, he can open the corresponding artifact in the *Eclipse* editor from within the visualization. When a programmer is assigned the task to maintain a system he has no experience with, he can visualize it with our tool and explore the system by exploring the visualized city. The ability to inspect code from within the visualization is most useful in such a situation, because it makes the exploration process much smoother and integrated with the code.

By exploiting the APIs offered by *Syde*, we inform developers about the activity of their colleagues with clear notifications, based on colors, shapes, and lights, that do not clutter the visualization. Every developer is notified about the changes performed by his colleagues and about the emerging conflicts in which he is involved. Moreover, conflicts can be examined in an instance of *Eclipse*'s compare editor opened from within the visualization.

1.4 Structure of the Document

Chapter 2 contains a description of the related work done in software visualization and collaboration support. In Chapter 3 we describe our approach. We used *Manhattan* to visualize some real-world systems and, in Chapter 4, we show the resulting cities and briefly describe interesting design aspects highlighted by the visualization. Later in the chapter we describe an exploratory study we conducted to validate our approach and present the results. In Chapter 5 we draw the conclusions and present future improvements.

Chapter 2

State of the Art

Since our work is focused on two different research areas, we first describe the related work belonging to the software visualization field. Then, in Section 2.2, we describe related work done in the area of collaboration support.

2.1 Software Visualization

According to Price et al., software visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to ease both the human understanding and the effective use of computer software [PBS93].

Diehl divides software visualizations in three main categories[Die07]:

- Structure visualizations - focused on the components of a system and their static relationships;
- Behavior visualizations - focused on information obtained by running a system;
- Evolution visualizations - focused on the changes a system undergoes throughout its lifetime.

Our work belongs to the category of structure visualizations. Out of the many contributions to this category, we describe, in the next paragraphs, the work that most affected the direction of our thesis.

In 1986, Müller et al. created *Rigi*, a visualization tool offering a structure visualization of systems in terms of components and relationships [Mul86, MK88].

Six years later, Eick et al. presented *SeeSoft*, a tool offering visualizations of systems from different perspectives, among which structural and evolutionary ones [ESJ92].

In 1996, Ball and Eick reported the successful application of *SeeSoft*, at Bell Laboratories, to ease the maintenance of "*a system system containing millions of lines of code, developed over the last two decades by thousands of software engineers*" [BE96].

In 1999, Lanza introduced the concept of polymetric views [Lan99] and implemented them in *CodeCrawler*, a tool to support reverse engineering of software systems. Polymetric views are simple interactive graphs, enriched with various software metrics. Lanza validated the effectiveness of *CodeCrawler*, by attempting to reverse engineer an industrial system in a few days. The quality of the obtained results is an additional proof of the supportive capabilities of software visualization.

In the 90s, 3D software visualizations began to appear. The early 3D visualizations didn't exploit the possibilities offered by having one more dimension [SB99], but later, novel metaphors, specially conceived for 3D, were presented [TC09].

In 2000, Knight et al., proposed and implemented an innovative 3D metaphor called *Software-World* [KM00]. In this metaphor the visualized system is the world, files are cities, classes are districts and methods are buildings.

In 2003, Panas et al. described a 3D city metaphor able to show both architectural and behavioral properties of a system, implemented in a visualization framework called *Vizz3D* [PBG03]. Four years later [PEQ⁺07], Panas et al., proposed a unified single-view visualization consisting of a 3D city enriched with information obtained from execution profiles and software repositories.

In 2007, Wetzel and Lanza presented another city metaphor, which they implemented in *CodeCity*, a visualization tool supporting program comprehension, evolution analysis and design assessment [WL07]. To support evolution analysis, a city for each observed version of a system is produced. Since a consistent layout is maintained across all cities (locality principle), users can observe the evolution of a system as they would observe the evolution of a city.

Our work is inspired by *CodeCity*, a software system that combines many ideas into a multi-purpose scalable visualization. Nevertheless, like any tool, *CodeCity* is not perfect. We believe that being a standalone tool is its main drawback, as it hinders its integration in the development process. A viable solution to this integration problem is re-implementing *CodeCity* as an *Eclipse* plugin. Wetzel has been working on *CodeCity* for multiple years, therefore in this master thesis we focus on implementing the essential features of *CodeCity* in an *Eclipse* plugin.

2.2 Collaboration Support

Awareness, defined as "an understanding of the activities of others, which provides a context for one's activities" [DB92], is fundamental for successful collaboration among developers. In the context of software engineering, awareness can be expressed as knowing who has recently worked on a module and who is expert on which modules [DISK07].

The first benefit of awareness, is that it decreases the likeliness of merge conflicts on SCMs: If a developer performing some changes notices that others are modifying artifacts affected by his modifications, he can contact these programmers to find a solution that does not generate conflicts.

The work done by Grinter [Gri96], in 1996, shows that developers tend to rush in their changes, or even do partial commits, to avoid dealing with merge conflicts. The fact that developers do not take their time to review the changes they performed, reduces the

quality of the software being built. Moreover, broken builds affect the work of the entire development team, thus causing planning and scheduling issues. This means that, by decreasing the likeliness of conflicts, awareness also has a positive effect on the system's quality and on the software process.

Developers become aware of the activity of others mainly through communication and inspection of code changes on SCMs [LVD06]. Since the publication by Curtis et al. in 1988, it is known that, in co-located development teams, informal meetings are the preferred communication means [CKI88].

The research done by Sarma et al. in 2006 and by Damian et al. in 2007 show that, in the context of global software development (GSD), awareness cannot be obtained with the same methods applied in co-located teams, as distance makes informal meetings impossible and differences in the organizational culture make text-based communication unreliable: Different teams are used to communicate the same kind of information on different channels: (e.g. discuss about bugs in mailing lists or in bug tracking systems) [DISK07, SvdH06].

Since the beginning of the 21st century, various researchers have focused on improving the implicit communication provided by SCMs, instead of improving explicit communication channels [BCSR07, dSCdW⁺06, HP08, SRvdH08, SGPP04, Sch01].

The common approach consists of notifying developers, in real-time, about changes performed by the other team members and alert them of emerging conflicts. These contributions share two drawbacks: They either treat source files as plain text, thus losing important information, or they discard the collected information right after use.

In 2009, Hattori presented *Syde*, a set of *Eclipse* plugins, which provide real-time notifications of changes and alerts of emerging conflicts [HL09b]. *Syde* records change information in a dedicated change repository, which has been used, in conjunction with SCM commits, by Hattori and Lanza to run code ownership analyses [HL09a]. Continuing her research, Hattori improved *Syde* to record more fine-grained change information up to the level of field and method changes, thus providing more precise conflict alerts [HL10].

During her master thesis, Anja Guzzi created *Scamp*, an *Eclipse* plugin to improve awareness through visualization [Guz09]. *Scamp* builds on top of *Syde*'s notifications to produce three different visualizations that give developers an insight into the recent activity of the team, regarding the most active modules and the most active developers.

To take full advantage of change information, an understanding of the system's structure is required. We believe that visualizing change and conflict information together with the components of a system can increase awareness and provide insights on the evolution of the system. Besides implementing the basic features of *CodeCity* in an *Eclipse* plugin, we concentrate our efforts on connecting this plugin with *Syde*, in order to provide a structural view of the visualized systems, enriched with information about team-activity and emerging conflicts.

Chapter 3

Manhattan

Manhattan is a software visualization tool, implemented as an *Eclipse* plugin to encourage its integration in the development process. The goal of this tool is to support developers to drive the evolution of software systems by combining software visualization and collaboration support techniques. We intend to help developers reasoning about and understanding software systems by producing visual representations of their structure.

To support collaborative development, we make developers more aware of the activity of their colleagues, in order to improve coordination, avoid duplicated work, and reduce the frequency of merge conflicts on SCM repositories.

We implemented these ideas in *Manhattan*, which provides a unified view on software systems, focused on their structure and on the activity of the developers.

Before describing in detail our approach, we present an imaginary use-case for *Manhattan* to contextualize later explanations and give an overview on some of the available features.

We advise to read this chapter from a color-enabled support, as we make extensive use of color images, and the ability to distinguish colors is important to understand the concepts we discuss.

3.1 Contributing to Commons Math - a Use-Case for Manhattan

In this section we describe how the developer Bob can benefit from our tool to contribute to a project he is not familiar with - *Commons Math*¹. Bob wants to add to the class *GeneticAlgorithm* an instance method that prints configuration information.

As a first step, Bob checks out the project from the Apache repository and visualizes it with *Manhattan*. Figure 3.1 shows *Manhattan* running inside of Bob's Eclipse instance. The Eclipse view where we render our visualization is on the top-right of the screen.

We visualize software systems as interactive 3D cities, where buildings represent classes and districts represent packages. The number of fields and methods in a class determine the size of its building. Since Bob is seeing the city for the first time he has no idea of which classes are represented by which buildings; therefore he explores the city for some time to build a mental map to orient himself in the visualization.

While moving around the city, Bob notices a very wide and short building (label **1** in Figure 3.2) and hovers the mouse on it to see a tooltip description of the corresponding class. From the name of the containing package, Bob understands that this class contains functionality to deal with exceptional behavior: By exploring the city, Bob is also exploring the project.

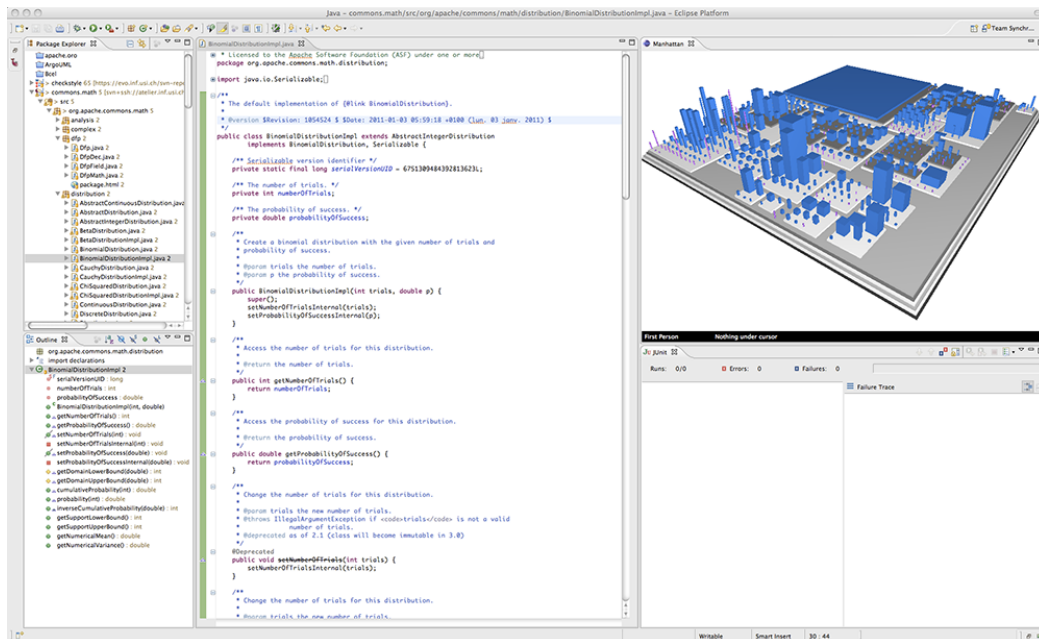


Figure 3.1. Manhattan running inside of Eclipse

¹<http://commons.apache.org/math/>

Bob notices some other interesting buildings (2) and right-clicks on them to open the corresponding classes in the Eclipse editor. By combining exploration with code inspection of “important” classes, Bob quickly gathers the structural information required to implement his contribution.

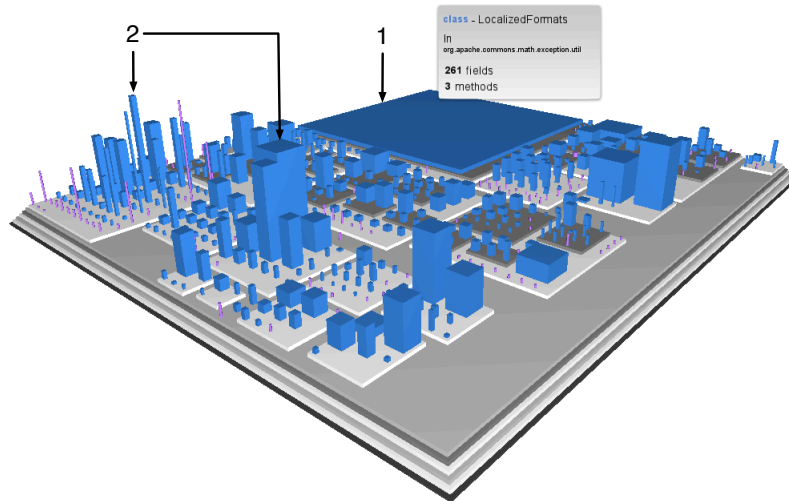


Figure 3.2. The tooltip for class *LocalizedFormats*

In the meantime, other contributors to *Commons Math* begin to work on the system and Bob is visually notified of their activity as shown in Figure 3.3. The buildings in yellow represent classes that have been modified by other developers since Bob started his development session. As nobody is working on the class he needs to modify (*GeneticAlgorithm*), he begins working on his contribution.

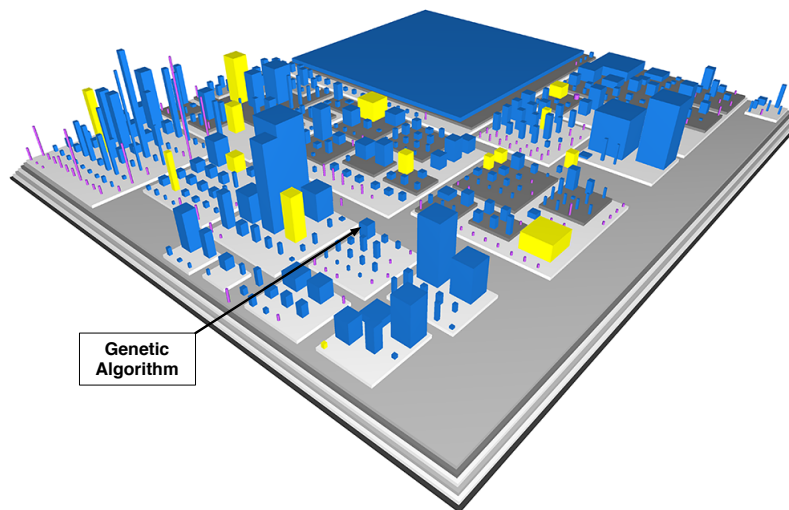


Figure 3.3. The activity of contributors to Commons Math

Nevertheless, when the contribution is almost complete, the building representing *GeneticAlgorithm* turns yellow and a sphere of the same color appears on top it. This sphere represents an alert for an emerging conflict: Another developer is concurrently modifying the class *GeneticAlgorithm*. Yellow spheres represent emerging conflicts, while red spheres indicate that one of the developers involved in the conflict has already committed his changes. Since the sphere is still yellow, Bob knows that the conflict can be resolved before it is committed to the repository.

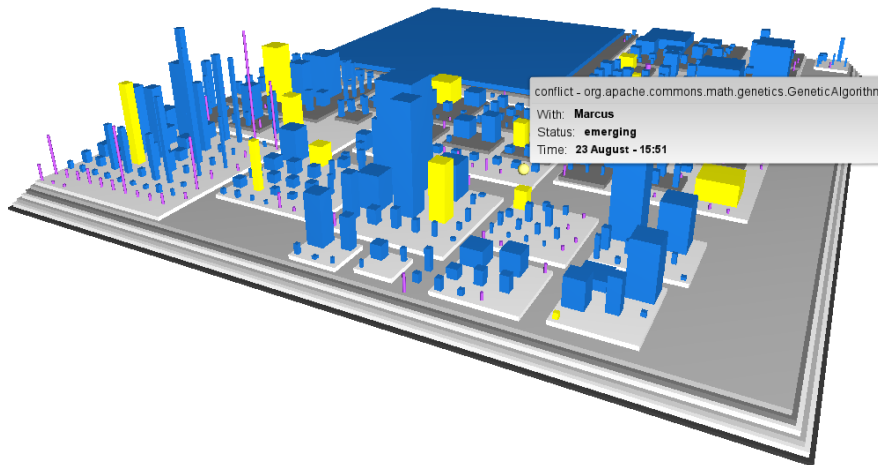


Figure 3.4. The tooltip for the emerging conflict on class *GeneticAlgorithm*

First he looks at the tooltip description for the conflict sphere (see Figure 3.4) to find out the username of the other contributor: Marcus. Then he gets in touch with Marcus and they begin to search for a strategy to solve the conflict. Bob inspects the conflict in an instance of the Eclipse compare editor (opened from within the visualization) and they discuss their respective contributions, while looking at each other's code. By looking at the compare editor they find the conflicting lines and modify them to solve the conflict. As soon as the conflict is resolved, the conflict sphere disappears. Moreover, during the discussion, Marcus informs Bob that he is adding a logging component to *GeneticAlgorithm*, therefore Bob changes the method he has added to make it work with this logging component. Finally they both commit their contributions without having to merge conflicts from the repository.

In the next sections we describe our approach in detail, explaining the rationales behind our decisions and how we deal with specific issues. First we explain our visualization of software systems, then we move to the visualization of developers' activity.

3.2 Visualizing Software Systems

3.2.1 The Code-Model

The source-code of a system contains all the information about the system's structure. However this information needs to be extracted from it into a representation at a higher level of abstraction, i.e. a code-model.

Depending on the level of detail, a model can be language independent, at the cost of having no notion of some language specific constructs. However, defining a model that works with all language paradigms requires to omit too much information. Since object-oriented languages are widely adopted, *Manhattan* focuses on object-oriented systems.

Figure 3.5 shows the artifacts and relationships described by the code-model used in *Manhattan*. The entities are the ones common to many OO language: packages, classes and methods. The modeled relationships are containment and inheritance: Method invocations are omitted, as they are not visualized. Classes can have multiple super-types in order to support mix-ins (used for example by *SmallTalk* and *Python*) and multiple inheritance (used by *C++*).

Although interfaces are a Java specific concept, the model distinguishes them from classes, as doing otherwise can be misleading when analyzing Java systems. This issue is explained in more detail in Section 3.2.2, after we describe the city-metaphor.

When extracting the code-model of a system, for every class we compute the number of fields (NOF) and number of methods (NOM) metrics.

Every instance of the code-model refers to a project in the workspace and only the entities defined in the project are included (i.e. external classes required by project members are ignored).

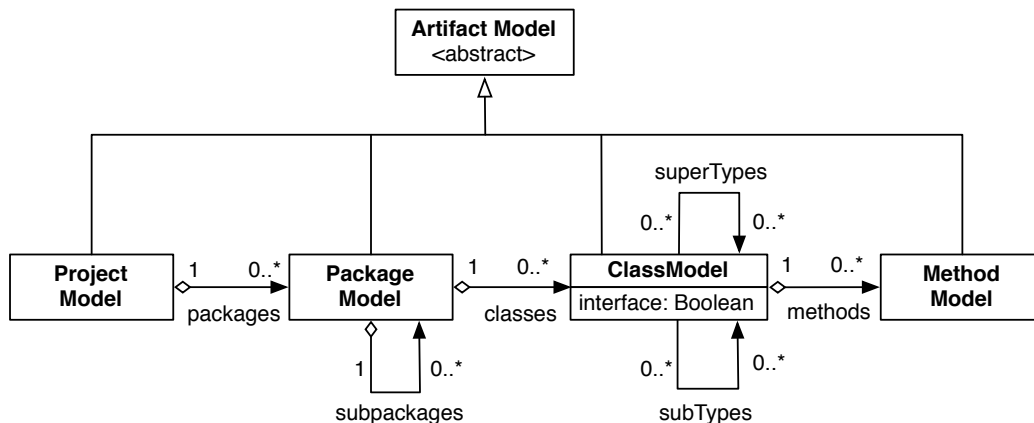


Figure 3.5. The code model used in Manhattan

To visualize a system written in a given language, *Manhattan* needs a language-specific parser that extracts the model from the system's source-code. The fact that *Manhattan* is an *Eclipse* plugin can simplify the implementation of such a parser: Most language-integration plugins (e.g. *JDT* for Java) build abstract syntax trees from source-code and provide an API to access them. By exploiting such APIs one can implement language-specific parsers without having to care about language-specific syntaxes. At the time of this writing, we have implemented only a parser for Java systems. This parser is built on top of *X-Ray*², a software visualization plugin, which uses the APIs offered by *JDT* to parse Java projects [Mal07].

²<http://xray.inf.usi.ch>

3.2.2 The City Metaphor

People use metaphors to explain concepts from an abstract or complex domain (target domain) by means of concepts that belong to a familiar domain (source domain). Software visualization metaphors describe software artifacts and their properties using concepts from a familiar more tangible domain. For a metaphor to be effective, the source domain needs to have some sort of strong similarity with the target domain, and concepts should be mapped from one domain to the other in the simplest and most direct way possible.

The metaphor used in *Manhattan* is a slightly modified version of the city metaphor introduced by Wetzel et al. I briefly describe the metaphor to contextualize our contribution; further details can be found in [WL07, Wet10].

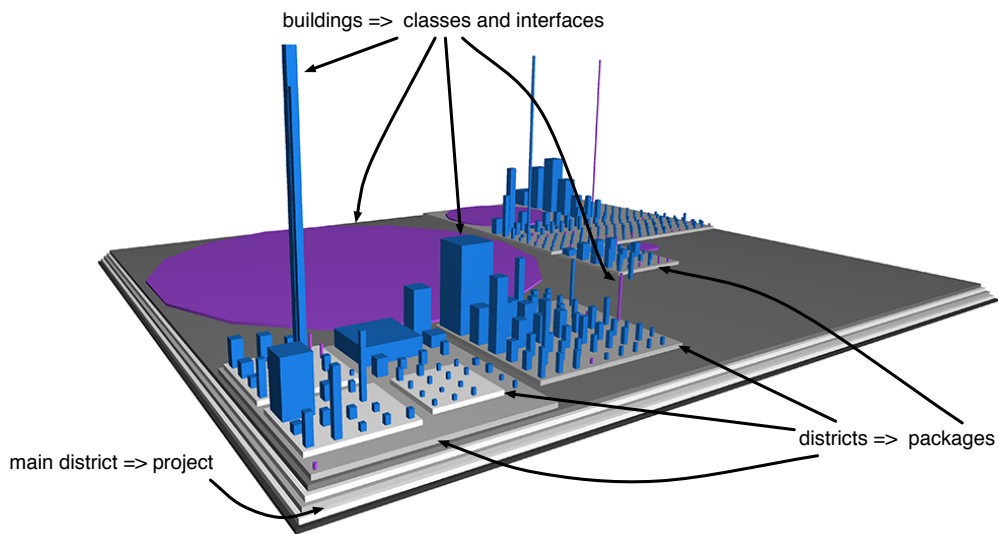


Figure 3.6. The city of Apache BCEL

The city metaphor maps architectural elements to software entities. Projects are represented as cities, packages as districts, and classes as buildings. Figure 3.6 shows the city for Apache BCEL, annotated with a key for the artifacts. The mapping from districts to packages is immediate, as both are members of a containment hierarchy and group related artifacts together. Representing classes as buildings is an effective mapping, since it links the first-class entities of the two domains. Besides defining analogies between concepts of the two domains, the metaphor introduces a set of mappings from software properties to visual properties of the architectural artifacts. These mappings are illustrated in Figure 3.7. Differently from Wetzel's metaphor, buildings representing Java interfaces are colored and shaped in a different fashion than the others. This allows users to clearly distinguish entities that contain behavior from those that only declare it.

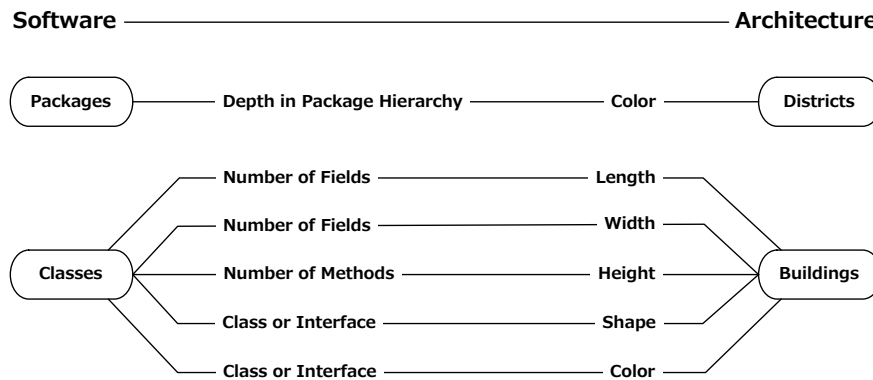


Figure 3.7. The mappings defined by the city metaphor

The described mapping produces three particular types of buildings shown in Figure 3.8. Skyscrapers are thin and tall buildings representing classes with few fields and many methods. Classes with many fields and few methods are represented by flat and wide buildings that we refer to as parking-lots. Office buildings are wide and tall and they represent classes with many fields and many methods. These types of buildings should be investigated first to understand the system under analysis, because (1) skyscrapers are classes that describe the behavior of a system or the interfaces it exposes, (2) parking lots represent entities that describe what kind of information is treated by the system (e.g. constants holders) and (3) office buildings contain information about both behavior and data. One of the main benefits of the city metaphor is that these kind of buildings are the first to be noticed also in very large and complex cities.

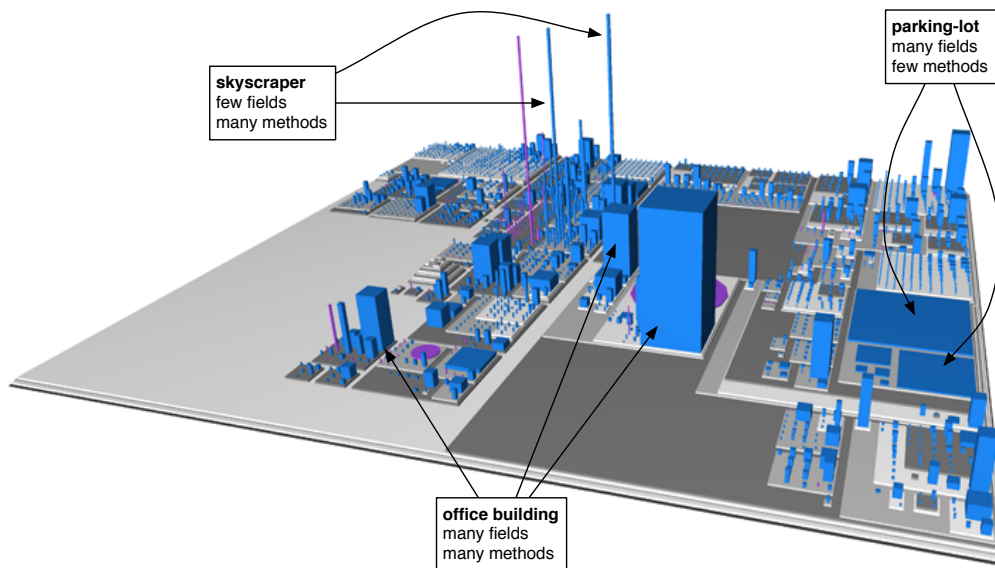


Figure 3.8. The city of ArgoUML

3.2.3 The View-Model

The first step to visualize a project is extracting its code-model. The view-model is a geometrical representation of the code-model in the form of a tree. Nodes in the tree are called glyphs.

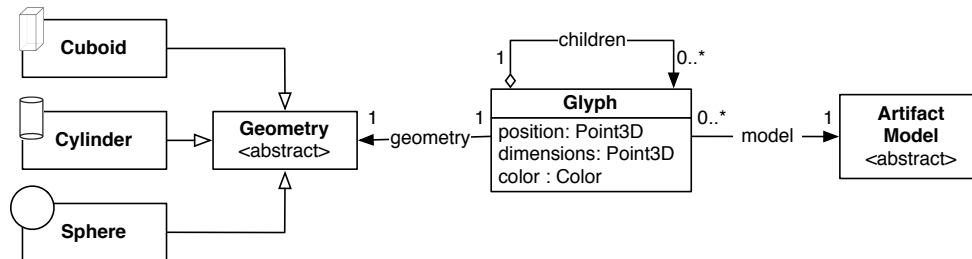


Figure 3.9. The view model

The view-model can be used to realize different metaphors, depending on (1) what entities in the code-model are represented by the glyphs, (2) what code-model relationship is represented by the *children* relationship in the view-model, (3) how the *children* relationship is visualized, (4) how the shape and size of the glyphs is computed, (5) and what layout algorithms are used to compute the position information of the glyphs.

To realize the city metaphor, glyphs are shaped as cuboids and cylinders, and they are used to represent projects, packages, classes, and interfaces. The *children* relationship between two glyphs represents different code-model relationships, depending on what code-model entities are represented by the two glyphs.

Nevertheless, in the city metaphor the *children* relationship only reflects the *containment* relationship between projects, packages, and classes; moreover the relationship is not shown with a dedicated visual element, but by putting the children (containees) of a glyph *g* (container) on top of it. In addition, the area of *g* depends on the surface occupied by its children, since the city layout is hierarchical.

Besides the city metaphor, many other metaphors are based on a hierarchical layout (e.g. all the metaphors that draw a tree structure). For this reason, glyphs are designed to be responsible for laying out their children. To execute a hierarchical layout, it is necessary to start from the leafs and climb the glyph tree up to the root, while running the layout algorithm of choice (that we describe in Section 3.2.4) on the children of every encountered node. After the layout phase is complete, the visualization can be rendered; our rendering strategy is described in Section 3.2.7.

Figure 3.10 shows a city and the underlying glyph tree side by side. The root of the tree is the city base and the immediate child of the root is the project. Every descendant of the project glyph is either a district (package) or a building (class), depending on whether it has children or it is a leaf node.

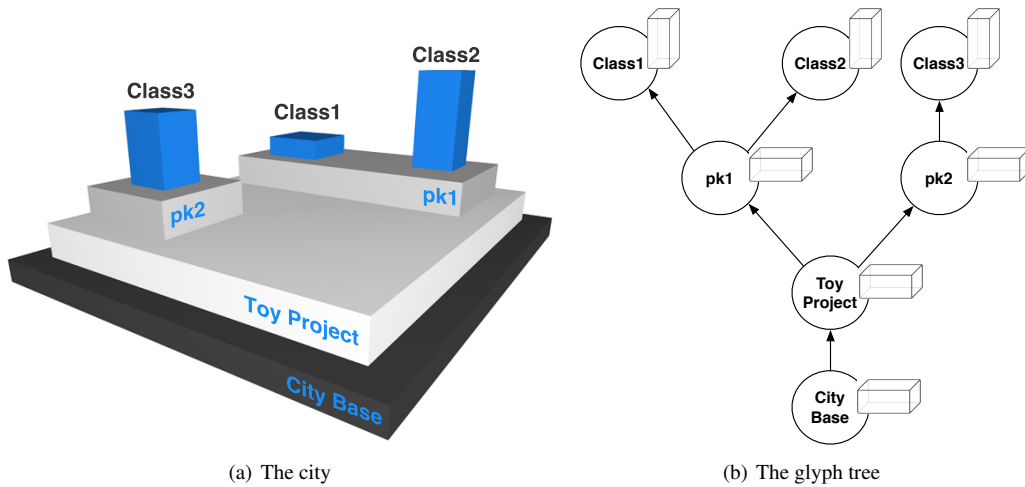


Figure 3.10. The city for a simple project (a) and the corresponding glyph tree (b)

3.2.4 The Layout Algorithm

The problem of finding a layout for a city can be reduced to the *rectangle packing* problem³, consisting of placing a set of rectangles of varying dimensions into a rectangular container, while minimizing the size of the enclosing container⁴. This problem has many real-world applications, for example in packaging, storage, transportation, computer graphics⁵, and chip manufacturing. Reducing the problem of computing a city layout to the *rectangle packing* problem is simple: The contents of a district are the rectangles to place and the district is the enclosing container. To create the layout algorithm for *Manhattan*, we re-implemented the *Rectangular-Packing* algorithm used by Wetzel in *CodeCity* and we added the support to real-time city updates triggered by code changes performed by developers.

³http://en.wikipedia.org/wiki/Packing_problem

⁴Although other algorithms can be used, minimizing wasted space improves usability as cities do not grow too large

⁵[http://en.wikipedia.org/wiki/Sprite_\(computer_graphics\)](http://en.wikipedia.org/wiki/Sprite_(computer_graphics))

The Rectangular-Packing Layout Algorithm

A thorough description of the algorithm can be found in Wettel's Ph.D. thesis [Wet10]. Here we give a high level description to provide the reader with a context to understand the issues introduced by the real-time city-updates and how we tackle them.

Given a list of rectangular items to layout, the algorithm computes a maximal container, in which all items fit, but with large unused surface. Then the items are inserted one after the other into a growing container that can't grow larger than the maximal one. Placing the items in this container means partitioning the maximal container in many lots: Every item is placed in a lot of its own, avoiding to overlap with other items. This partitioning is implemented with a two-dimensional *k-d tree*⁶. In this tree, leaf nodes are empty lots, while the other nodes represent a space partitioning. To place an item in the container, the algorithm searches all the leaves in which the item fits, in order to find a specific leaf *l* such that placing the item in *l* does not require the container to grow and wastes the least space possible. If no suitable leaf is found, the container is expanded by splitting an existing leaf.

If the collection of items is fixed, then the container will never grow larger than the maximal container and every element in the least will find a place. Nevertheless, if the collection of items can grow, the items added later to the collection, might not fit in the maximal container computed from the initial collection. When this happens, it is necessary to re-layout all the items.

In the next section we describe why in *Manhattan* both collections and individual items can grow, how this affects the usability and effectiveness of the visualization, and the approaches to tackle this issue. Figure 3.11, kindly provided by Richard Wettel, shows the execution steps of the algorithm.

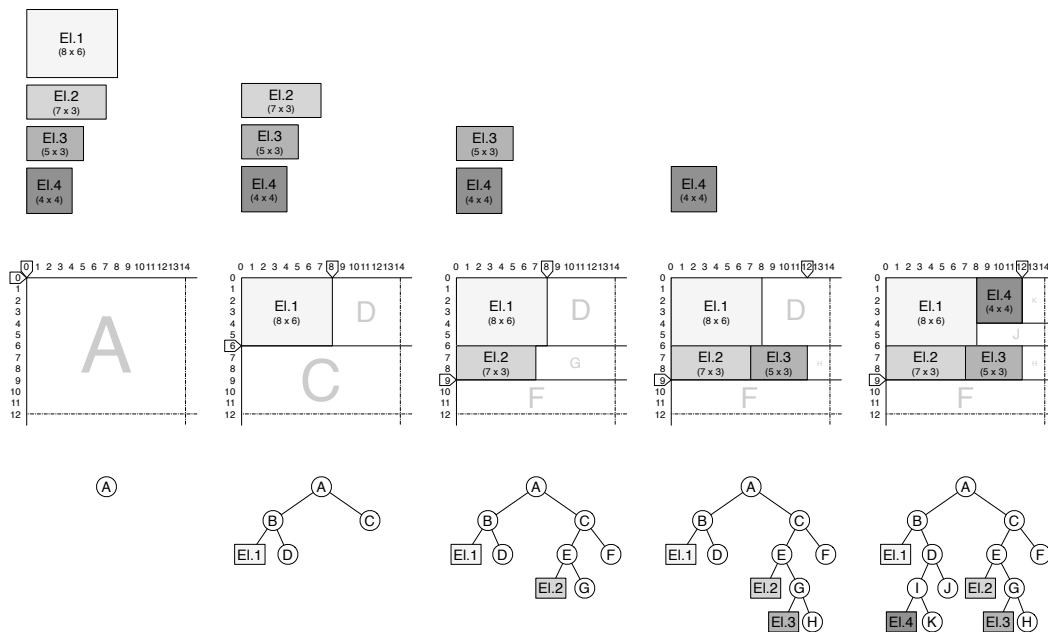


Figure 3.11. The execution steps for the Rectangular-Packing algorithm, provided by Richard Wettel

⁶http://en.wikipedia.org/wiki/K-d_tree

3.2.5 Reacting to Changes

The visualization we implemented eases program comprehension, thus it can support software maintainers in reverse-engineering tasks. However, we are more focused on helping developers to drive the evolution of software systems by visualizing their structure. To support evolution, the visualization should reflect the current state of the system, as developers are not concerned with how the system was in the past. Therefore, the visualization should update itself according to the modifications performed on the system. These modifications are not only the changes done by a developer to his working copy, but also the changes, done by his colleagues, that he checks out from the repository.

Eclipse provides a service to which plugins can register in order to be notified when a developer modifies the contents of a project. Using this service, *Manhattan* detects changes done to the system and updates accordingly both the code-model and the view-model.

View-Model updates are the insertion of new glyphs, the resizing of glyphs representing buildings, and the removal of existing glyphs. Insertions take place when a new package or class is created, resizings are the result of a change in the number of fields or methods in a class or interface⁷, and removals are caused by the deletion of packages or classes.

Insertions and resizings can modify the layout of the city, as a district d could become too small for its contents and a re-layout of the contents would be required. Moreover, since the city layout is hierarchical, d could in turn become too large for its container, which in turn could also become too large for its own container. This leads to a series of recursive re-layouts that, in the worst case, ends at the root of the the glyph tree (Figure 3.12). Removals can affect the city layout only if one tries to fill the empty space left in a district by compacting its contents with a re-layout.

Since, while navigating a city, a developer builds a mental map that he uses to orient himself and to quickly locate a particular district or building [Wet10], these re-layouts should be avoided as much as possible to avoid disorienting developers.

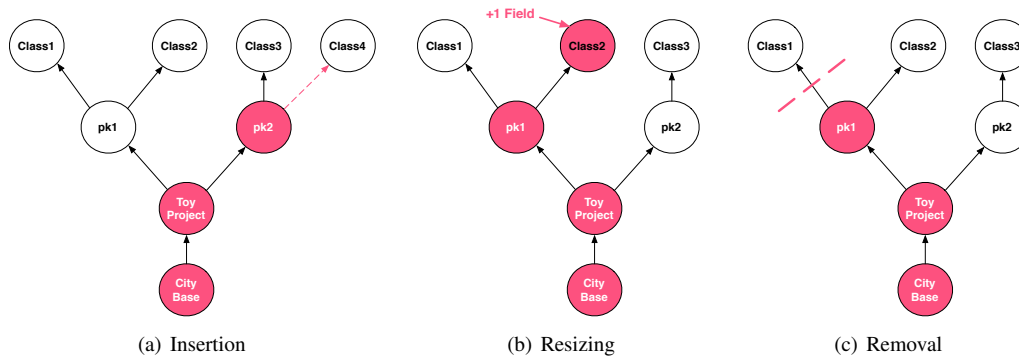


Figure 3.12. The glyphs affected by an insertion (a), a resizing (b) and a removal (c)

⁷Only changes in the number of fields can change the layout, as the number of methods does not affect the area of buildings

The *Rectangular-Packing* algorithm is not the best choice for an evolving hierarchical layout, like the one we described. In fact, the ideal algorithm should allow a city element to grow and push the elements around it, so that the city layout undergoes only minimal changes. The *corner-stitch* layout⁸ fits the requirements, but we have not found an open-source implementation of it, and implementing it was not a viable solution given the time constraints, as the algorithm is very complicated.

To mitigate this layout issue, we implemented a workaround to reduce the number of cases where the growth of a glyph causes a re-layout. This workaround consists of setting two growth tolerance thresholds based on the margin around a glyph and the padding inside of a container glyph, as shown in Figure 3.13. A re-layout happens only if a glyph overflows the tolerance area defined by the two thresholds. To further decrease the number of re-layouts, we tackle in the specific the three kinds of view-model updates.

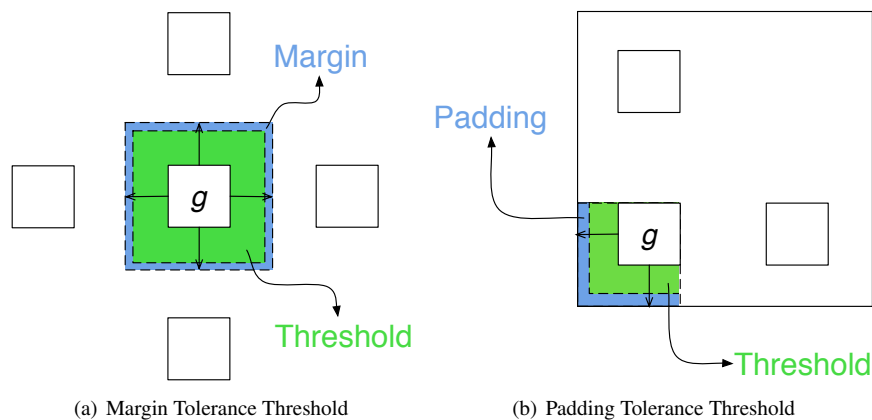


Figure 3.13. The growth tolerance thresholds used to decrease the number of re-layouts

Insertion

To insert a new glyph g , the first step is to attach it to its intended parent and container glyph c , which represents either a package or the project itself. Then it is necessary to compute g 's position and the new dimensions for c . The easiest way to do this is to re-layout c , but we want to avoid this. A better way is exploiting the *Rectangular-Packing* algorithm by trying to insert c in the *kd-tree* of g : If this succeeds, the container is either unchanged (best case) or expanded (worst case). If there is no leaf in the *kd-tree* that can contain c , then the only option left is to re-layout g . Both the re-layout and the worst case of a successful insertion in the *kd-tree* cause g to grow. In case that the growth is larger than the tolerance thresholds, a recursive re-layout routine starts from g 's parent.

Resizing

This update is triggered by changes in the number of fields or methods in a class or interface. When a glyph g is resized, a recursive re-layout is started if g 's growth is beyond the tolerance thresholds. In case that g has shrunk, no action is taken, as a little waste of space is preferred over a re-layout.

⁸The complete publication is available at: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1982/6352.html>

Removal

The first step to remove a glyph g is detaching it from its parent p . Then g is removed from its containing leaf in p 's *kd-tree*. No other actions is taken, so that a re-layout is avoided and an empty space is left instead of g . This solution is very appropriate as another glyph can be later placed in this empty space: Like in a real city a construction site is created where an old building was previously demolished.

The presented workarounds are designed to stop, as deep as possible in the glyph tree, the sequence of recursive re-layouts shown in Figure 3.12, so that the number of glyphs involved in the sequence is reduced as much as possible. However, the impact of a view-model update on the city layout mainly depends on the district in which the update takes place and on the update itself: If an update inside of a minor district forces a re-layout, the city structure undergoes only small changes; but if an update taking place in one of the main districts causes a re-layout, then the impact on the city structure can be substantial.

We have tested these workarounds by modifying different systems in different ways and we observed the following results:

- With the tolerance thresholds we defined, re-layouts do not take place until more than four fields have been added to a class since the last time that its container was laid out. Such an increase in the number of fields frequently happens in a young system. However, in more mature systems, such a change can be part of a refactoring or a more important change in the design.
- Insertions in small districts often cause re-layouts, while larger districts usually have some unused space available for new glyphs.
- Removals do not affect the city layout, as we take no layout-related action when performing this kind of update.

3.2.6 Caching

If the view-model for a project is recreated every time that the system is visualized, the city is laid out from scratch, thus disorienting developers and making the presented workarounds useless. Therefore, we have implemented a simple caching mechanism that saves the view-model to a file using the default Java serialization protocol. Since the view-model is a representation of the code-model and it depends on the information contained in this model, we also cache the code-model in the same way. Every plugin has access to a dedicated hidden directory located under the *.metadata* folder inside the workspace. Since we store the cache in this dedicated folder, there is a separate cache for every workspace.

Eclipse provides a hook that plugins can use to trigger the execution of a given routine right after the IDE has finished booting. A similar hook allows to execute routines before *Eclipse* shuts down. We use these hooks to load our models from the cache at the beginning of the development session and to store them back when users quit *Eclipse*.

Besides maintaining the city layout consistent across *Eclipse* restarts, the cache also speeds up the visualization process, as the code-model does not have to be extracted every time that a system is visualized.

3.2.7 Rendering the Visualization

As mentioned in Section 3.2.3, glyphs have the shape, color, position, and dimension information needed to render the city. However, the view-model is not designed to be responsible for the rendering, in order to ease moving *Manhattan* to another rendering framework.

Our design supports only scene-graph-based rendering frameworks, because having a node in the scene graph for every node in the glyph tree yields various benefits, among which making it easier to deal with user interactions on the city elements (e.g., hovering the mouse on a building to see its tooltip description).

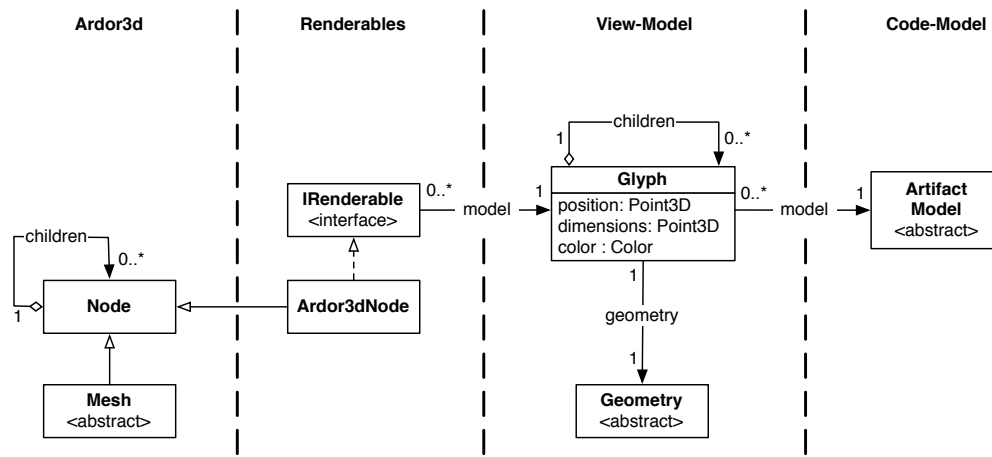


Figure 3.14. The design we conceived to render the visualization

*Ardor3d*⁹ is the framework we are using at the moment. As shown in Figure 3.14, we have defined the *IRenderable* interface, which declares methods to manage color, position, and dimension information. Methods to manage textures, animations, and beacons (see Section 3.3.4) are also declared by this interface. The children relationship in the view-model is reflected by the children relationship in the scene graph. With this design, in order to use a rendering framework it is necessary to define a subclass of the framework's scene graph node class, and to make it implement *IRenderable*.

⁹<http://ardor3d.com>

3.2.8 Interacting with the Visualization

We render our visualization in an Eclipse view composed of a 3D canvas and a status bar, where the active navigation mode is indicated. Two navigation modes are available: *first-person* and *orbital*. Both navigation modes are controlled with the keyboard and users can switch the navigation mode by pressing O.

Using the *first-person* mode, developers can freely move around the city, controlling a free-fly camera able to translate and rotate along the three dimensional axis. In this mode, users can observe the city, or a district, from the angle they prefer: They can either “walk” in the city, move further away from it, or stand on the roof-top of a building.

In the *orbital* navigation, we place an invisible dome on top of the city. This dome covers the whole city and is centered on it. The camera is attached to the dome and can move along its surface.

By moving along the dome, users can orbit around the city and quickly move to another observation point. When using this mode, the camera rotations are blocked.

While navigating the city, developers need to know what class is represented by a certain building. By hovering the mouse on the building, users can get a tooltip description of the represented class (Figure 3.2). Moreover, a textual description is shown in the status bar. If a user needs more information about a class than what is provided in the tooltip, he can right-click on its building to open the class in the Eclipse editor.

During a development session, a programmer usually works on a limited amount of code artifacts related to each other (e.g. members of the same package or members of packages that are conceptually related, like model and GUI), therefore he might prefer to visualize only these artifacts instead of the whole project. Based on this observation, we added the ability to build focused cities that contain only those parts of the city on which the programmer is going to work. Users first have to select the districts and buildings they are interested in by left-clicking on them, then they can build the focused city by pressing F. The selection step and the resulting city are shown in Figure 3.15; selected districts are in green.

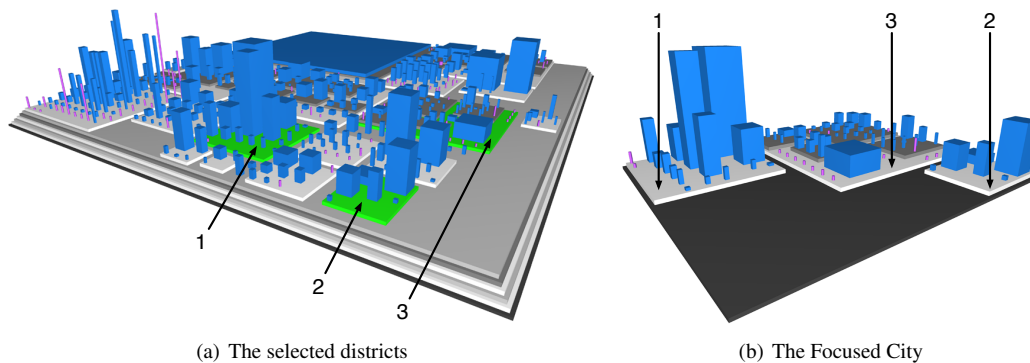


Figure 3.15. The selection step (a) and the resulting focused city (b)

3.3 Visualizing Team Activity

Much research done in collaboration support is focused on making developers more aware of the activity of their colleagues. Awareness gives various benefits for collaboration, like, for example, avoiding duplicated work and conflicting changes. The visualization implemented in *Manhattan* provides a unified view on software systems, focused on their structure and on the activity of their development teams. In this section we describe our approach to improve awareness through visualization.

As stated in Section 2.2, awareness is mainly achieved through communication and through inspection of changes in commits on SCMs. Nevertheless, research shows that the communication means used by co-located development teams are not suitable in the context of global software engineering (GSD). Moreover, SCMs are not able to provide a sufficient level of awareness by themselves, because: (1) they ignore important semantic information by treating source-code as plain text, and (2) the way in which they propagate changes does not reflect the dynamics of collaborative software development.

From an awareness perspective, treating source-code as plain text is inappropriate, because it forces developers to read the textual changes in order to derive how the system was modified by their colleagues (relying on commit comments is not feasible, as they either contain information at a too high level of abstraction, or they do not mention all the changes in the commit).

In the change propagation strategy employed by SCMs, a developer can see the changes performed by his colleagues only after they are published to the repository and he retrieves them from it. Because of this, some developers might work on the same piece of code without knowing, leading to merge conflicts that, depending on the nature of the conflicting changes, may hinder the work of many developers.

Various researchers focused their efforts in developing systems that improve awareness by providing developers with real-time notifications of changes performed by their colleagues and with alerts of emerging conflicts.

Section 2.2 describes two important drawbacks shared by these contributions: They either treat source-code still as plain text, or do not record the collected fine-grained change information.

The Hattori's approach implemented in a tool called *Syde*, does not suffer from these drawbacks. In the next section we briefly describe this tool, on top of which we based our approach to improve awareness.

3.3.1 Syde

Syde, which Hattori is developing in the context of her Ph.D. research, provides developers with real-time change notifications and alerts about emerging conflicts, from within *Eclipse*.

Syde focuses on Java systems and describes them with a language-dependent model aware of packages, files, classes, fields, and methods. Since this model represents a developer's local copy of the system, there is one model instance per developer, like, in the context of SCMs, every developer has his own working copy. Code changes are modeled as *change operations* that represent additions, deletions, and modifications of the entities in the system model. Real-time change notifications, based on the detailed information contained in these two models, provide a higher level of awareness than the diffs and commit comments provided by SCMs.

The ability of awareness to reduce the amount of conflicts relies on developers' cognitive activities, which can be compromised by external factors (such as fatigue); therefore early conflict detection should be automated to avoid that conflicts remain unnoticed until they show up in SCM repositories.

By modeling the working copy of every developer and by tracking all change operations in real-time, *Syde* can automatically detect conflict events and notify developers about them. Three conflict events are considered: (1) a conflict emerges, (2) an emerged conflict is committed to the SCM repository, and (3) a conflict is resolved. Since conflicts are detected early, they can be resolved before they are committed.

Syde is implemented as a client-server application as shown in Figure 3.16. The client is a set of *Eclipse* plugins grouped in two components: *Inspector* and *Viewer*. Whenever a developer modifies the code, the *Inspector* creates the change operations representing his change and sends them to the *Syde* server. The server is in charge of (1) recording the received change operations in the *Change Repository*, (2) broadcast these change operations to the interested clients, (3) detect conflicts and warn involved clients about them, and (4) provide clients with an interface to query the *Change Repository*.

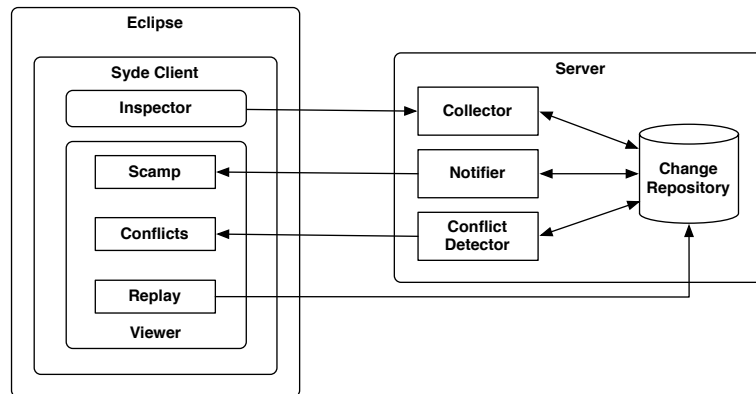


Figure 3.16. An architectural view of *Syde*

Change notifications and conflict warnings are sent by the server to the *Viewer* component in the clients. This component contains plugins that exploit the information sent by the server to provide developers with awareness information with different focuses and in different ways: The *Conflicts* plugin is focused on making developers aware of conflicts, while *Scamp* informs developers about the overall activity of the team on the system's classes. *Conflicts* presents information in a list-based manner, while *Scamp* employs simple 2D visualizations based on typography and color [Guz09]. *Replay* is radically different from the other viewers, as it ignores messages from the server, and allows developers to review the changes performed on the system, by querying the *Change Repository*.

3.3.2 Awareness in the City

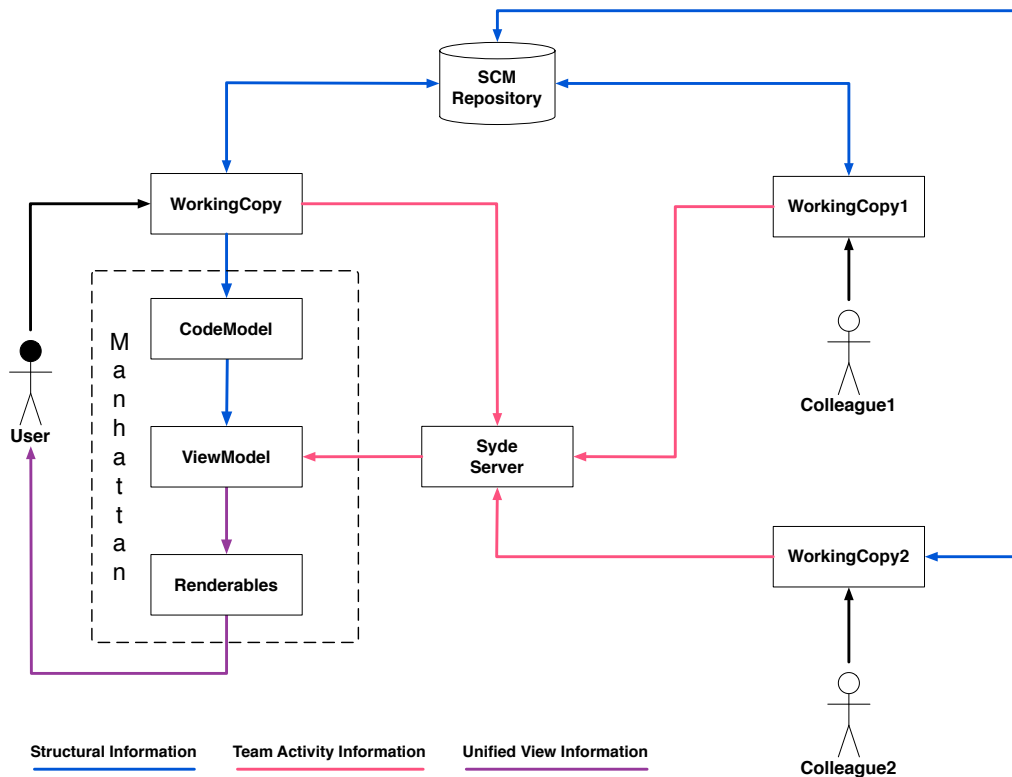


Figure 3.17. The information flow defined by our unified view

In the following sections, we describe our approach to visualize team activity. To better contextualize later explanations, we briefly comment the scheme in Figure 3.17, which shows the information flow defined by our unified view for a team of three developers¹⁰.

Developers modify their working copies (black arrows) and share their modifications with each other by checking them in and out of the repository.

Modifications to the working copy, either performed by the user or caused by a check out from the repository, are propagated to the *CodeModel* and reflected in the *ViewModel* and *Renderables*.

At the same time, the *Inspector* plugin¹¹ detects changes to the working copy in real-time and sends them to the *Syde* server, which then broadcasts information regarding changes and emerging conflicts to all interested clients.

In the context of *Syde*'s architecture, *Manhattan* is a viewer plugin interested in change notifications and conflict alerts sent by the *Syde* server. The team activity information coming from the server is combined, in the *ViewModel*, with the structural information coming from the *CodeModel* to create the unified view that is then rendered on the *Renderables*.

Finally, the rendered visualization feeds back information to the user, who will act accordingly.

¹⁰Also the two colleagues are using *Manhattan*, but this is not shown in the scheme for the sake of simplicity

¹¹See the architectural description of *Syde* in Section 3.3.1

3.3.3 Visualizing Change Notifications

In the context of *Syde*'s architecture, *Manhattan* is a viewer plugin interested in change notifications and conflict alerts. The awareness information it visualizes consists of (1) the emerging conflicts in which the developer is involved, (2) classes that have been changed or deleted¹² by the other developers, and (3), among the developers that modified a class, who is the one that modified it more frequently. Before describing the strategies we employ to visualize this information, we describe the rationales behind them.

From the point of view of a single developer, the state of the system is the code in his working copy and the changes he is performing. Nevertheless, the developer sees only a local state of the system, which does not take into account the changes being performed by the other developers. The union of the local state of every developer creates the global system state, which includes all the changes being performed by the development team and the conflicts that may arise from these changes.

With the information received from *Syde*, *Manhattan* has access to the global system state and can visualize it. To represent this state, the visualization seen by a developer should react not only to his changes, but also to the changes performed by his colleagues. For example, if a developer adds a method to a class *c*, his colleagues should see the building representing *c* grow taller¹³. Nevertheless, we believe that such a visualization would disorient developers, as we are convinced that they are used to reason on the system in the perspective of the local state they see. More precisely, our hypothesis is that, unless developers check out changes from their colleagues, they expect the system to change only according to changes they perform on their working copy.

Based on this hypothesis, we opted for a less invasive approach, where we ignore change notifications about the creation of new classes by other developers, and we map the remaining change information to color, a visual property of buildings that is marginally used in our metaphor. Although *Syde* distinguishes between field changes and method changes, we believe that such distinction improves awareness only if the displayed information is detailed enough, i.e.: for every developer that modified a given class, there should be a detailed description of which fields and which methods he changed, added or removed. Such a detailed description can not be shown with color alone and in a textual form it would clutter the visualization and cause information overload. Therefore we treat both fields and method changes as modifications to the class they belong to.

¹²Deletions of packages are also visualized

¹³In the city metaphor, the height of a building depends on the number of methods inside the class or interface it represents

Figure 3.18 provides an aerial view of the city for *Commons Math*, enriched with our color-based change notifications. With these notifications, we inform developers about which classes are being modified. When a class c is changed by one or more of his colleagues, developer d will see the corresponding building turn yellow. If one of d 's colleagues removes c from his working copy, c 's building will become orange.

A problem with this notification strategy is that if class c is modified multiple times, d might only notice the first change, because the later ones are not reflected by a visible color change, since the building has already become yellow. To solve this problem, we have conceived an approach that combines color notifications with texture changes and animations, but we have not implemented it, because of some shortcomings, in terms of scalability and visual noise, that we have not managed to solve yet. In Section 5.1, we describe this approach and its shortcomings in detail. Despite the described drawbacks, color-based notifications are still effective as they are clearly visible and scale well to large projects, as shown in Figure 3.19.

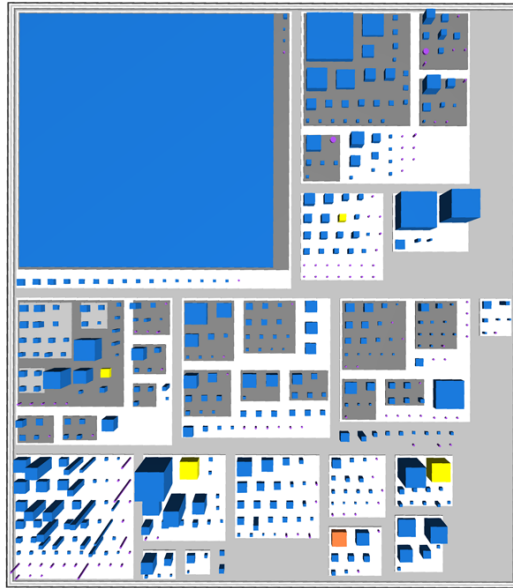


Figure 3.18. An aerial view of Commons Math showing notifications of developers' activity

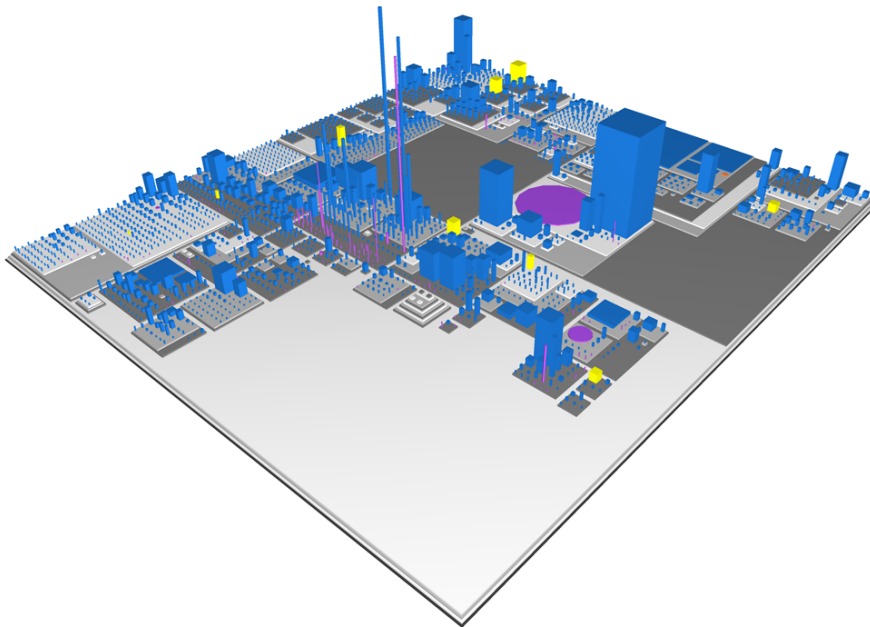


Figure 3.19. A bird's view of the city of ArgoUML, enriched with change notifications

To let developers know who modified a class, we list the change authors in its tooltip description, as shown in Figure 3.20. Developers are grouped by the kind of change they did and those that deleted the class are put first. Inside each group, developers are sorted first according to the number of changes done to the class and then alphabetically.

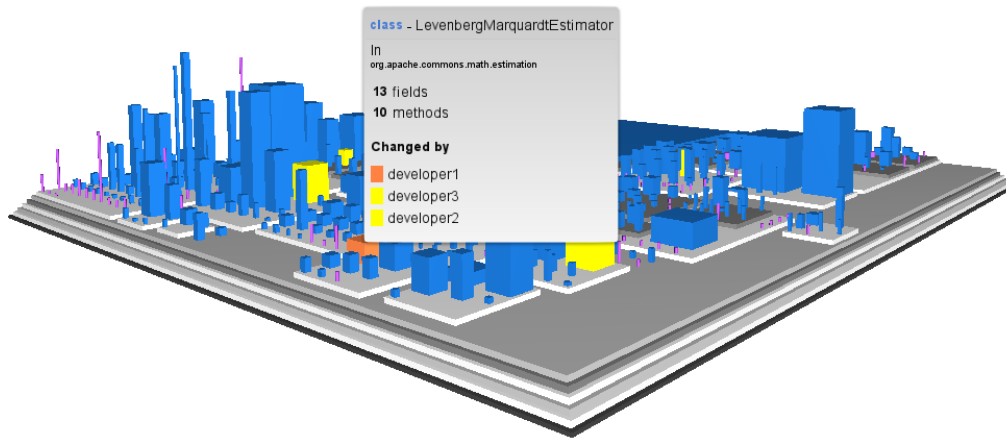


Figure 3.20. The tooltip description for a class modified by various developers

3.3.4 Visualizing Conflict Alerts

Syde broadcasts conflict alerts to all clients as it does for change notifications. However we visually notify developers only about the conflicts in which they are involved. To visualize a conflict alert on a class, we put a sphere on top of its building. The color of the sphere depends on the kind of conflict it represents: Emerging conflicts are shown in yellow, while committed conflicts, those where one of the involved developers has committed his changes, are shown in red.

While verifying the effectiveness of this notification strategy, we realized that depending on the size of the building and on its surroundings, the conflict sphere can blend in with the background, leading to unnoticed conflict alerts and, consequently, to conflicts on the repository. To solve this issue, we introduced the concept of *conflict beacons*. A conflict beacon is a spotlight positioned above of a conflict sphere and pointing towards the ground. These beacons illuminate an area of the city around their associated conflict spheres, so that conflict alerts are clearly visible. The algorithm that computes the height of the beacon is a function of the bounding box of the whole city, so that the beacon illuminates an area that is large enough to be noticed and yet with a well-defined proximity field (in which developers can search for the conflict)¹⁴. Figure 3.21 shows how conflict beacons allow to notice conflict spheres that are blending with the background: The buildings close to the conflict sphere are colored differently from the others because of the beacon, and this difference in color is clearly visible.

¹⁴The higher a beacon is placed, the larger is the area it illuminates

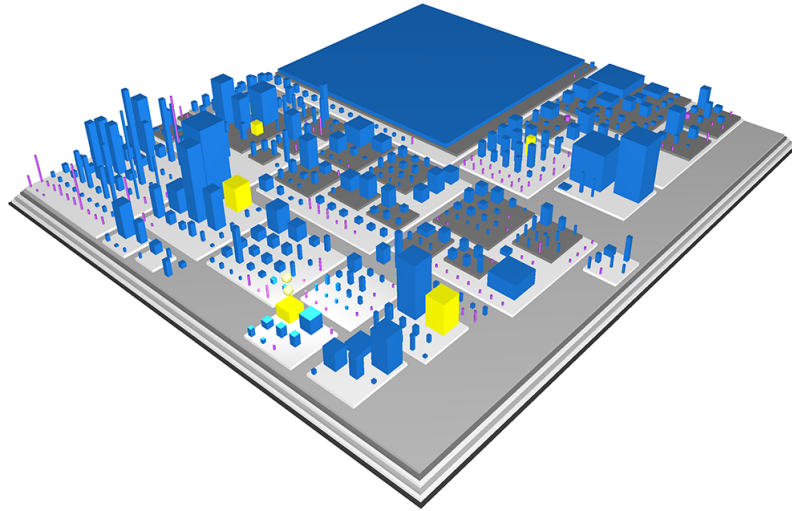


Figure 3.21. The city of CommonsMath with conflict alerts made visible using conflict beacons

If multiple developers concurrently modify a class c , each of them is involved in multiple conflicts on c . This situation is represented by a set of conflict spheres, one for each conflict, stacked on top of each other. Moreover, we still put one conflict beacon per conflict sphere, so that a stack of conflict spheres is highlighted by a stronger light than a single sphere. Putting the cursor on a conflict sphere deactivates its beacon and shows a tooltip description for the conflict (Figure 3.22). This description includes the state of the conflict (emerging or committed), the name of the other developer involved in the conflict, and the time of the last event on this conflict, that is the last time the conflict was committed or changed. A conflict changes when the involved developers modify the class and their changes do not solve the conflict.

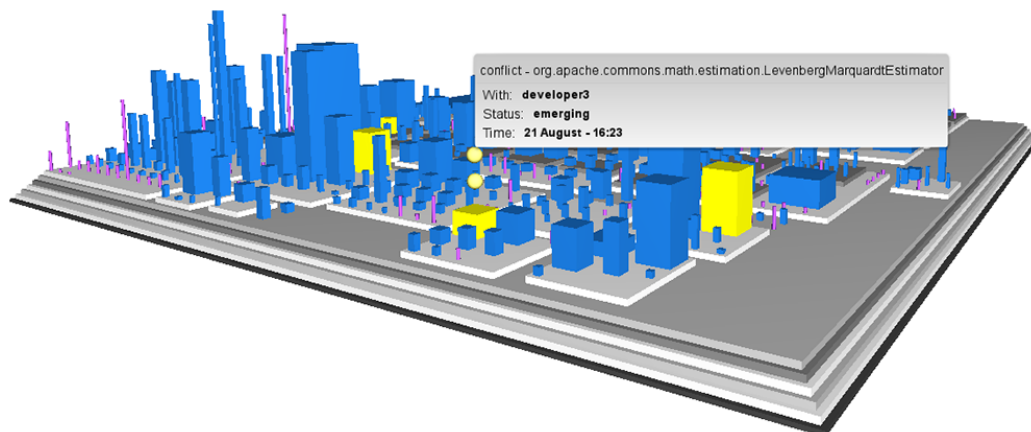


Figure 3.22. The tooltip description for a conflict sphere

When a developer discusses on how to solve a conflict on a class, the first step is to understand the modifications done by his colleague and his motivation. We ease this step by providing the possibility to inspect conflicts from within the visualization: Right-clicking on a conflict sphere opens an instance of the *Eclipse Compare Editor*, from where a programmer can see the differences between the version he has and the version in the working copy of his colleague. When the conflict is resolved, the corresponding conflict sphere is removed.

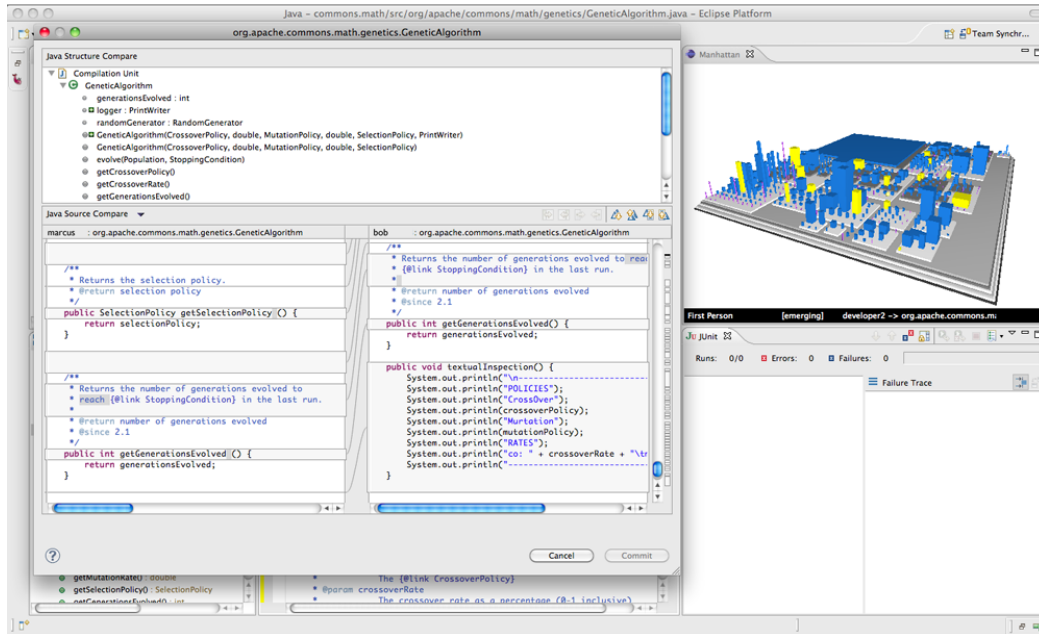


Figure 3.23. Conflict inspection in Manhattan

Chapter 4

Applications and Evaluation

In Chapter 3, we discussed our unified view on systems' structure and team activity. We have also illustrated a possible use-case for *Manhattan*, which shows its intended use and how we expect developers to benefit from the support it provides. In this chapter, we want to verify our approach by checking that the use-case we describe can actually fit in the users' workflow and that the visualization is clear and intuitive. Moreover, we also want to verify that the implementation scales to projects larger than *CommonsMath*, which we have extensively used in our tests while developing the tool.

In the next section we explain how we verified the implementation, while in Section 4.2 we describe an exploratory study we conducted to validate our approach.

We advise to read this chapter from a color-enabled support, as we make extensive use of color images, and the ability to distinguish colors is important to understand the concepts we discuss.

4.1 Exploring a few Case Studies

In order to verify the scalability of our implementation, we have selected a few systems of diverse size and belonging to different fields. We have visualized these systems and briefly explored the resulting cities to check that the tool remains responsive. To visualize these systems, we have used the same machine on which we developed our tool: A *MacBookPro 4.1*, with a 2.4 GHz Core2Duo processor and 4 GB of RAM.

We has selected the following systems: *ActiveMQ*, *Ant*, *Cobertura*, *jEdit*, *Vuze*, and *NetBeans*.

For each system we show a picture of its city and briefly comment about the system's architecture.

4.1.1 ActiveMQ

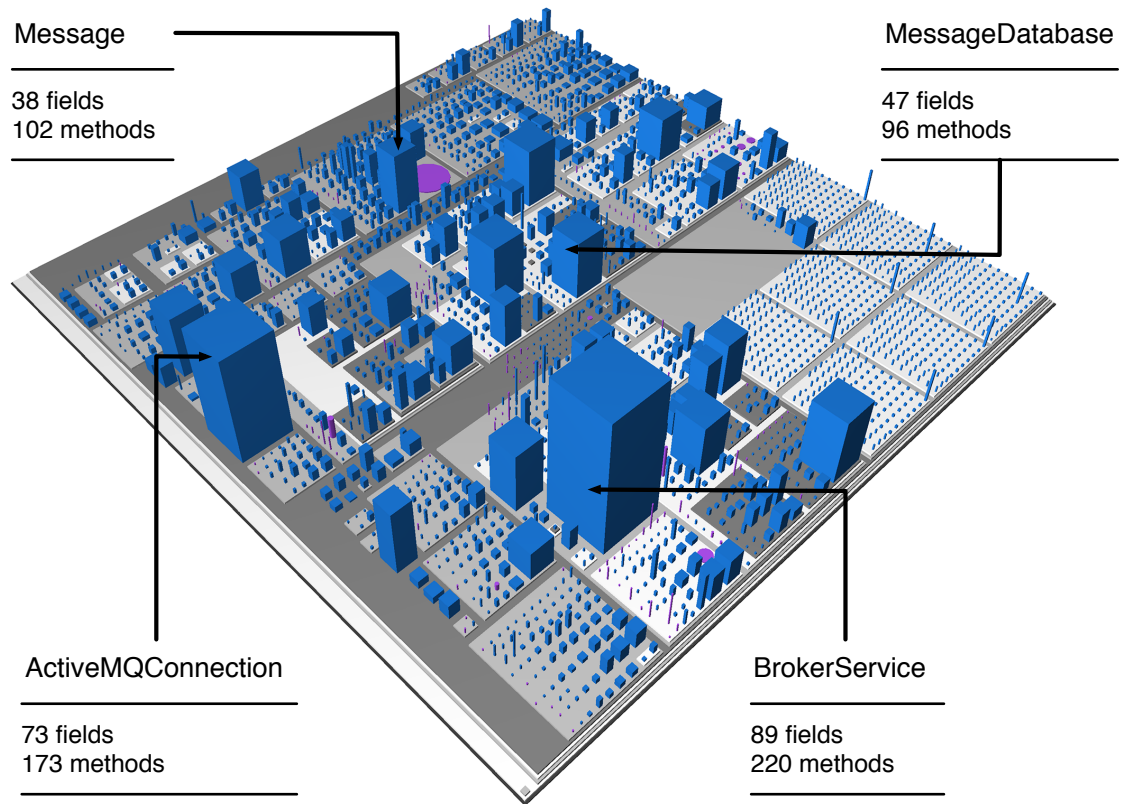


Figure 4.1. The city of Apache ActiveMQ

ActiveMQ is a message broker implementing *JMS 1.1*. It provides language specific clients for many programming languages and can be deployed on clusters. We visualized only the project's core, which consists of 2,456 classes, 185 interfaces, and 109 packages. There are many office buildings that dominate the city skyline. The average number of classes per package is more than 260, but, when looking at the city, it seems much lower. The reason for this skew in the average lies in the particular setting on the right edge of the city: Seven districts that look identical in size and structure and even contain the same number of classes (110). These districts are named as version numbers and their parent district is named *openwire*. *OpenWire* is a serialization protocol, native to *ActiveMQ*, that turns live objects from different programming languages into binary streams. The protocol evolves over time and seven version currently exist. We have not further investigated for the exact reason why a new package is "replicated" for every version. A possible explanation for this decision is the aim for backward-compatibility of new versions of the system with older deployments.

4.1.2 Ant

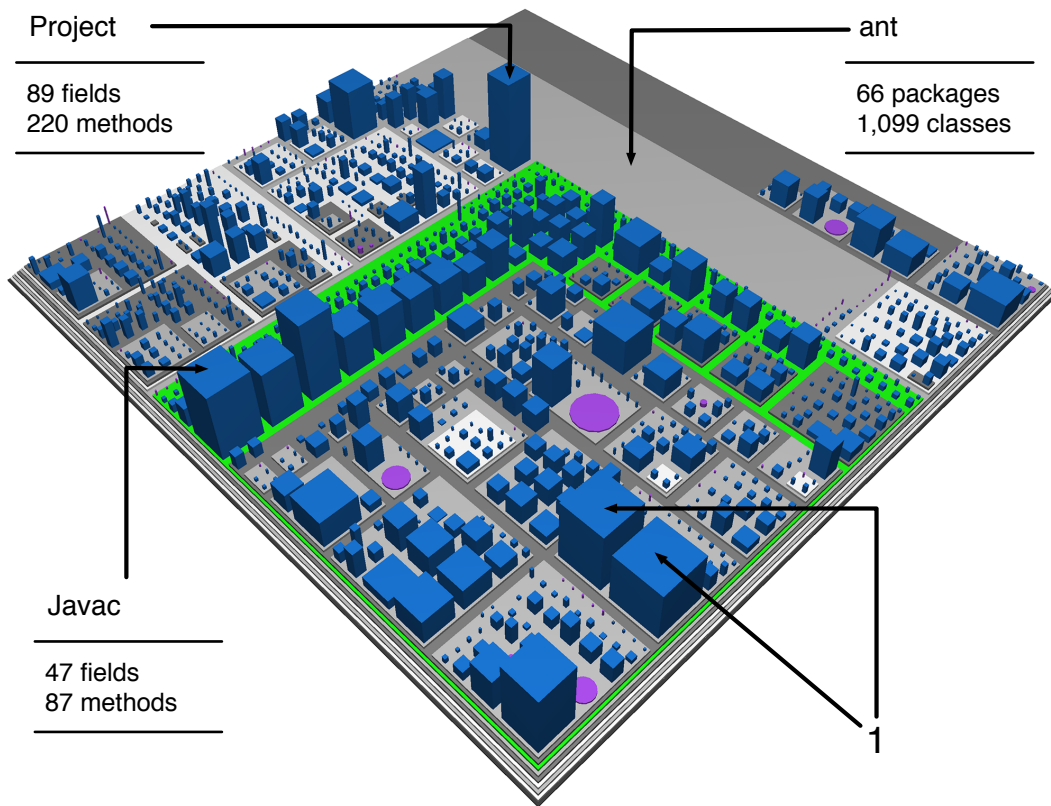


Figure 4.2. The city of Apache Ant

Ant is a tool to automate the execution of build processes for software systems. It is based on the concept of tasks, which are steps in build recipes called targets. The system contains 1,194 classes and 178 interfaces divided in 74 packages. The city consists of a main district named *ant* and a set of “satellite” districts. These districts contain classes to interface *Ant* with e-mail and the *bzip*, *tar* and *zip* archive formats.

In the *ant* district, there are many office buildings and almost all of them are located inside of the district we highlighted in green. This district is named *ant.taskdefs* and it contains the definitions for all built-in tasks that ship with the default distribution of *Ant*. This package contains more than half of the classes in the system and its district covers more than half of the city surface.

There are a few round parking-lots colored in purple: These are interfaces used as constants holders. Among these, *XMLConstants* contains the constants defining the properties of the XML reports produced by *JUnitTask*, a built-in task to automatically run test suites. On the bottom edge of the city we see two very similar and adjacent office buildings (1) named *FTP* and *FTPTask*. We inspected the classes for these two buildings and noticed that they have exactly the same Javadoc comment. It is not clear whether one is the successor of the other or if they have a different relationship.

4.1.3 Cobertura

Cobertura is a test coverage tool for Java systems. It consists of 23 packages, 110 classes, and 8 interfaces. This system is not as large as the other projects we are showing in this section, but it has some interesting peculiarities that we discovered while exploring its city.

The city is dominated by four massive skyscrapers having similar names and dimensions. The structure of the *parser* district is recursive: The same compound consisting of a parking-lot, a small office building and a skyscraper, is replicated four times. Moreover, using *diff* we have verified that the four parking-lots contain the same constants. While inspecting the skyscrapers we have found some lines, included in the comments for the code license, which state that these are automatically generated classes. We also discovered that the whole *javancss* package contains classes that belong to *JavaNCSS*, a project that computes various metrics for Java systems. We have not investigated the reason why these classes have been directly included in *Cobertura*.

Another interesting trait of this system is the small district (1) on the bottom left corner: *Cobertura* is used as input for its own test-suite, and this package is put in a different source folder to verify that *Cobertura* works properly on systems with multiple source folders.

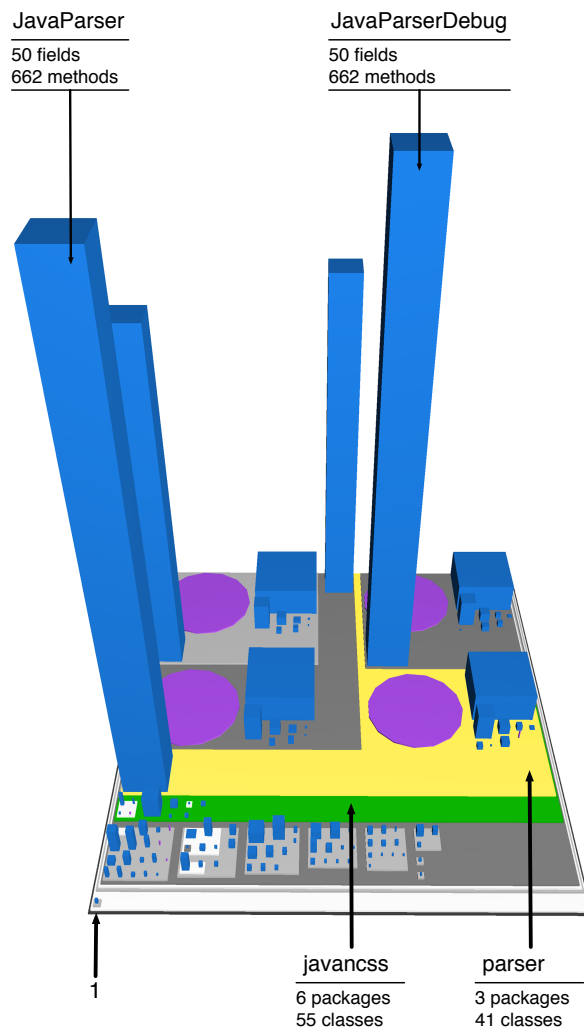


Figure 4.3. The city of Cobertura

4.1.4 jEdit

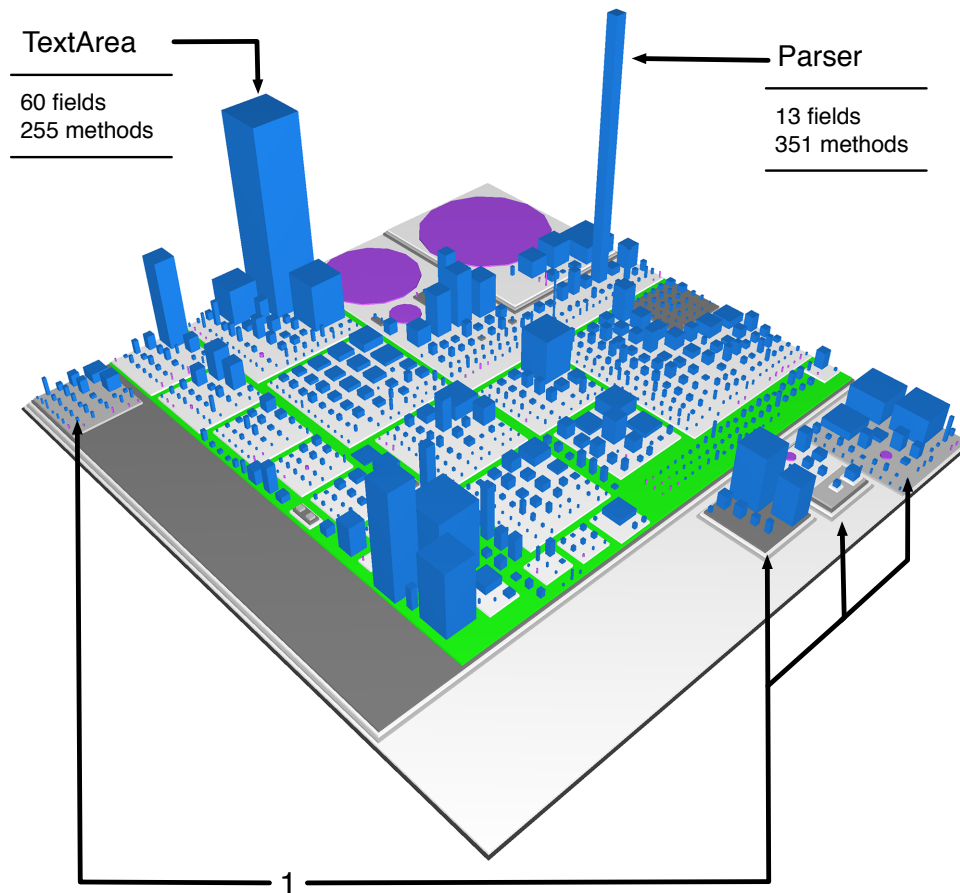


Figure 4.4. The city of jEdit

jEdit is a text editor, written in Java, featuring macros, extendible syntax highlighting for many file types, and many plugins that can be installed from a built-in plugin manager. The system consists of 927 classes and 67 interfaces organized in 50 packages. There are two skyscrapers and three parking-lots in the northern edge of the city. *Parser* is responsible, together with the other classes in its package, to parse and interpret *BeanShell*¹ based macros. All text editors providing advanced editing functionality and syntax coloring have to represent text with some object. Modeling every character in a text document with a live object would be an overkill. *jEdit* represents portions of text with instance of *TextArea*², which inherits from *JComponent*, a class belonging to *Swing*³. Outside of the *jedit* district, the one highlighted in green, there are four satellite districts (1) that contain a threading pool and classes to build package installers for OSX and Linux.

¹<http://www.beanshell.org/>

²More precisely it represents portions of text using *JEditTextArea*, a subclass of *TextArea*

³[http://en.wikipedia.org/wiki/Swing_\(Java\)](http://en.wikipedia.org/wiki/Swing_(Java))

4.1.5 Vuze

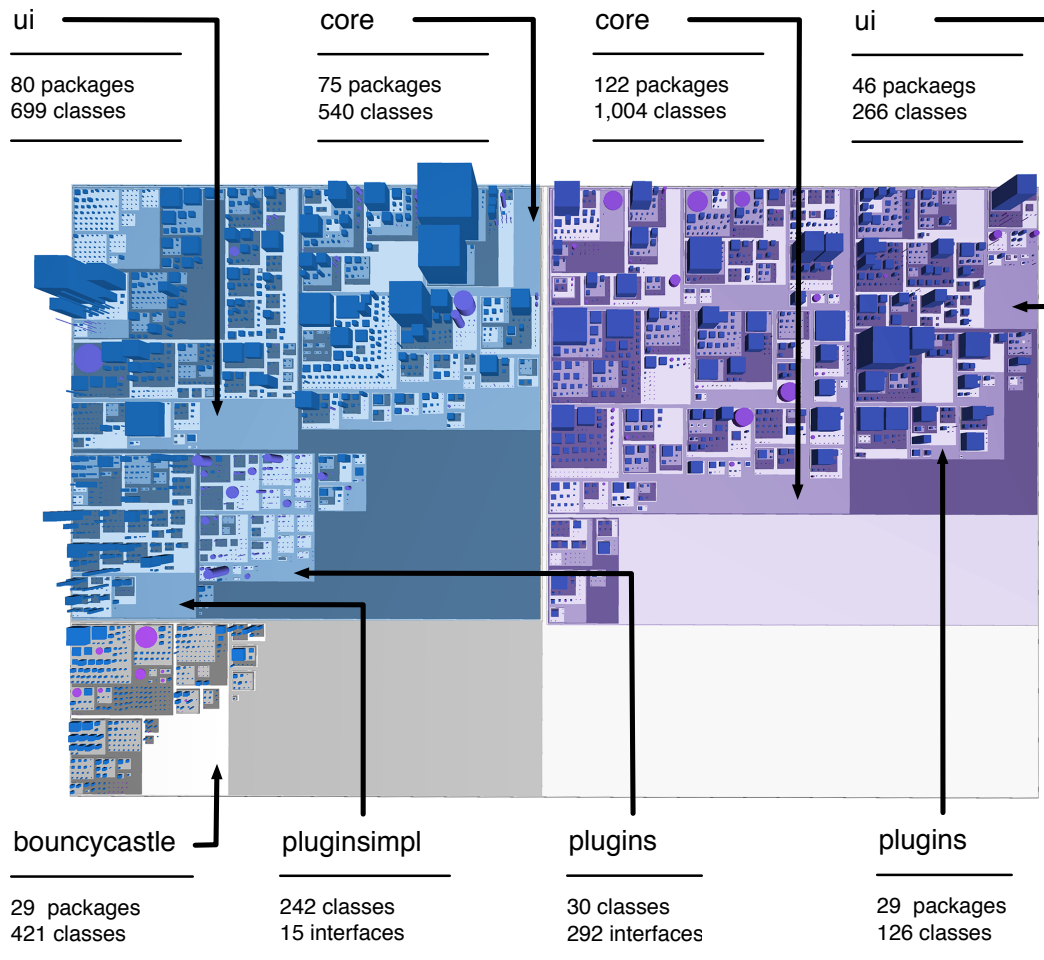


Figure 4.5. The city of Vuze

Vuze is a famous bit-torrent client written in Java. It was previously named *Azureus*, but it was renamed in 2006, after a large new release that enhanced the system with a new user interface and a content distribution platform that provides users with access to HD content organized in channels [Wik]. Since not all users are interested in the content distribution platform, after installing *Vuze* it is possible to disable the new components and run the “old” *Azureus*.

The city as a whole contains 511 districts and 3,800 buildings. The district with a blueish overlay is *azureus2* (*Azureus*), while the district with a purple overlay is *azureus3* (*Vuze*). *azureus2* contains 258 packages, 1,332 classes, and 534 interfaces, while *azureus3* contains 224 packages, 1,422 classes, and 91 interfaces.

The first difference that can be noticed is that the core of *Vuze* has 500 classes more than the core of *Azureus* and that the opposite is true for the *ui* districts. In *Azureus* plugins are divided in two districts, one defining the plugin interfaces and one containing the implementations of these interfaces.

Vuze does not have a package containing plugin interfaces. Both *Vuze* and the “old” *Azureus* allow to encrypt traffic and they use the *BouncyCastle* cryptography APIs to do so.

It is not clear whether the *azureus2* district is the ancestor of *azureus3* or if the latter contains the implementation of the *Vuze* content distribution platform and *azureus2* is still actively used in *Vuze* to transfer data. Although we have inspected various classes in both districts, we have not found information clarifying this question in either comments or Javadoc.

4.1.6 NetBeans

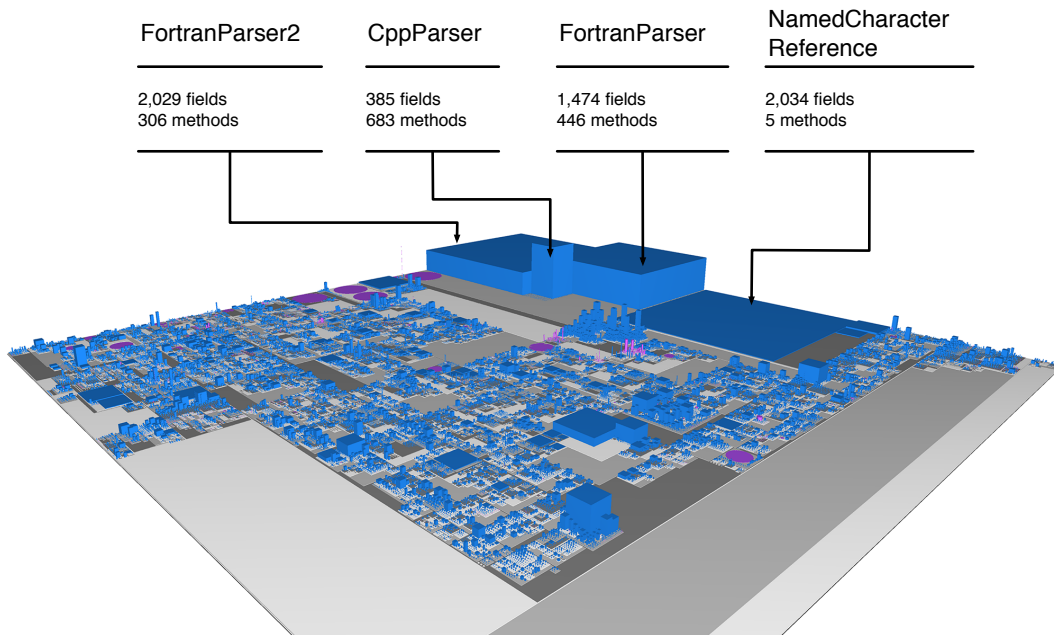


Figure 4.6. The metropolis of NetBeans

As a final stress test, we tried to visualize all the modules of *NetBeans*. We have cloned the whole *Mercurial* repository, which contains 817 modules. It took more than 47 minutes to build the project on our benchmark machine and it took almost 15 minutes to visualize the project.

The end result is a metropolis containing 4,844 districts and 42,514 buildings.

The three massive office buildings and the parking-lot in the back are a quite unexpected surprise and are an exception in a very large city where there are not many buildings particularly larger than the others. Also *NamedCharacterReference* is related to language parsing, as it belongs to the *html.parser* module.

With a system of this size, the visualization is unusable due to a considerable amount of lag.

Moreover, we have noticed depth-fighting⁴ issues when looking at the city from too far. We had to move closer to the city in order to get a screenshot without much depth-fighting.

⁴<http://en.wikipedia.org/wiki/Z-fighting>

4.1.7 Exploration Wrap Up

To test the scalability of our implementation, we have visualized a few systems of diverse size.

We have briefly explored every system and illustrated some of the findings we obtained.

Apart from *NetBeans*, *Manhattan* visualized all systems in less than 30 seconds.

When exploring *ActiveMQ* and *Vuze*, we have experienced lag in the tooltip descriptions: It took more than one second for the tooltips to appear. We ignore the reason why *OpenGL* picking takes so long with these systems and we have created a thread about this issue in the forum of the rendering framework we are using to verify whether we are using the provided picking functionality in the wrong way. We have also noticed that the plugin memory consumption was much higher than expected. Suspecting a memory leak, we have investigated the issue with the *Memory Analyzer* plugin⁵. We fixed the memory leak and reduced memory consumption by more than 50%.

Although *Manhattan* was not responsive enough to be usable when visualizing *NetBeans*, we still managed to see its city and we discovered four very large classes.

4.2 Evaluating our Approach with an Exploratory Study

Manhattan is focused on software visualization and collaboration support. A quantitative study on such a tool requires a large population of developers and a considerable amount of time, because the experiment participants have to get used to the visualization and they should be observed for a period of time long enough to provide us with meaningful data. Therefore we have opted for a qualitative exploratory study aimed at verifying whether we are on the right track with our approach by collecting feedback from the participants.

More precisely we want to understand (1) whether our visual notifications of changes are visible and intuitive, (2) if our conflict alerts are usable and provide relevant information, and (3) if the participants believe that a more mature version of our tool would help them in their everyday development sessions.

4.2.1 Study Description

Hattori has designed an experiment to verify the conflict detection capabilities of *Syde*. In this experiment she asks participants to perform a set of programming tasks on *Checkstyle*⁶, to fix a set of broken tests from the project's test suite. Participants are given three tasks each and whenever they complete a task, they have to commit their changes to a repository. Before moving to the next task the participants have to successfully incorporate each other's changes so that, at the end of the experiment, the whole test suite is passing for the working copy of both participants. Each pair of programming tasks is designed to lead to conflicts on the repository. Hattori verifies the effectiveness of her approach by allowing developers to use, for every pair of tasks, a different subset of *Syde*'s conflict-related features. Participants are also asked to fill two questionnaires, one before and one after the programming tasks. We asked Hattori the permission to include the described experimental setup in our experiment, to validate the strategies we employ to visualize team activity.

⁵<http://www.eclipse.org/mat/>

⁶<http://checkstyle.sourceforge.net>

Our experiment is structured as follows: At the beginning, participants are asked to fill-in a pre-experiment questionnaire. Then, after a brief tutorial on *Manhattan*, the participants have to perform a program comprehension task on *Checkstyle* using our tool. After that, we ask participants to perform, with the support of *Manhattan*, the programming tasks designed by Hattori. Finally, when the participants are done with their tasks, we ask them to fill a post-experiment questionnaire. Before discussing the results, we describe each phase more in detail.

Pre-Experiment Questionnaire

This questionnaire is made of two parts. The first one is focused on assessing participants' experience in software development. More precisely participants are asked about the size of the systems they work with, the size of their development teams, and their experience with SCMs. In the second part we ask participants about their knowledge in software visualization and their reverse engineering experience and habits. We also ask some questions to assess whether the features of *Manhattan* match what participants expect from tools to support collaborative development. The first part of the questionnaire is taken from the experiment designed by Hattori.

Program Comprehension Task

Participants are given ten minutes to explore *Checkstyle* using our tool and to report their findings about the design of this system. The main goal of this task is to give participants some time to get acquainted with the visualization, so that they can use it more efficiently during the programming tasks that they will be assigned later. Ten minutes is not a realistic amount of time for a program comprehension task and we are not specifically interested in the quality of the findings reported by the participants. Still we ask them a report to verify that they understand how to use the tool.

Programming Tasks

The assigned tasks are those designed by Hattori. While performing the first task, participants are not allowed to use *Manhattan* and they will have to merge conflicting changes. In the remaining tasks, participants are supported by our tool and we expect the number of merge operations to decrease thanks to the information we visualize. During all programming tasks, participants are allowed to communicate with each other via *Skype*⁷.

Post-Experiment Questionnaire

The first part of this questionnaire aims at verifying that the experiment was designed appropriately, meaning that it was not too complicated for the participants to understand, and that our guidance during the experiment was sufficient. After that, we ask a few questions to assess the usability of *Manhattan* during the program comprehension task. Then participants are asked to report their experience during the programming tasks, by answering a set of questions for every task. These questions aim at discovering if participants managed to avoid merge conflicts thanks to our conflict alerts and to what extent they communicated with each other to resolve emerging or committed conflicts. After that, we ask participants feedback about the way we visualize awareness information and about the tool in general. Finally, participants are asked to write the positive and negative aspects of *Manhattan* and the future improvements they are most interested in. The first part of the questionnaire and the questions on the programming tasks have been conceived by Hattori.

⁷<http://en.wikipedia.org/wiki/Skype>

4.2.2 Results

Given the feedback received by participants, both in the questionnaires and in the open-answer questions, we believe that we are on the right track. In fact we have received very positive feedback about the possibility to “see” the activity of the development team from all participants but one.

We find the feedback provided by two participants in particular very interesting; we refer to them as *a* and *b*. *a* really appreciates our unified view, while *b* likes our structural visualization, but is not interested in seeing the activity of the development team. We first discuss the comments from the participant that has given positive feedback on the unified view as a whole.

Participant *a* has almost eight years of experience in the industry, working on large industrial systems. When asked about the positive aspects of *Manhattan*, the participant wrote “*It is a very intuitive way to get familiar with a code base, and also for informing of conflicts before they become too painful to fix*”. We find this feedback a strong argument in favor of our approach, because it comes from a person with much experience, who is used to merge conflicts “*one third of the times*” before doing the check in of changes.

Since the results presented by Grinter show that developers commit even incomplete changes to avoid conflicts [Gri96], we were not expecting a feedback such as that given by *b*, because we visualize team activity in order to reduce merge conflicts. We informally asked the participant a more thorough explanation to the answers provided in the post-experiment questionnaire. From this further investigation, we understood that *b* is used to work in a small co-located team of classmates where the degree of communication is very high and each member of the development team is well aware of the activity of others. With such a high level of awareness, the participant can merge conflicting changes without much trouble, therefore merging conflicts once a programming task is complete, is preferred over solving emerging conflicts.

Our approach is focused on settings, such as GSD, where the level of awareness drops, because the explicit communication channels can not be used. Participant *b* is used to work in a setting opposite to GSD, therefore the given feedback is only partially a relevant argument against our approach: In the perspective of the participant, the information we provide is not relevant, as our approach is focused on a setting different from the one in which the participant is used to work.

The participants shared some critical remarks about the usability of the navigation system, which lacks mouse interactions such as rotating the camera by dragging the mouse or zooming by scrolling. We agree with most of these suggestions, which can be easily applied. In fact we have already implemented part of them after receiving the participants’ feedback.

We lack the support of a quantitative study of appropriate scale to draw any definitive conclusion proving that our approach is appropriate for all kinds of systems and all kinds of development teams. However, the questions in the exploratory study give us detailed information that can not be obtained with a quantitative study alone. Moreover, although they are not many, the participants we managed to gather come from different backgrounds and one of them has much experience working in the industry. For these reasons, we find the positive feedback given by the participants encouraging indeed: We are convinced to be on the right track and that our approach deserves further exploration and a later quantitative study.

Chapter 5

Conclusions

Being convinced that the visualization implemented in *CodeCity* should be deeply integrated in the development process, we implemented its essential features in an *Eclipse* plugin named *Manhattan*. By including the team activity information provided by *Syde*, we created a unified view on the structure of systems and on the activity of their development teams.

To assess the scalability of our implementation we have visualized on our laptop some well known Java systems and briefly explored their cities. These systems have been visualized in less than thirty seconds and for almost all of them, the visualization was responsive: When exploring the cities for *ActiveMQ* (2,500+ classes) and *Vuze* (3,800+ classes), we have experienced lag in the tooltip descriptions. We also tried to visualize the *NetBeans* IDE: It took more than fifteen minutes to visualize the system and the resulting visualization was not responsive enough to be usable, but we still managed to take a few pictures and discover four massive classes in the system. We have devised a strategy to improve performance, described in Section 5.1.

Finally, we have conducted a qualitative exploratory study, where we asked the participants to perform a set of tasks on a system and then collected their feedback about our unified view. The participants shared some remarks about the lack of mouse interactions in the navigation, such as rotating the camera by dragging the mouse. These interactions are simple to add and we have already implemented some of them. Regarding our unified view, we have received very positive feedback, especially from a participant with almost eight years of experience in the industry.

We have submitted a paper to the Eclipse-IT ¹ workshop: The paper has been accepted and, later this month, we will present *Manhattan* to various members of the italian *Eclipse* community. To conclude, we have brought to the *Eclipse* platform a software visualization tool to help developers to reason about the systems they are building and to support their collaborative effort; the feedback we received has convinced us that our approach is worth of being further explored and refined.

¹<http://2011.eclipse-it.org>

5.1 Future Work

5.1.1 Corner-Stitch Layout

The *rectangular-packing* layout algorithm is well suited for the city metaphor, as it improves scalability and helps to convey the metaphor, by producing compact and realistic layouts. Nevertheless, it is not able to cope with changes in the elements to layout. Although the *corner-stitch* layout algorithm² is the ideal choice for our requirements, implementing it was not a feasible solution given the time constraints we had. Therefore we have used the *rectangular-packing* algorithm and we implemented some workarounds to reduce the impact of code changes on the city layout.

We have tested these workarounds by performing realistic modifications to a few systems.

The results of these tests convinced us that the workarounds are not effective enough, and, that implementing the *corner-stitch* algorithm is the only solution to keep the visualization up-to-date with the contents of a developer's working copy, while preserving the layout of the city and the mental map learnt by the developer.

5.1.2 Improve the Visualization of Changes

While devising an approach to efficiently visualize the information provided by *Syde*, we have made the hypothesis that, since collaboration among developers has been supported by SCMs for decades, developers reason about the systems they develop from the point of view of their working copy. Based on this hypothesis, we have decided to visually notify a developer about the changes performed by his colleagues only when they modify, or delete, the code artifacts that are also in his working copy, and to show conflict alerts only for the conflicts in which the developer is involved. In order to maintain the visualization as clean as possible, we have designed visible and clear notifications of team activity based on color, shape, and lighting.

Nevertheless, as described in Section 3.3.3, this strategy has one main drawback: It does not notify developers about consecutive modifications to a class.

The first step to solve this issue is to use textures: Besides painting the building for a modified artifact in yellow³, a texture (e.g.: vertical stripes) should be applied on the building. If the artifact is modified again, a very different texture (e.g.: horizontal stripes) should be applied, and for later modifications, the two textures should be alternated. In this way developers will notice a change in the texture of a building and deduce that the class represented by the building has being changed once more. However, texture changes are hard to notice for small buildings, therefore they should be accompanied by minimalistic animations that make such buildings more visible. We claim that the animation that best fits the requirements is bouncing. To limit the amount of visual noise, only buildings smaller than a certain threshold should be animated. Moreover, the algorithms computing the threshold and the height of the bounce should be functions of the bounding box of the city, so that buildings bounce just high enough to be noticed. Although we find this strategy promising, there is still an open issue: Depending on the city and on the activity of the team, too many buildings might start bouncing, thus annoying users with visual noise.

²The complete publication is available at: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1982/6352.html>

³Or in orange in case that the artifact is deleted

5.1.3 Improve Performance

We have noticed performance issues when visualizing systems with more than 2,500 classes. We know that the main reason for this performance overhead is due to how the rendering framework we have used integrates with *SWT*. Moreover, the threading model offered by this framework is not sophisticated enough. For these reasons, we have decided to switch to a more promising and better documented framework, which at the moment does not support *SWT*. We plan to ask guidance to the framework's core developers to include support for the toolkit and to later update *Manhattan* to use this framework.

5.1.4 Support other Programming Languages

At the moment we are able to visualize only Java systems. However, *Eclipse* provides support for many other programming languages through language integration plugins. We plan to leverage these plugins in order to implement code-model extractors for *C++* and *Python*.

Appendix A

A

In this appendix we provide the complete handouts of the textual instructions we have given every pair of developers that participated in our exploratory study.

<p>Manhattan Evaluation Experiment T2</p> <p>Participant: <input type="text"/></p> <p>Introduction</p> <p>Understanding software systems is difficult, because software is large and complex. Moreover, like all conceptual artifacts, software is intangible, thus even more difficult to understand. Software visualization techniques have been investigated since the end of the 80s to produce visual representations of systems in order to ease reasoning about various aspects of software (e.g. architecture, evolution, performance, ...).</p> <p>Software development is a collaborative process, where members of a development team concurrently modify the system. In order to share and coordinate changes to the system, developers employ software configuration management systems (SCM), such as Subversion or Git. However, a developer only knows what his colleague has changed after he checks in the code. As a consequence, when people change the same parts of the code, they have to deal with merging and resolving conflicts. If developers were aware of the activity of their colleagues (i.e. which code artifacts they are modifying), they could discuss with each other to find a way to complete their tasks without running into a conflict.</p> <p>In the context of my master thesis I have developed Manhattan, a software visualization tool in the form of an Eclipse plugin. The visualization provided by this tool is focused on easing program comprehension (understanding architecture) and supporting collaborative development by making developers aware of the activity of their colleagues.</p> <p>The goal of this experiment is to assess the effectiveness of the proposed visualization.</p> <p>The system you will work with in this experiment is Checkstyle</p> <p><i>"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard."</i> Checkstyle documentation</p> <p>In short, Checkstyle takes as input a Java source file and an XML configuration file that specifies the coding standards that must be enforced (i.e., the checks that are to be used). Most people are not familiar with Checkstyle's implementation. However, IDEs (such as Eclipse) may be able to assist in understanding Checkstyle's inner workings, and most of the source code is fairly well documented.</p>	<p>Manhattan Evaluation Experiment T2</p> <p>You and the other participant will be assigned 4 tasks to perform on the project.</p> <p>We kindly ask you to:</p> <ul style="list-style-type: none"> perform the tasks in the specified order; write down the current time before starting to work on a new task and once after completing all tasks; not return to earlier tasks because it affects the experiment; notify the experimenter before starting a task, and wait for his authorization. <p>The experiment begins with a questionnaire and ends with another questionnaire and a debriefing talk.</p> <p>Thank you for participating in this experiment!</p> <p>Francesco Rigotti, Michele Lanza, Alberto Bacchelli and Lile Hattori</p>
	<p>Pre-experiment Questionnaire</p>

Figure A.1. Participant1, pages 1-4

Pre-experiment Questionnaire

General Survey

In this short survey a number of questions regarding your experience with software development and use of software configuration management systems will be asked to get an impression of your skills and expectations.

1 The first questions are about you. Please answer the following questions about your personal background. Your answers will be kept private and will only serve to put your other answers in context.

What is your age?

Education background (e.g., computer science, electrical engineering)

Current job/education position(s) (e.g., developer, project manager, master student)

Current affiliation(s) (company and/or University)

2 Below a few statements regarding your software development experience are shown. Please indicate: 1) the number of years of experience, and 2) rate each statement about your personal experience according to the following scale:

- 0 - None (you don't know this subject);
- 1 - Beginner (you are familiar with this subject but still have some difficulties to use it);
- 2 - Knowledgeable (you are comfortable in this subject);
- 3 - Advanced (you know the subject well and use it on a daily basis);
- 4 - Expert (you consider yourself highly proficient in this subject).

	#years of experience	0	1	2	3	4
Java development		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Development in a team		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Developing industrial size systems		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using IDEs (Eclipse, VisualStudio, NetBeans, etc)		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using Eclipse for Java development		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using SCM (e.g., CVS, SVN, Git)		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Familiarity with Checkstyle		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testing with JUnit		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Pre-experiment Questionnaire

3 Some questions regarding the use of software configuration management (SCM) systems are shown below. Please answer them according to your experience.

What SCM system(s) do you currently use?

Do you usually work in teams?

If yes, what is the size of your team?

With what frequency do you check out a project (or part of it) from the repository?

With what frequency do you check in a project (or part of it)?

With what frequency do you have to resolve conflicts during merging?

4 Some questions regarding the use of software visualization tools are shown below. Please answer them according to your experience.

Have you ever heard about software visualization before participating in the experiment?

Name any software visualization tools you used in the past

Have you used software visualization tools to understand a system's architecture before?

Pre-experiment Questionnaire

5 Some questions about your experience and habits in reverse engineering systems to understand how they work and how to use them. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5 - strongly agree.

	yes / no
Have you ever being assigned the task to reverse engineer a system to maintain it?	<input type="radio"/>

Rank how often you use these techniques to understand a system and how to use it

	1	2	3	4	5
Read documentation before reading the code	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Read code and comments	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Look at architectural descriptions and UML diagrams	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Run the system using a debugger	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Run the system after placing logging instructions	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

6 Some questions regarding your expectations from a tool to support collaborative development. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5 - strongly agree.

	1	2	3	4	5
The tool should make me aware of what the other developers are doing (e.g. which code artifacts they are modifying)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should warn me if my changes are leading to a conflict that will show up in the SCM repository	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Information about what artifacts are being modified by others and about conflicting changes should be very fine-grained	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should inform me about the resolution of a conflict	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should notify me when the other developers interact with the repository	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure A.2. Participant1, pages 5-8

Tutorial on Manhattan

Manhattan visualizes software systems as 3D cities. The metaphor used to create these cities is described below.

Buildings represent classes and interfaces, while districts represent packages. The height of a building depends on the number of methods inside the class it represents. The width and length of a building depend on the number of fields. Blue buildings are classes, while purple buildings are interfaces.

This metaphor produces some "special" kinds of buildings that represent classes important to the system. The figure below shows the city for ArguJML, annotated with a description of these "special" buildings.

Before starting the experiment, I will show a brief tutorial on how to use Manhattan.

Topic	Keys	Notes
Navigation		
Switch navigation mode	O	The navigation mode is indicated on the left in the status bar
Controls for the first-person navigation mode	Move using WASD Turn using arrow keys	
Controls for the orbital navigation mode	Move on the orbit using WASD	When entering this mode you will be moved to an observation point on the orbit
Going back to the default observation point	0 (zero)	
Tooltip descriptions		
See the description for a city element	Hover on it with the mouse for a little time	
Open the class/interface represented by a building	Right-click on the building	
Tooltip descriptions for changes and conflicts		
See the description for a change	Hover the mouse on the city element highlighted by the change	
See the description for a conflict	Hover the mouse on the conflict's sphere	
See the two versions of a class with a conflict on it	Right-click on the conflict's sphere	

Figure A.3. Participant1, pages 9-12

<div data-bbox="344 551 767 645" style="text-align: center;"> <h2>Collaboration Support</h2> </div>	<div data-bbox="839 349 1251 394" style="background-color: #cccccc; padding: 5px;"> <h3>Overview of Checkstyle</h3> </div> <p>Typical execution stages:</p> <p>A typical execution of Checkstyle takes as inputs a set of Java source files and an XML configuration file that specifies the coding standards that must be enforced. The execution itself can be divided into 4 main stages:</p> <ol style="list-style-type: none"> 1. Initialization. It sets the environment by parsing the command, and reading the configuration. 2. Source parsing. Reads and parses the source input files. It constructs an abstract syntax tree (AST) for each source file. 3. Checking. Checks each input file. 4. Error reporting. It outputs the report of the checks. The output can be in plain text, as an XML file, or other formats. <p>These execution stages can be easily identified in the class <code>com.puppycrawl.tools.checkstyle.Main</code>.</p> <p>Architectural view:</p> <p>Checkstyle is divided into 7 main packages:</p> <ol style="list-style-type: none"> 1. <code>com.puppycrawl.tools.checkstyle</code> - the main package containing the Main, Checker, DefaultConfiguration and logging/auditing classes 2. <code>com.puppycrawl.tools.checkstyle.api</code> - the core API to be used to implement a check 3. <code>com.puppycrawl.tools.checkstyle.checks</code> - the checks that are bundled with the main distribution 4. <code>com.puppycrawl.tools.checkstyle.doclets</code> 5. <code>com.puppycrawl.tools.checkstyle.filters</code> 6. <code>com.puppycrawl.tools.checkstyle.gnomers</code> 7. <code>com.puppycrawl.tools.checkstyle.gui</code> <p>The tasks of this assignment are concentrated in the first three packages.</p>
<div data-bbox="344 1055 756 1099" style="background-color: #cccccc; padding: 5px;"> <h3>Instructions to perform the assignment</h3> </div> <p>There are a few broken tests at the moment and you are responsible to fix half of them. Your and your pair's ultimate goal is to have fixed all the tests by the end of the assignment.</p> <p>Each task contains the name of the test you need to make pass and the class you need to change in order to fix the test. You are not allowed to change or commit the tests.</p> <p>Running the tests:</p> <p>Your Eclipse setup contains a project with Checkstyle's source (and links to its external libraries).</p> <ul style="list-style-type: none"> • To run the tests, click on the arrow besides the run button and choose the pre-configured JUnit called 'checkstyle'. Alternatively, right-click on 'src/test', and choose 'Run As' -> 'Run Configurations' ... and then select 'JUnit/checkstyle'. <p>8</p> <p>Communicating with the other participant:</p> <p>You can only consider a task done when your pair also finished his task and both succeeded in fixing the tests. In addition, the code changes you and the other participant are going to perform will most likely conflict with one another. Skype is at your disposal, and you can use it at any time to communicate with your pair to better coordinate your tasks.</p> <p>Coordinating begin/end of programming tasks:</p> <p>At the beginning of each task, read the description and, when you are ready to start changing the code, notify the experimenter. You should wait for the experimenter's authorization to start coding.</p> <p>When you finish a task, check with the other participant whether (s)he also finished. When both of you have finished, you can go to the next task.</p> <p>Troubleshooting:</p> <ol style="list-style-type: none"> 1. My test is failing because it cannot find the input file (File not found). This can happen when: i) you run a single test - try running the complete test suite; ii) you do not select the project in the package explorer before running the tests - try selecting the project first. 2. I tried running the tests and got the following error: 'Launching checkstyle' has encountered a problem. Variable reference empty selection: \$[project_loc]. Before running the tests, select the project in the package explorer. <p>Should you have trouble while performing the task, please consult us.</p>	<div data-bbox="839 1055 1251 1099" style="background-color: #cccccc; padding: 5px;"> <h3>Current Time</h3> </div> <div data-bbox="884 1305 1206 1435" style="border: 1px solid black; padding: 10px; text-align: center;"> <p>hours : minutes</p> </div>

Figure A.5. Participant1, pages 17-20

<h3>Warm up!</h3> <p>Let's start to get used to the Checks. The goal of this warm up is to fix the test <code>EqualsAvoidNullTest</code>.</p> <p>Class <code>EqualsAvoidNullCheck</code> checks that any combination of String literals with optional assignment is on the left side of an equality comparison. Here is an example:</p> <pre>String person = "myself"; if (person.equals("you")) { ... }</pre> <p>In this case, the string literal is in the right side, which can potentially cause a <code>NullPointerException</code> if <code>person</code> is null. Hence, the check logs a warning indicating that its expression should be reversed.</p> <p>The same rule applies for <code>equalsIgnoreCase()</code>, however it is not being checked.</p> <p>Modify method <code>visitToken(Final DetailAST aMethodCall)</code> to add the check for method <code>equalsIgnoreCase()</code>.</p> <p>Note: you do not need to coordinate with the other participant for this warm up task.</p> <p>Test to pass: <code>com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullTest</code></p> <p>Class to modify: <code>com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullCheck</code></p>	<h3>Current Time</h3> <div style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; width: 100px; margin: 20px auto;"><p>__ : __</p><p>hours minutes</p></div>
	<h3>Tasks</h3>

Figure A.6. Participant1, pages 21-24

	<div data-bbox="837 349 1262 394" style="background-color: #cccccc; padding: 5px;">Preparing for Task 1</div> <div data-bbox="837 405 1262 495" style="border: 1px solid #ccc; padding: 5px;"> <p>For Task 1, you are not allowed to use the visualization, please minimize it (do not close it)</p> </div>
<div data-bbox="344 1055 756 1099" style="background-color: #cccccc; padding: 5px; text-align: center;">Current Time</div> <div data-bbox="389 1305 713 1435" style="border: 1px solid #ccc; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <p>— — . — — hours minutes</p> </div>	<div data-bbox="837 1055 1262 1099" style="background-color: #cccccc; padding: 5px;"> Improving MethodCountCheck Task 1 </div> <div data-bbox="837 1115 1262 1155" style="border: 1px solid #ccc; padding: 5px;"> <p>The goal of this task is to fix the test <code>MethodCountCheckTest</code> by changing the code of class <code>MethodCountCheck</code>. It is divided into 2 parts. After you have finished them, all tests from <code>MethodCountCheckTest</code> should be passing.</p> </div> <div data-bbox="837 1171 1262 1234" style="border: 1px solid #ccc; padding: 5px;"> <p>1. Fix testThrees: This test is failing because <code>checkCounters(MethodCounter cCounter, DetailLAST oAst)</code> in <code>MethodCountCheck</code> is not verifying the number of package methods (those with default visibility). Implement this verification and test again.</p> </div> <div data-bbox="837 1249 1262 1312" style="border: 1px solid #ccc; padding: 5px;"> <p>2. Fix testEnum: This test is failing because <code>checkCounters(MethodCounter cCounter, DetailLAST oAst)</code> in <code>MethodCountCheck</code> is not verifying the number of private methods. Implement this verification and test again.</p> </div> <div data-bbox="837 1328 1262 1346" style="border: 1px solid #ccc; padding: 5px;"> <p>To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="837 1608 1262 1671" style="background-color: #cccccc; padding: 5px; border: 1px solid #ccc;"> <p>Test to pass: <code>com.puppcrawl.tools.checkstyle.checks.sizes.MethodCountCheckTest</code> Class to modify and check in: <code>com.puppcrawl.tools.checkstyle.checks.sizes.MethodCountCheck</code></p> </div>

Figure A.7. Participant1, pages 25-28

<div data-bbox="344 349 756 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 600 711 725" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 30px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 30px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 349 1251 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Preparing for Task 2</div> <div data-bbox="839 412 1251 519" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 200px;"> <p>For Task 2, you are allowed to use the visualization and the notifications about the activity of your colleague.</p> <p>The experiment manager will tell you how to enable the notifications. After this, notifications and alerts will appear in the visualization.</p> </div>
<div data-bbox="344 1052 756 1097" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 1303 711 1429" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 30px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 30px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 1052 1251 1097" style="background-color: #cccccc; text-align: center; padding: 5px;">Finishing PlainTextLogger Task 2</div> <div data-bbox="839 1115 1251 1585" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 200px;"> <p>The goal of this task is to fix the test <code>PlainTextLoggerTest</code> by changing class <code>PlainTextLogger</code>.</p> <p><code>PlainTextLogger</code> is a class to output the violations as plain text, similarly to <code>DefaultLogger</code> but, with customized log message.</p> <p>In the following, we show an example of an output of a check formatted in plain text and default text.</p> <p>Plain text:</p> <pre>Starting audit... Starting file=Test.java Test.java line=1 column=1 severity=warning message=key Finished file=Test.java Audit done.</pre> <p>Default text:</p> <pre>Starting audit... Test.java:1:1: warning: key Audit done.</pre> <p>Currently, there are two broken tests: <code>testAddError</code> and <code>testFileStarted</code>. Fix them in the following order:</p> <ol style="list-style-type: none"> 1. Fix <code>testAddError</code>: This test is failing because method <code>addError(AuditEvent evt)</code> in <code>PlainTextLogger</code> is not checking whether the severity level is 'ERROR'. Implement this check and make sure the test pass before you go to the next fix. 2. Fix <code>testFileStarted</code>: This test is failing because method <code>fileStarted(AuditEvent evt)</code> in <code>PlainTextLogger</code> is currently empty. Implement this method and rerun the tests. <p>To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="839 1603 1251 1666" style="background-color: #cccccc; padding: 5px; margin-top: 10px;"> <p>Test to pass: <code>com.puppcrawl.tools.checkstyle.PlainTextLoggerTest</code> Class to modify and check in: <code>com.puppcrawl.tools.checkstyle.PlainTextLogger</code></p> </div>

Figure A.8. Participant1, pages 29-32

<div data-bbox="344 344 756 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 595 711 725" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 344 1262 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Preparing for Task 3</div> <div data-bbox="839 405 1262 515" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 250px;"> <p>No further preparation steps are required.</p> </div>
<div data-bbox="344 1055 756 1104" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 1301 711 1431" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 1055 1262 1104" style="background-color: #cccccc; text-align: center; padding: 5px;">Finishing JsonLogger Task 3</div> <div data-bbox="839 1115 1262 1592" style="border: 1px solid black; padding: 10px;"> <p>The goal of this task is to fix the test <code>JsonLoggerTest</code>.</p> <p><code>JsonLogger</code> is a class to output the violations in JSON (Javascript Object Notation) format. JSON is a data-interchange format that is easy to read/write and parse/generate. JSON is built in two structures:</p> <ul style="list-style-type: none"> • A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. • An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. <p>We show an example of an output of a check formatted in JSON and XML (See <code>XMLLogger</code>).</p> <pre> {"checkstyle version":"5.3" {"file":{ "file":{ "error":{ "line":"9" "column":"40" "severity":"error" "message":"Parameter text should be final." "source":"Test" }} }} } <?xml version="1.0" encoding="UTF-8"?> <checkstyle version="5.3"> <file name="Test.java"> <error line="9" column="40" severity="error" message="Parameter text should be final." source="Test"/> </file> </checkstyle> </pre> <p>In this task, you should fix two tests in the following order:</p> <ol style="list-style-type: none"> 1. <code>testAddErrorMessage</code>: This test is failing because there is an error in method <code>addError(AuditEvent eEvt)</code> in <code>JsonLogger</code>. It should only print the message when <code>eEvt.getMessage()</code> is not null nor empty, but it's printing it every time. Fix it to make the test pass. 1. <code>testFileFinished</code>: This test is failing because the method <code>fileFinished(AuditEvent eEvt)</code> in <code>JsonLogger</code> is empty. Implement it and make the test pass. <p>To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="839 1603 1262 1671" style="background-color: #cccccc; padding: 5px;"> <p>Test to pass: <code>com.puppycrawl.tools.checkstyle.JsonLoggerTest</code> Class to modify and check in: <code>com.puppycrawl.tools.checkstyle.JsonLogger</code></p> </div>

Figure A.9. Participant1, pages 33-36

<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Current Time</div> <div style="text-align: center; border: 1px solid black; border-radius: 10px; padding: 20px; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> ____ hours </div> <div style="text-align: center;"> : </div> <div style="text-align: center;"> ____ minutes </div> </div> </div>	<div style="text-align: center; background-color: #cccccc; padding: 20px; margin: 20px auto; width: 250px;">Post-experiment Questionnaire</div>																																																																																																						
<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Post-experiment Questionnaire</div> <div style="background-color: #cccccc; padding: 5px; margin-bottom: 10px;">Experiment evaluation</div> <p>Thanks for completing the tasks! To get an impression of your experience with the experiment and to allow you to give your comments, please fill in the questions below.</p> <p>1 This question is about your overall experience in performing the experiment. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> </tr> </thead> <tbody> <tr> <td>Overall, the tasks were feasible</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I felt time pressure</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I would have needed more guidance to complete the tasks</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The warm up phase was useful</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The tasks were interesting to do</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The tasks were realistic</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The experiment was fun to do</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> </tbody> </table>		1	2	3	4	5	Overall, the tasks were feasible	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I felt time pressure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I would have needed more guidance to complete the tasks	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The warm up phase was useful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The tasks were interesting to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The tasks were realistic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The experiment was fun to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Post-experiment Questionnaire</div> <p>2 These statements relate to the usability of the visualization and its effectiveness to support program comprehension. Please rate the following statements on a scale from 1 to 5. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> </tr> </thead> <tbody> <tr> <td>The visualization is effective in supporting program comprehension</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The ability to access code from the visualization is very important</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The navigation system is easy to use</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>Tooltips are usable and provide useful information</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>Overall the visualization is usable and intuitive</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The visualization is not usable on a laptop screen, as it is too small</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The visualization highlights important system components</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I would use the visualization in everyday coding</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> </tbody> </table>		1	2	3	4	5	The visualization is effective in supporting program comprehension	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The ability to access code from the visualization is very important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The navigation system is easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Tooltips are usable and provide useful information	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Overall the visualization is usable and intuitive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The visualization is not usable on a laptop screen, as it is too small	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The visualization highlights important system components	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I would use the visualization in everyday coding	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	1	2	3	4	5																																																																																																		
Overall, the tasks were feasible	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I felt time pressure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I would have needed more guidance to complete the tasks	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The warm up phase was useful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The tasks were interesting to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The tasks were realistic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The experiment was fun to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
	1	2	3	4	5																																																																																																		
The visualization is effective in supporting program comprehension	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The ability to access code from the visualization is very important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The navigation system is easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
Tooltips are usable and provide useful information	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
Overall the visualization is usable and intuitive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The visualization is not usable on a laptop screen, as it is too small	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The visualization highlights important system components	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I would use the visualization in everyday coding	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		

Figure A.10. Participant1, pages 37-40

<p>Manhattan Evaluation Experiment T2</p> <p>Participant: <input type="text"/></p> <p>Introduction</p> <p>Understanding software systems is difficult, because software is large and complex. Moreover, like all conceptual artifacts, software is intangible, thus even more difficult to understand. Software visualization techniques have been investigated since the end of the 80s to produce visual representations of systems in order to ease reasoning about various aspects of software (e.g. architecture, evolution, performance, ...).</p> <p>Software development is a collaborative process, where members of a development team concurrently modify the system. In order to share and coordinate changes to the system, developers employ software configuration management systems (SCM), such as Subversion or Git. However, a developer only knows what his colleague has changed after he checks in the code. As a consequence, when people change the same parts of the code, they have to deal with merging and resolving conflicts. If developers were aware of the activity of their colleagues (i.e. which code artifacts they are modifying), they could discuss with each other to find a way to complete their tasks without running into a conflict.</p> <p>In the context of my master thesis I have developed Manhattan, a software visualization tool in the form of an Eclipse plugin. The visualization provided by this tool is focused on easing program comprehension (understanding architecture) and supporting collaborative development by making developers aware of the activity of their colleagues.</p> <p>The goal of this experiment is to assess the effectiveness of the proposed visualization.</p> <p>The system you will work with in this experiment is Checkstyle</p> <p><i>"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard."</i> Checkstyle documentation</p> <p>In short, Checkstyle takes as input a Java source file and an XML configuration file that specifies the coding standards that must be enforced (i.e., the checks that are to be used). Most people are not familiar with Checkstyle's implementation. However, IDEs (such as Eclipse) may be able to assist in understanding Checkstyle's inner workings, and most of the source code is fairly well documented.</p>	<p>Manhattan Evaluation Experiment T2</p> <p>You and the other participant will be assigned 4 tasks to perform on the project.</p> <p>We kindly ask you to:</p> <ul style="list-style-type: none"> perform the tasks in the specified order; write down the current time before starting to work on a new task and once after completing all tasks; not return to earlier tasks because it affects the experiment; notify the experimenter before starting a task, and wait for his authorization. <p>The experiment begins with a questionnaire and ends with another questionnaire and a debriefing talk.</p> <p>Thank you for participating in this experiment!</p> <p>Francesco Rigotti, Michele Lanza, Alberto Bacchelli and Lile Hattori</p>
	<p>Pre-experiment Questionnaire</p>

Figure A.13. Participant2, pages 1-4

Pre-experiment Questionnaire

General Survey

In this short survey a number of questions regarding your experience with software development and use of software configuration management systems will be asked to get an impression of your skills and expectations.

1 The first questions are about you. Please answer the following questions about your personal background. Your answers will be kept private and will only serve to put your other answers in context.

What is your age?

Education background (e.g., computer science, electrical engineering)

Current job/education position(s) (e.g., developer, project manager, master student)

Current affiliation(s) (company and/or University)

2 Below a few statements regarding your software development experience are shown. Please indicate: 1) the number of years of experience, and 2) rate each statement about your personal experience according to the following scale:

- 0 - None (you don't know this subject);
- 1 - Beginner (you are familiar with this subject but still have some difficulties to use it);
- 2 - Knowledgeable (you are comfortable in this subject);
- 3 - Advanced (you know the subject well and use it on a daily basis);
- 4 - Expert (you consider yourself highly proficient in this subject).

	#years of experience	0	1	2	3	4
Java development		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Development in a team		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Developing industrial size systems		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using IDEs (Eclipse, VisualStudio, NetBeans, etc)		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using Eclipse for Java development		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Using SCM (e.g., CVS, SVN, Git)		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Familiarity with Checkstyle		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testing with JUnit		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Pre-experiment Questionnaire

3 Some questions regarding the use of software configuration management (SCM) systems are shown below. Please answer them according to your experience.

What SCM system(s) do you currently use?

Do you usually work in teams?

If yes, what is the size of your team?

With what frequency do you check out a project (or part of it) from the repository?

With what frequency do you check in a project (or part of it)?

With what frequency do you have to resolve conflicts during merging?

4 Some questions regarding the use of software visualization tools are shown below. Please answer them according to your experience.

Have you ever heard about software visualization before participating in the experiment?

Name any software visualization tools you used in the past

Have you used software visualization tools to understand a system's architecture before?

Pre-experiment Questionnaire

5 Some questions about your experience and habits in reverse engineering systems to understand how they work and how to use them. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5 - strongly agree.

	yes	no
Have you ever being assigned the task to reverse engineer a system to maintain it?	<input type="radio"/>	<input type="radio"/>

Rank how often you use these techniques to understand a system and how to use it

	1	2	3	4	5
Read documentation before reading the code	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Read code and comments	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Look at architectural descriptions and UML diagrams	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Run the system using a debugger	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Run the system after placing logging instructions	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

6 Some questions regarding your expectations from a tool to support collaborative development. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5 - strongly agree.

	1	2	3	4	5
The tool should make me aware of what the other developers are doing (e.g. which code artifacts they are modifying)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should warn me if my changes are leading to a conflict that will show up in the SCM repository	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Information about what artifacts are being modified by others and about conflicting changes should be very fine-grained	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should inform me about the resolution of a conflict	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The tool should notify me when the other developers interact with the repository	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

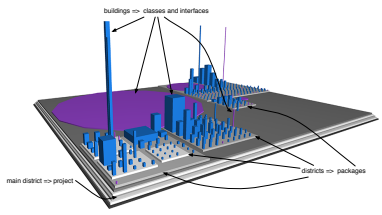
Figure A.14. Participant2, pages 5-8

Tutorial on Manhattan

Manhattan visualizes software systems as 3D cities. The metaphor used to create these cities is described below.

Buildings represent classes and interfaces, while districts represent packages. The height of a building depends on the number of methods inside the class it represents. The width and length of a building depend on the number of fields. Blue buildings are classes, while purple buildings are interfaces.

This metaphor produces some "special" kinds of buildings that represent classes important to the system. The figure below shows the city for ArguJML, annotated with a description of these "special" buildings.



buildings => classes and interfaces
main district => project
districts => packages

Before starting the experiment, I will show a brief tutorial on how to use Manhattan.

Topic	Keys	Notes
Navigation		
Switch navigation mode	O	The navigation mode is indicated on the left in the status bar
Controls for the first-person navigation mode	Move using WASD Turn using arrow keys	
Controls for the orbital navigation mode	Move on the orbit using WASD	When entering this mode you will be moved to an observation point on the orbit
Going back to the default observation point	0 (zero)	
Tooltip descriptions		
See the description for a city element	Hover on it with the mouse for a little time	
Open the class/interface represented by a building	Right-click on the building	
Tooltip descriptions for changes and conflicts		
See the description for a change	Hover the mouse on the city element highlighted by the change	
See the description for a conflict	Hover the mouse on the conflict's sphere	
See the two versions of a class with a conflict on it	Right-click on the conflict's sphere	

Figure A.15. Participant2, pages 9-12

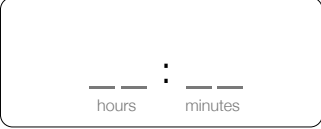
<p style="text-align: center;">Collaboration Support</p>	<h3 style="background-color: #cccccc; padding: 5px;">Overview of Checkstyle</h3> <p>Typical execution stages:</p> <p>A typical execution of Checkstyle takes as inputs a set of Java source files and an XML configuration file that specifies the coding standards that must be enforced. The execution itself can be divided into 4 main stages:</p> <ol style="list-style-type: none"> 1. Initialization. It sets the environment by parsing the command, and reading the configuration. 2. Source parsing. Reads and parses the source input files. It constructs an abstract syntax tree (AST) for each source file. 3. Checking. Checks each input file. 4. Error reporting. It outputs the report of the checks. The output can be in plain text, as an XML file, or other formats. <p>These execution stages can be easily identified in the class <code>com.puppycrawl.tools.checkstyle.Main</code>.</p> <p>Architectural view:</p> <p>Checkstyle is divided into 7 main packages:</p> <ol style="list-style-type: none"> 1. <code>com.puppycrawl.tools.checkstyle</code> - the main package containing the Main, Checker, DefaultConfiguration and logging/auditing classes 2. <code>com.puppycrawl.tools.checkstyle.api</code> - the core API to be used to implement a check 3. <code>com.puppycrawl.tools.checkstyle.checks</code> - the checks that are bundled with the main distribution 4. <code>com.puppycrawl.tools.checkstyle.doclets</code> 5. <code>com.puppycrawl.tools.checkstyle.filters</code> 6. <code>com.puppycrawl.tools.checkstyle.gnomers</code> 7. <code>com.puppycrawl.tools.checkstyle.gui</code> <p>The tasks of this assignment are concentrated in the first three packages.</p>
<h3 style="background-color: #cccccc; padding: 5px;">Instructions to perform the assignment</h3> <p>There are a few broken tests at the moment and you are responsible to fix half of them. Your and your pair's ultimate goal is to have fixed all the tests by the end of the assignment.</p> <p>Each task contains the name of the test you need to make pass and the class you need to change in order to fix the test. You are not allowed to change or commit the tests.</p> <p>Running the tests:</p> <p>Your Eclipse setup contains a project with Checkstyle's source (and links to its external libraries).</p> <ul style="list-style-type: none"> • To run the tests, click on the arrow besides the run button and choose the pre-configured JUnit called 'checkstyle'. Alternatively, right-click on 'src/test', and choose 'Run As' -> 'Run Configurations' ... and then select 'JUnit/checkstyle'. <p>8</p> <p>Communicating with the other participant:</p> <p>You can only consider a task done when your pair also finished his task and both succeeded in fixing the tests. In addition, the code changes you and the other participant are going to perform will most likely conflict with one another. Skype is at your disposal, and you can use it at any time to communicate with your pair to better coordinate your tasks.</p> <p>Coordinating begin/end of programming tasks:</p> <p>At the beginning of each task, read the description and, when you are ready to start changing the code, notify the experimenter. You should wait for the experimenter's authorization to start coding.</p> <p>When you finish a task, check with the other participant whether (s)he also finished. When both of you have finished, you can go to the next task.</p> <p>Troubleshooting:</p> <ol style="list-style-type: none"> 1. My test is failing because it cannot find the input file (File not found). This can happen when: i) you run a single test - try running the complete test suite; ii) you do not select the project in the package explorer before running the tests - try selecting the project first. 2. I tried running the tests and got the following error: 'Launching checkstyle' has encountered a problem. Variable reference empty selection: \$[project_loc]. Before running the tests, select the project in the package explorer. <p>Should you have trouble while performing the task, please consult us.</p>	<h3 style="background-color: #cccccc; padding: 5px;">Current Time</h3> <div style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; width: fit-content; margin: 20px auto;">  </div>

Figure A.17. Participant2, pages 17-20

<h3>Warm up!</h3> <p>Let's start to get used to the Checks. The goal of this warm up is to fix the test <code>EqualsAvoidNullTest</code>.</p> <p>Class <code>EqualsAvoidNullCheck</code> checks that any combination of String literals with optional assignment is on the left side of an equality comparison. Here is an example:</p> <pre>String person = "myself"; if (person.equals("you")) { ... }</pre> <p>In this case, the string literal is in the right side, which can potentially cause a <code>NullPointerException</code> if <code>person</code> is null. Hence, the check logs a warning indicating that its expression should be reversed.</p> <p>The same rule applies for <code>equalsIgnoreCase()</code>, however it is not being checked.</p> <p>Modify method <code>visitToken(Final DetailAST aMethodCall)</code> to add the check for method <code>equalsIgnoreCase()</code>.</p> <p>Note: you do not need to coordinate with the other participant for this warm up task.</p> <p>Test to pass: <code>com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullTest</code></p> <p>Class to modify: <code>com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullCheck</code></p>	<h3>Current Time</h3> <div style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center;"><p>__ : __</p><p>hours minutes</p></div>
	<h3>Tasks</h3>

Figure A.18. Participant2, pages 21-24

	<div data-bbox="839 349 1262 394" style="background-color: #cccccc; padding: 5px;">Preparing for Task 1</div> <div data-bbox="839 405 1262 495" style="border: 1px solid #ccc; padding: 5px;"> <p>For Task 1, you are not allowed to use the visualization, please minimize it (do not close it)</p> </div>
<div data-bbox="344 1055 756 1099" style="background-color: #cccccc; padding: 5px; text-align: center;">Current Time</div> <div data-bbox="389 1305 711 1435" style="border: 1px solid #ccc; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <p>— — . — — hours minutes</p> </div>	<div data-bbox="839 1055 1262 1099" style="background-color: #cccccc; padding: 5px;">Improving MethodCountCheck Task 1</div> <div data-bbox="839 1115 1262 1149" style="border: 1px solid #ccc; padding: 5px;"> <p>The goal of this task is to improve the code of class <code>MethodCountCheck</code>, making sure you do not break the tests. You should perform the refactoring described below.</p> </div> <div data-bbox="839 1160 1262 1205" style="border: 1px solid #ccc; padding: 5px;"> <p>Refactoring of checkCounters(MethodCounter oCounter, DetailAST oAst): Right now <code>checkCounters</code> has many repetition of "if" statements. Create a utility method that reports if a maximum has been exceeded and refactor <code>checkCounters</code> to call it.</p> </div> <div data-bbox="839 1216 1262 1261" style="border: 1px solid #ccc; padding: 5px;"> <p>Your utility method should have the following signature: <code>checkMax(int oMax, int oValue, String oMsg, DetailAST oAst)</code> where <code>oMax</code> is the maximum allowed value, <code>oValue</code> is the actual value, <code>oMsg</code> is the message to log, and <code>oAst</code> is the AST to associate the message with.</p> </div> <div data-bbox="839 1272 1262 1305" style="border: 1px solid #ccc; padding: 5px;"> <p>After the refactoring is done, rerun the tests to make sure <code>MethodCountCheckTest</code> is still passing. To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="839 1608 1262 1671" style="background-color: #cccccc; padding: 5px; border: 1px solid #ccc;"> <p>Test to pass: <code>com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheckTest</code> Class to modify and check in: <code>com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheck</code></p> </div>

Figure A.19. Participant2, pages 25-28

<div data-bbox="344 349 756 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 600 711 725" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="841 349 1259 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Preparing for Task 2</div> <div data-bbox="841 412 1259 510" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 250px;"> <p>For Task 2, you are allowed to use the visualization and the notifications about the activity of your colleague.</p> <p>The experiment manager will tell you how to enable the notifications.</p> <p>After this, notifications and alerts will appear in the visualization.</p> </div>
<div data-bbox="344 1052 756 1097" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 1303 711 1429" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 20px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="841 1052 1259 1097" style="background-color: #cccccc; text-align: center; padding: 5px;">Finishing PlainTextLogger Task 2</div> <div data-bbox="841 1115 1259 1585" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 250px;"> <p>The goal of this task is to fix the test <code>PlainTextLoggerTest</code> by changing class <code>PlainTextLogger</code>.</p> <p><code>PlainTextLogger</code> is a class to output the violations as plain text, similarly to <code>DefaultLogger</code> but, with customized log message.</p> <p>In the following, we show an example of an output of a check formatted in plain text and default text.</p> <p>Plain text:</p> <pre>Starting audit... Starting file=Test.java Test.java line=1 column=1 severity=warning message=key Finished file=Test.java Audit done.</pre> <p>Default text:</p> <pre>Starting audit... Test.java:1:1: warning: key Audit done.</pre> <p>Currently, there are two broken tests: <code>testAddError</code> and <code>testFileStarted</code>. Fix them in the following order:</p> <ol style="list-style-type: none"> 1. Fix <code>testAddErrorColumn</code>: This test is failing because method <code>addError(AuditEvent evt)</code> in <code>PlainTextLogger</code> is not printing the column information. Add this information and make sure the test pass before you go to the next fix. The column information should only be printed if it's greater than 0 (attention not to break <code>testAddErrorColumn2</code>). 2. Fix <code>testFileFinished</code>: This test is failing because method <code>fileFinished(AuditEvent evt)</code> in <code>PlainTextLogger</code> is currently empty. Implement this method and rerun the tests. <p>To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="841 1603 1259 1666" style="background-color: #cccccc; padding: 5px; margin-top: 10px;"> <p>Test to pass: <code>com.puppcrawl.tools.checkstyle.PlainTextLoggerTest</code> Class to modify and check in: <code>com.puppcrawl.tools.checkstyle.PlainTextLogger</code></p> </div>

Figure A.20. Participant2, pages 29-32

<div data-bbox="344 349 756 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 600 711 725" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 40px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 40px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 349 1251 394" style="background-color: #cccccc; text-align: center; padding: 5px;">Preparing for Task 3</div> <div data-bbox="839 412 1251 510" style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 200px;"> <p>No further preparations are necessary</p> </div>
<div data-bbox="344 1052 756 1097" style="background-color: #cccccc; text-align: center; padding: 5px;">Current Time</div> <div data-bbox="389 1303 711 1429" style="border: 1px solid black; border-radius: 10px; padding: 20px; text-align: center; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 40px; margin: 0 auto;"></div> <p>hours</p> </div> <div style="text-align: center;"> <p>·</p> </div> <div style="text-align: center;"> <div style="border-bottom: 1px solid black; width: 40px; margin: 0 auto;"></div> <p>minutes</p> </div> </div> </div>	<div data-bbox="839 1052 1251 1097" style="background-color: #cccccc; text-align: center; padding: 5px;"> Finishing JsonLogger Task 3 </div> <div data-bbox="839 1115 1251 1590" style="border: 1px solid black; padding: 10px;"> <p>The goal of this task is to fix the test <code>JsonLoggerTest</code>.</p> <p><code>JsonLogger</code> is a class to output the violations in JSON (Javascript Object Notation) format. JSON is a data-interchange format that is easy to read/write and parse/generate. JSON is built in two structures:</p> <ul style="list-style-type: none"> • A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. • An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. <p>We show an example of an output of a check formatted in JSON and XML (See <code>XMLLogger</code>).</p> <pre> {"checkstyle version":"5.3" {"file":{ "name":"Test.java" {"error":{ "line":"9" "column":"48" "severity":"error" "message":"Parameter text should be final." "source":"Test" }} }} } </checkstyle> <?xml version="1.0" encoding="UTF-8"?> <checkstyle version="5.3"> <file name="Test.java"> <error line="9" column="48" severity="error" message="Parameter text should be final." source="Test"/> </file> </checkstyle> </pre> <p>In this task, you should fix two tests in the following order:</p> <ol style="list-style-type: none"> 1. <code>testAddInfo</code>: This test is failing because the method <code>addError(AuditEvent evt)</code> in <code>JsonLogger</code> is printing "error" when the severity level is INFO. Change it to print "info" when the severity level is INFO (attention not to break <code>testAddError</code>). 1. <code>testFileStarted</code>: This test is failing because the method <code>fileStarted(AuditEvent evt)</code> in <code>JsonLogger</code> is empty. Implement it and make the test pass. <p>To complete the task, check in the changed class to the repository.</p> </div> <div data-bbox="839 1608 1251 1666" style="background-color: #cccccc; padding: 5px;"> <p>Test to pass: <code>com.puppycrawl.tools.checkstyle.JsonLoggerTest</code> Class to modify and check in: <code>com.puppycrawl.tools.checkstyle.JsonLogger</code></p> </div>

Figure A.21. Participant2, pages 33-36

<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Current Time</div> <div style="text-align: center; border: 1px solid black; border-radius: 10px; padding: 20px; margin: 20px auto; width: 150px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> ____ hours </div> <div style="text-align: center;"> : </div> <div style="text-align: center;"> ____ minutes </div> </div> </div>	<div style="text-align: center; background-color: #cccccc; padding: 20px; margin: 20px auto; width: 250px;">Post-experiment Questionnaire</div>																																																																																																						
<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Post-experiment Questionnaire</div> <div style="background-color: #cccccc; padding: 5px; margin-bottom: 10px;">Experiment evaluation</div> <p>Thanks for completing the tasks! To get an impression of your experience with the experiment and to allow you to give your comments, please fill in the questions below.</p> <p>1 This question is about your overall experience in performing the experiment. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> </tr> </thead> <tbody> <tr> <td>Overall, the tasks were feasible</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I felt time pressure</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I would have needed more guidance to complete the tasks</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The warm up phase was useful</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The tasks were interesting to do</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The tasks were realistic</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The experiment was fun to do</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> </tbody> </table>		1	2	3	4	5	Overall, the tasks were feasible	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I felt time pressure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I would have needed more guidance to complete the tasks	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The warm up phase was useful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The tasks were interesting to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The tasks were realistic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The experiment was fun to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<div style="text-align: center; background-color: #cccccc; padding: 5px; margin-bottom: 20px;">Post-experiment Questionnaire</div> <p>2 These statements relate to the usability of the visualization and its effectiveness to support program comprehension. Please rate the following statements on a scale from 1 to 5. 1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> </tr> </thead> <tbody> <tr> <td>The visualization is effective in supporting program comprehension</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The ability to access code from the visualization is very important</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The navigation system is easy to use</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>Tooltips are usable and provide useful information</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>Overall the visualization is usable and intuitive</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The visualization is not usable on a laptop screen, as it is too small</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>The visualization highlights important system components</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> <tr> <td>I would use the visualization in everyday coding</td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> <td><input type="radio"/></td> </tr> </tbody> </table>		1	2	3	4	5	The visualization is effective in supporting program comprehension	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The ability to access code from the visualization is very important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The navigation system is easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Tooltips are usable and provide useful information	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Overall the visualization is usable and intuitive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The visualization is not usable on a laptop screen, as it is too small	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	The visualization highlights important system components	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	I would use the visualization in everyday coding	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	1	2	3	4	5																																																																																																		
Overall, the tasks were feasible	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I felt time pressure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I would have needed more guidance to complete the tasks	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The warm up phase was useful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The tasks were interesting to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The tasks were realistic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The experiment was fun to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
	1	2	3	4	5																																																																																																		
The visualization is effective in supporting program comprehension	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The ability to access code from the visualization is very important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The navigation system is easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
Tooltips are usable and provide useful information	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
Overall the visualization is usable and intuitive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The visualization is not usable on a laptop screen, as it is too small	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
The visualization highlights important system components	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		
I would use the visualization in everyday coding	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																																																																		

Figure A.22. Participant2, pages 37-40

Bibliography

- [BCSR07] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, *Fastdash: a visual dashboard for fostering awareness in software teams*, CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems), ACM, 2007, pp. 1313–1322.
- [BE96] Thomas Ball and Stephen G. Eick, *Software visualization in the large*, IEEE Computer **29** (1996), no. 4, 33–43.
- [CKI88] B. Curtis, H. Krasner, and N. Iscoe, *A field study of the software design process for large systems*, Communications of the ACM **31** (1988), no. 11, 1268–1287.
- [DB92] P. Dourish and V. Bellotti, *Awareness and coordination in shared workspaces*, Proceedings of the CSCW 1992 (ACM Conference on Computer-supported Cooperative Work), ACM Press, 1992, pp. 107–114.
- [Die07] Stephan Diehl, *Software visualization: Visualizing the structure, behaviour, and evolution of software*, Springer, 2007.
- [DISK07] D. Damian, L. Izquierdo, J. Singer, and I. Kwan, *Awareness in the wild: Why communication breakdowns occur*, Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering), IEEE Computer Society, 2007, pp. 21–30.
- [dSCdW⁺06] I. da Silva, P. Chen, C. V. der Westhuizen, R. Ripley, and A. van der Hoek, *Lighthouse: Coordination through emerging design*, ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange), ACM Press, 2006, pp. 11–15.
- [ESJ92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E. Jr, *Seesoft: A tool for visualizing line oriented software statistics*, IEEE Transactions on Software Engineering **18** (1992), no. 11, 957–968.
- [Gri96] R. Grinter, *Supporting articulation work using software configuration management systems*, Computer Supported Cooperative Work **5** (1996), no. 4, 447–465.
- [Guz09] Anja Guzzi, *Supporting collaboration awareness in multi-developer projects*, Master’s thesis, University of Lugano, Switzerland, 2009.
- [HL09a] L. Hattori and M. Lanza, *Mining the history of synchronous changes to refine code ownership*, MSR 2009 (6th IEEE Working Conference on Mining Software Repositories), IEEE CS Press, 2009, pp. 141–150.

- [HL09b] Lile Hattori and Michele Lanza, *An environment for synchronous software development*, Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering, ACM, 2009, pp. 223–226.
- [HL10] ———, *Syde: A tool for collaborative software development*, Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering), ACM, 2010, pp. 235–238.
- [HP08] R. Hegde and P. Dewan, *Connecting programming environments to support ad-hoc collaboration*, ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering), IEEE CS Press, 2008, pp. 178–187.
- [KM00] Claire Knight and Malcolm C. Munro, *Virtual but visible software*, Proceedings of the International Conference on Information Visualisation, IEEE Computer Society Press, 2000, pp. 198–205.
- [Lan99] Michele Lanza, *Combining metrics and graphs for object-oriented reverse engineering*, Master’s thesis, University of Berne, Switzerland, 1999.
- [LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine, *Maintaining mental models: a study of developer work habits*, Experience report in International Conference on Software Engineering (ICSE) 2006, ACM, 2006, pp. 492–501.
- [Mal07] Jacopo Malnati, *X-Ray - An Eclipse Plug-in for Software Visualization*, Bachelor’s thesis, University of Lugano, Switzerland, 2007.
- [MK88] H.A. Muller and K. Klashinsky, *Rigi: a system for programming-in-the-large*, ICSE ’88: Proceedings of the 10th International Conference on Software Engineering, IEEE Computer Society Press, 1988, pp. 80–86.
- [Mul86] Hausi A. Muller, *Rigi Ñ a model for software system construction, integration, and evaluation based on module interface specifications*, Ph.D. thesis, Rice University, 1986.
- [Par94] David Lorge Parnas, *Software aging*.
- [PBG03] Thomas Panas, Rebecca Berrigan, and John Grundy, *A 3d metaphor for software production visualization*, Proceedings of the Seventh International Conference on Information Visualization, IEEE Computer Society Press, 2003, pp. 314–319.
- [PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small, *A principled taxonomy of software visualization*, Journal of Visual Languages and Computing **4** (1993), no. 3, 211–266.
- [PEQ⁺07] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, *Communicating software architecture using a unified single-view visualization*, Proceedings of 12th the IEEE International Conference on Engineering Complex Computer Systems, IEEE Computer Society Press, 2007, pp. 217–228.
- [SB99] Matthew L. Staples and James M. Bieman, *3-d visualization of software structure*, Advances in Computers **49** (1999), 96–143.

- [Sch01] T. Schümmer, *Lost and found in software space*, HICSS 2001 (34th Annual Hawaii International Conference on System Sciences), IEEE Computer Society, 2001.
- [SGPP04] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette, *Mining a software developer's local interaction history*, MSR 2004 (1st International Workshop on Mining Software Repositories), 2004, pp. 106–110.
- [SRvdH08] A. Sarma, D. Redmiles, and A. van der Hoek, *Empirical evidence of the benefits of workspace awareness in software configuration management*, FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Engineering), ACM Press, 2008, pp. 113–123.
- [SvdH06] A. Sarma and A. van der Hoek, *Towards awareness in the large*, First International Conference on Global Software Engineering, 2006, pp. 127–131.
- [TC09] A. R. Teyseyre and M. R. Campo, *An overview of 3d software visualization*, IEEE Transactions on Visualization and Computer Graphics **15** (2009), no. 1, 87–105.
- [War04] Colin Ware, *Information visualization: perception for design*, Morgan Kaufmann Publishers Inc., 2004.
- [Wet10] Richard Wettel, *Software systems as cities*, Ph.D. thesis, University of Lugano, Switzerland, 2010.
- [Wik] Wikipedia, *Vuze*, <http://en.wikipedia.org/wiki/Vuze>.
- [WL07] Richard Wettel and Michele Lanza, *Program comprehension through software habitability*, Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension), IEEE Computer Society, 2007, pp. 231–240.