Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 15, 2016

CSI:Cub8

Augmenting Software System Representation with Corollary Information

Marco Bedulli

Abstract

Understanding a software system is one of the most time-consuming activities of a developer. The information that is available for a piece of code could come from the code itself or from online resources. One of the most well known Q&A website in which a huge amount of information is available on StackOverflow. All this information is useful to understand the information coverage of a software system. The information not strictly related to a software system, likes forum discussions and code documentation can be useful to understand how much knowledge is available about the source code. This information comes from different sources and there is no way to visualize in once. Collecting this data and using some visualisation method could helps the developer to spot the part of the code that require more documentation or they could give an idea about the effort require to understand the software system. Cub8 uses an augmented city metaphor as visualization method, it allow the developers to evaluate the information coverage of the system. A developer is thus able to visualize which part needs more documentation and also directly access the online information related to it.

Advisor
Prof. Dr. Michele Lanza
Assistants
Phd. Andrea Mocci, Luca Ponzanelli

Advisor's approval (Prof. Dr. Michele Lanza): Date:

Contents

List of Figures

1 Introduction

Developers invest great effort recovering implicit knowledge by exploring code or by interrupting teammates. This knowledge was only saved in their memory [8]. According to Corbi, developers understand programs by reading documentation, reading source code, and running the program itself[4]. Understanding absorbs about half of the time of developers [4, 6]. In fact researchers showed that developers spend more time reading than writing source code [9].

To alleviate this problem and try to reduce the time spent by the developers to understand the code, we propose a tool that visualize, using a 3d city metaphor, the information coverage of a system and helps the developers to spot the parts of the system that need more documentation. The city is created using a mix of all the information available to the code, mapped to assemble the building of the city. The use of a metaphor from the physical world is the key point that makes this system particularly intuitive and effective [7]. This kind of visualization empowers the capacity of the viewer to transfer existing perceptual abilities through the understanding of the representation [16].

R. P. Gabriel [5] said that "Habitability is the characteristic of source code that enables programmer, code, bug-fixer, and people coming to the code later in life to understand his construction and his intentions[...]". Starting from this concept, we use this idea of habitability [17], where this metaphor is used as a way to allow the developers to get a better understanding of a specific software system.

As main aim, we thought to help all the developers that, coming to the system later during its development, need to be filled in quickly about all the reference and the information available on it. The developers can navigate and interact with all the city's components, from the folder (the basement) to all the file that compose the project (building), live in Figure 1.

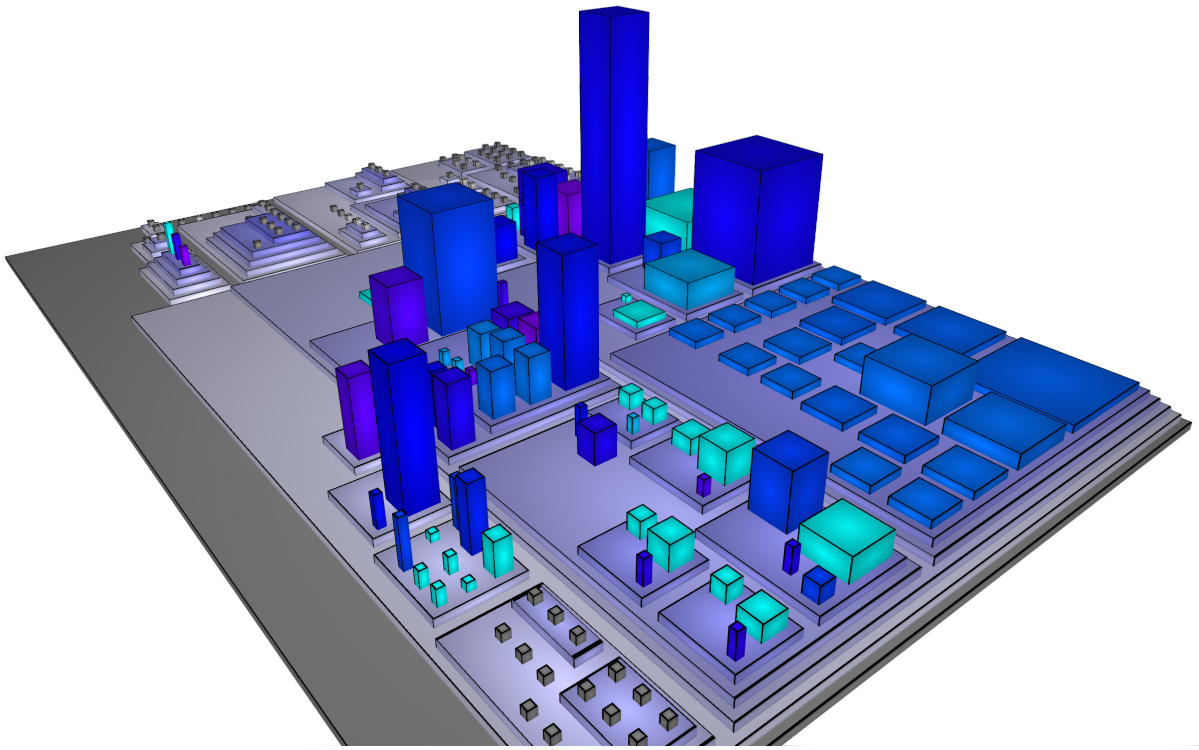


Figure 1. A first example of a city

Figure 1 depicts an example of a city generated using Cube8. Other approaches focus they intents to found code disharmony [16] or code quality [15] all aspect strong related to the code structure but there are no approaches that analyze the information related to a software. The developers understand programs by reading documentation, whose information is useful to reduce the effort spent by the developers to understand code [4].

The information that is available for a piece of code could come from the code itself or from online resources. In both cases we refer to this data as a corollary information. The Java doc can be used as a simple case of corollary information because it is not essential for the design purpose, but is extremely useful to understand what the code does. Usually developers do not write down knowledge in design documents [8]. With this tool we can spot this information issues. The majority of the developers use Q&A website to make how-to questions about code issues [13]. Stack Overflow is one example of Q&A website, where questions are asked on a daily basis [11]. According

to the last data dump of August 2015, Stack Overflow relies on a community counting 4.5M users, who produced about 26M posts, out of which 900K questions concern the Java language. All this information is very useful through the process of code comprehension. The goal of this thesis it to give to the developers a tool that can visualyze an entire system with all the associated corollary information. Cub8 provides a way to interact and visualize all this information coming from Stack Overflow and Java Documentations.

In Section 2 we present the related work. We describe the functionalities of Cube8 and we briefly explain how StORMeD works. In Section 3, we explain the approach and the different metrics that we used. In Section 4 we show two different use cases on how Cub8 can be used and which kind of information is possible to retrieve. Finally, we present the conclusions and we discuss the future work.

2 Related Work

In the past decades, researchers studied several approaches to visualize a software system. The basic idea is to approximate a software system to a more familiar system in which the developers are able to orient and navigate. There are two main metaphor for visualization methods the landscape metaphor and a city metaphor. The term landscape metaphor is not to be understood exclusively as a detailed image of reality, but rather stands for structures that are similar to those of a real landscape [2]. The city metaphor as visualization method, used by Knight and Munro [7], and Component City [3] and later by [17, 16] is intended to represent a software system as a city. The main difference between these systems is the data that they analyze. In [18] the intent is to render the structural evolution of object-oriented software systems at both a coarse-grained and a fine-grained level. *Software word* [7] is a system in which the buildings represents the functions, the size of a building represent the number of lines of code and the color the modifier of the function.

Wettel and Lanza [16] propose a city metaphor in which there is a fixed number of building types such as skyscraper, office building, apartment block,mansion, and house. They propose two mappings: boxplot-based mapping and threshold-based mapping. Also, they use a box-packing algorithm to visualize the city. The city metaphor consist into classes represented as a building located in city districts which in turn represent packages. The goal of the visualization is to show the structure of a software. [17] propose a 3d environment in which the software system is represented as a city, whit different class of buildings. It also implements a way to navigate and interact with the system. It is possible to select any artifacts and interact with them, spawning complementary views(i.e., a tagging system and a query system). Tymchuk *et al.*[15] propose an approach to augment code review by integrating software quality evaluation, and more general design assessment, not only as a first class citizen, but as the core concern of code review. In Cub8 we have get some of this ideas: the navigation and the interaction through the system and the boxing building methodology to generate the city but we also delegate to the user the decision of the different metrics to assign to the dimensions of a building. The corollary information that we are using to give to the developers a more complete idea about the information available is taken from Stack Overflow. We use the StORMeD dataset, it consists of a set of JSON files that contain an H-AST of each discussion. It implements an island-parsing algorithm [12]. There are also other approaches that used Stack Overflow as a way to retrieve data.

SISE [14] use machine learning based approach where the features are the sentences themselves, their formatting, their question, their answer, their authors ,the part-of-speech tags and the similarity of a sentence to the corresponding API. Arwan *et al.*[1] proposes a mechanism for finding code on Stack Overflow uses Latent Dirichlet Allocation (LDA) using concept location in the preprocessing stage.

3 Approach

Cub8 allows the developers to apply different metrics on a city and mix different views together to get a global idea about the system coverage. In this chapter we are going to exploit and analyze what these metrics does and which kind of information is possible to retrieve. We concentrate our studies on Java based system. Java is a well known Object oriented language and therefore we refer to classes, interfaces, methods and fields as the main part of project. We also use some notation for identify the different metrics:

- NOM: Number of Methods, represent the number of method declarations contained in a file
- NOF: Number of Fields, represent the number of fields contained in a file.
- NOJ: Number of Java Documentation, count how many method declaration are commented; its absolute value represent the number of method documented, its percentage represent the number of method documented over the total amount of the methods of the file.
- NOD: Number of Forum Discussion, count how many method call and import are discussed; its absolute value represent the number of method call and import discussed, its percentage represent the number of method call and import discussed over the total amount of the possible method call and import. We do not count local package and methods that are declare in the project.

3.1 Information strictly related to the code

The code related information are used to give to the developers a better understanding about the locality of the code. To demonstrate this concept we compare two different cities computed on the same code. The system consists of only two classes. ClassA has 4 methods and ClassB has 4 methods plus 4 other fields. The main goal of this example is to find which class needs more documentation.

Figures 2 and 3 show the mapping; the color schema goes from light blue to purple. Purple mean the maximum and light blue the minimum amount of data either in percentage or in an absolute value. In 2 the color represent the NOJ and in 3 the NOM. In Figure 2, we can notice that the big building on the left has adequate documentation, instead, the right one has no documentation. By looking at the mapping and the class description ClassA is our target, in fact it is thin and tall this means that it has the NOM » NOF. In Figure 3 instead we represent more corollary information at the same time using different axes. The city becomes unusable because it is indistinguishable which is the ClassA and which is the ClassB since the only difference is on the number of fields. Figure 3 the developers can make some general observation about the relationship between java documentation and discussions. Later in this section, we are going to analyze in more detail the color metrics system and the purpose of the other adopted metrics.

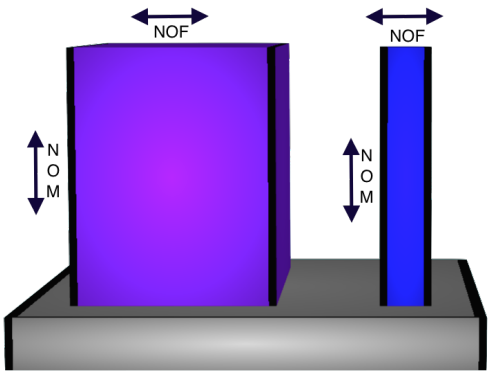


Figure 2. Mapping as Width: number of methods, height: number of fields, colour: javaDoc

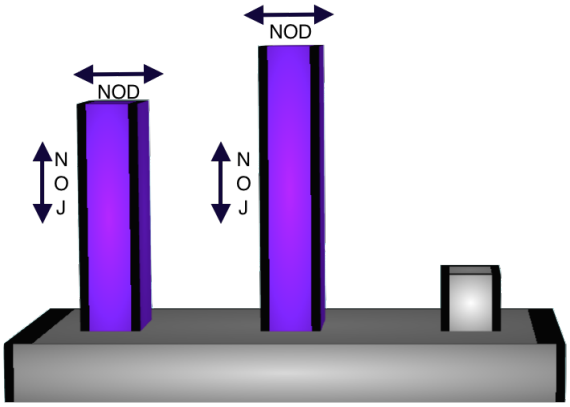


Figure 3. Mapping as width: number of discussions , height: number of Java documentations, color: number of methods

3.1.1 Classes and interfaces

Classes and Interface are the only two possible code containers in Java. Cub8 allows the developers to analyse their distribution and highlight possible bad design choics. Note that as Oracle [10] says: "Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, the developers can put them in the same source file as the public class", the direct consequence that arises is that we could have more classes in a single source file and therefore it is also legal to have buildings with different size on the resulted city.

When there are to many classes in the same file, an important information that can be retrieve from this visualization is the average level of coupling in the code. In a file with a huge amount of classes; there could be a high degree of coupling, could it will be harder to maintain it. We will discuss an example of this representation method during the analysis of TomCat in Section 4.

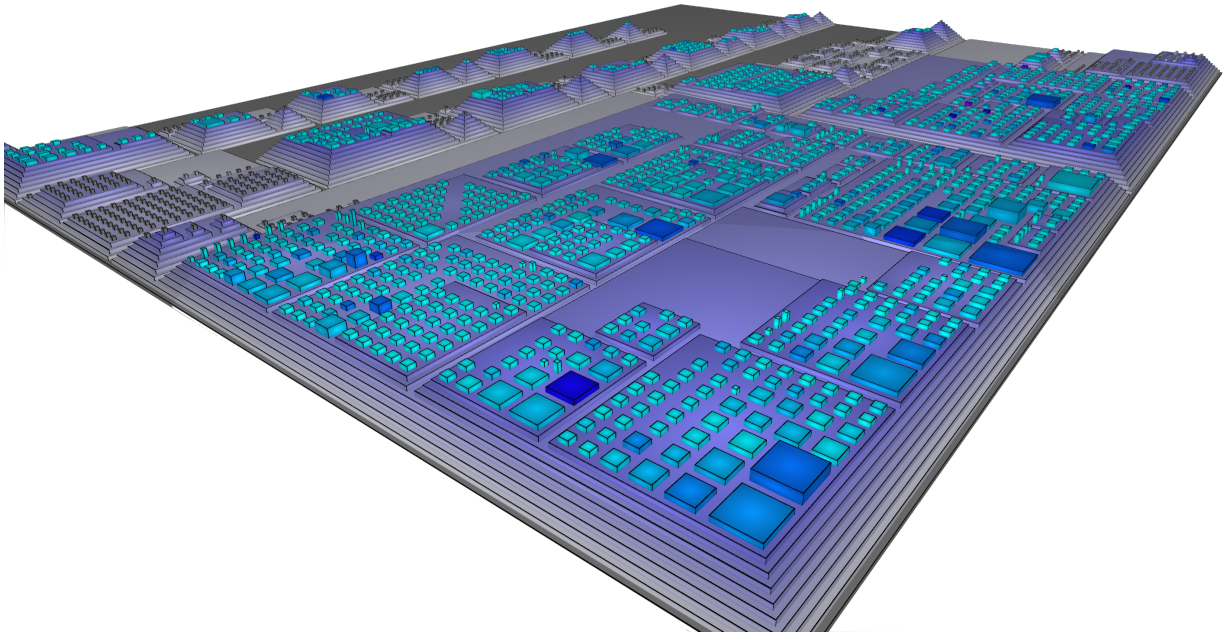


Figure 4. Mapping as Width:Number of Classed, Height: Number of interfaces

Figure 4, shows an example where we analyze this two concepts in a big project. There are a few classes that need some check to make sure that this design principle is respected. We use the width to show that it is possible to change metrics respect to what we are going to look at. Recall the concept of habitability [16] we could also maintain the building size as NOM and NOF and changing the color depending of the number of classes. In this way we can still have a idea about the class distributions and keep the structure unchanged.

3.1.2 Identity harmony

Design disharmonies are formalised design shortcomings to detonate pieces of system that exhibit design problem [19]. There are three different type of harmony:

- Identity harmony: every entity in the system must justify its existences.
- Collaboration Harmony: the entity collaborate from each other to perform operations.
- Classification Harmony: The entity could inherit from other entity.

With our tool we can only identify some of the identity harmony. We do not visualize any of the relationship between classes(inheritance) or method collaborations. The three identity harmony that we are able to visualize are

- God Class: it is a class that does too much. In our representation appear like a big box.
- Brain Class: it is a class that accumulate an excessive amount of intelligence,usually has a lot of methods: it is look like an antenna
- Data Class: it is a class that hold a lot of data and does not perform any operation:it appear to a be a big and thin box.

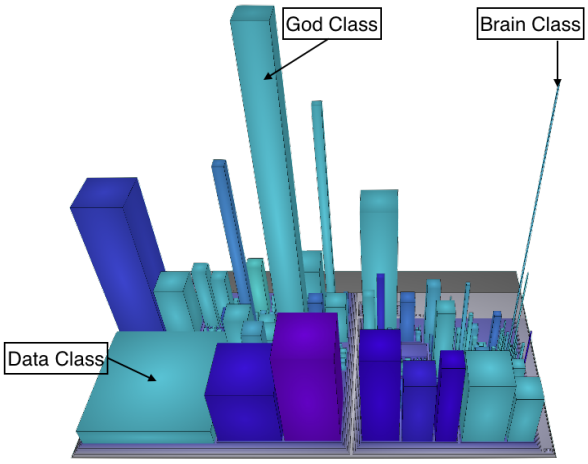


Figure 5. Identity harmony

Figure 5 is an example of this 3 kind of disharmony and how they should look like in the city representation.

3.1.3 Methods and fields

Methods and fields are the main elements that compose a class. We are using this two measure to draft the size of the buildings. The reason why we choose this pair is that they give the correct granularity to have a better perception of the system; Wattel and Lanza [19, 17] use the same mapping. Since we are interested in the visualization of the information coverage that is computed respect to methods.

Using this metrics we are able to identify a potential disharmony as describe in the preview sections. Figure 5 shows an example of each possible disharmonies.

3.2 Information not strictly related to the code

The information not strictly related to the code is the focus of this thesis. This knowledge is essential to get an idea about the complexity of understanding a software system and where more effort must be put. At the same time, this visualization could be used by the developers to monitor which part of their code require more documentation.

3.2.1 Java documentation

Collect and visualize the java documentation is the first step of the process to collect the coverage information because it is integrated with the code and it does not require any particular computations. It covers an important role in the process of understanding the functionality of a given code since is written directly by the developers and should be used in each method, field and class definitions.

This views allows the developers to spot the pieces of the system that require more documentation. We are collect only the documentation related to the methods. We choose this level of granularity because it gives a good level of details. It is impossible to have code documentation per line and is also not much useful class documentation to understand its details. We usually map the java documentation using the colours. In this way we give to the developers a fast way to understand all the data information.

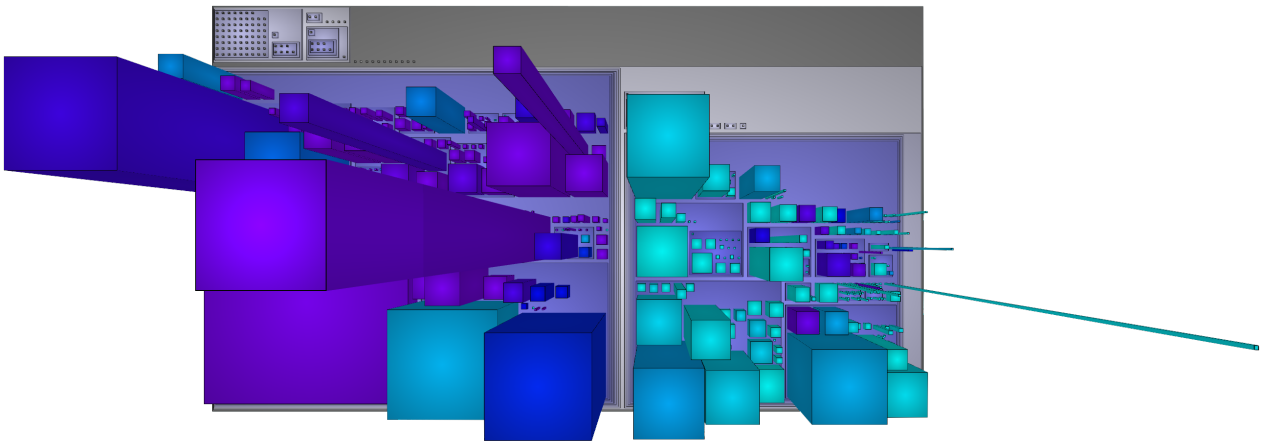


Figure 6. Mapping as Width:N of Fields, height: Number of Method,Colour: Percentage of documented methods

Figure 6 is an example of a city in which the colour represent the percentage of documented methods of the *apache common-lang*[1] library. It is very interesting to see that half of the project has a documentation coverage around 80%, while the other half documentation is very limited. In reality, this is a common case as a lot of projects do not report an adequate documentation in the tests.

To help developers to fully understand the documentation coverage, we provide a package base colouring system, where the colour of each package is the average of its child components. Figure 7 depicts the same project showed in Figure 6. It is more clear to see the global characteristics of the project (at package level) and it appears clear that the test are less documented than the core.

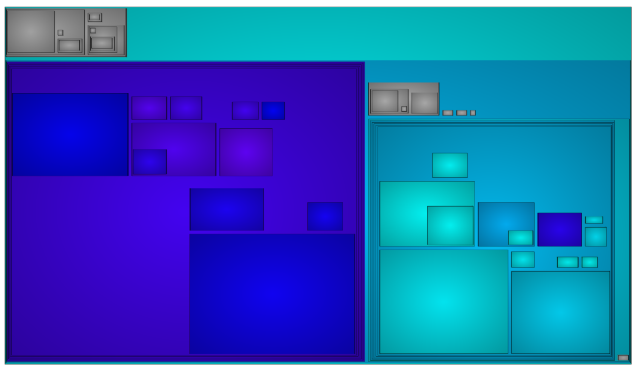


Figure 7. Mapping as Width:N of Fields, height: Number of Method,Colour: Percentage of documented methods

[1] https://commons.apache.org/proper/commons-lang/

3.2.2 Stack Overflow Discussion

Stack Overflow[2] is one of the most popular Q&A website for developers. It contains a lot of code snippets and text related to the code. What we try to do using this visualization method is to show to the user all the available discussions related to each method call. The granularity is different respect to the java doc metric. That allows understanding the complexity to read and understand the methods code not what the method itself does. We get the dataset updated in august 2015. In this stage, we have all the repository code and all the discussions information (methods call and imports) from the StORMeD Dataset[3]. StORMeD contains the code snippet and the code reference in text of each discussion, and provides tools to extract the method calls and the import declarations from the dataset. The majority of the snippet are incomplete so we do not have the type of each call. Also the imports are not used in all snippets [12]. To match the method calls found on the discussions to the method calls found in the source code we simply match the names and the number of arguments. This simplification implies some false positives. The imports are still matched using the package name.

By double-clicking over each building it is possible to get a list of the discussions found. Figure 8 is an example of the discussion found in respect to a building. The color represents the number of discussions in an absolute way. There are two classes that have more discussions that the others. Classes that have more field are colored in light blue.

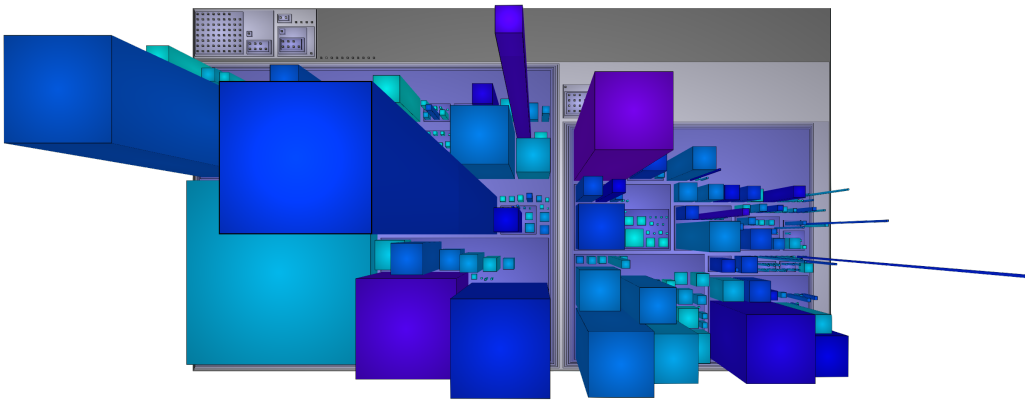


Figure 8. Mapping as Width:N of Fields, height: Number of Method,Colour: Absolute number of discussions

Figure 9 shows an example of list of links for each method call or import declaration.

Discussion URL	
java.util.Arrays	http://stackoverflow.com/questions/25680943 http://stackoverflow.com/questions/21888201 http://stackoverflow.com/questions/2426380 http://stackoverflow.com/questions/17848025 http://stackoverflow.com/questions/20005140 http://stackoverflow.com/questions/27324495 http://stackoverflow.com/questions/26288671 http://stackoverflow.com/questions/23049314 http://stackoverflow.com/questions/27587522 http://stackoverflow.com/questions/23010150
length(0)	http://stackoverflow.com/questions/27223172 http://stackoverflow.com/questions/23820026 http://stackoverflow.com/questions/23530127 http://stackoverflow.com/questions/3699141 http://stackoverflow.com/questions/16570195 http://stackoverflow.com/questions/26089316 http://stackoverflow.com/questions/10541245 http://stackoverflow.com/questions/21764334 http://stackoverflow.com/questions/22419285 http://stackoverflow.com/questions/10564867
java.util.Iterator	http://stackoverflow.com/questions/3939447 http://stackoverflow.com/questions/29472128 http://stackoverflow.com/questions/20344851 http://stackoverflow.com/questions/23049314 http://stackoverflow.com/questions/4385003 http://stackoverflow.com/questions/13870322 http://stackoverflow.com/questions/25366639 http://stackoverflow.com/questions/17729475 http://stackoverflow.com/questions/10596744 http://stackoverflow.com/questions/27366643

Figure 9. List Of Discussions

[2]http://stackoverflow.com/
[3]http://stormed.inf.usi.ch/

3.3 Merge code related information with corollary information

The code related information helps to identify the different components of the city and also it helps to find design problems over the system. The corollary information instead, gives an idea about the information coverage. How can we mix this information together to get a global overview of the entire system? In Figure 8, we use the information related to the code to give the dimension of the building, and we use the color to represent the information coverage. In this way, we improve the concept of locality since a developers should remember a file not for the number of documented methods but for his structure.

3.3.1 Percentage and absolute number of information

The information could be represented as an absolute or a percentage value. In the former way, we count the number of information available and it is possible to see which file contains more information. The latter, instead, it is computed over the total amount of information that could be found. This metric is useful to spot which files have more data. A special case is the forum discussion metrics: we decide to give 0% were we can not find anything related to the code. An example is the number of java doc. If we use the absolute value the color represents the number of the methods documented; otherwise, by using the percentage value, the color represent the methods documented over the total amount of the methods.

3.3.2 Using java doc and discussions together

To get a better understanding about the information coverage we have to join the java documentation and the information related to code. We obtain an average of both since they correspond to two difference level of granularity. The java documentation refers to a method and the discussion refers to either import or method calls. We can show the result in both ways: percentage and absolute. In the former case, the developers can get a better understanding about the percentage of the information available. This is useful to guess the effort require to understand the system. The latter, instead, is used to see where there is more concentration of information and where is not. It could be useful to identify bad documented packages.

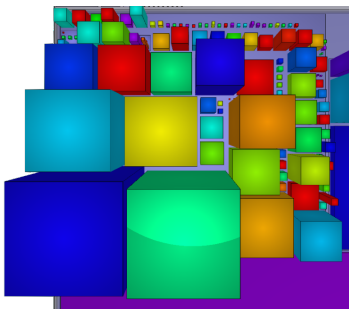
3.4 Colors

The colors are used to show other metrics. They spread from light blue to purple and are very useful to give to the developers a quick impression about the system. In the next section colors are used to map corollary information and to identify some code disharmony.

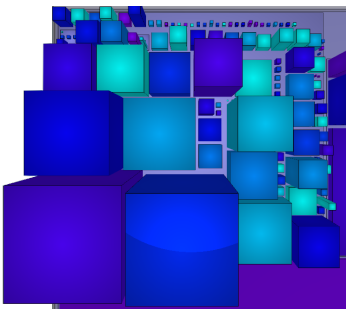
3.4.1 Corollary information colour meaning

The color used to compute the corollary information it is computed as absolute or percentage. In the former case, the purplish building represents the file that has more information coverage; the light blue highlight the file which has less coverage. In the latter case, we see the percentage over the methods. This give a local view about the percentage coverage respect to a file. Colors depend only on a file, not on the whole project like in absolute view.

An example is the discussion coverage. Using the percentage distribution, we get all the full discussed file in purple, independently from the number of method calls. Otherwise, using the absolute distribution we have the file with a higher number of discussions in purple.



(a) Color HSV



(b) Range of Colors

Figure 10. Colors model

We use a subrange of the HSV color model, as shown in Figure 10(b). Figure 10(a), instead, is an example where we use the whole HSV color model. The color goes from red to purple pass thought yellow, green and blue. By comparing Figure 10(a) and Figure 10(b) we note that the whole HSV color model adds more noise to the representation making the city hard to be understand. Figure 11 depicts two different color options, absolute in

(a) Absolute (b) Percentage

Figure 11. Colors Absolute & percentage

Figure 11(a) and percentage Figure 11(b). It is interesting to note that there are some buildings that have the majority of discussions in the absolute view (colored in purple) and at the same time the method has not complete information coverage (in percentage View). An example is circled in Figure 11.

3.4.2 Code related Colour meaning

The color used to compute the code related information represents the number of methods, fields, classes or interfaces in files. It is useful to check the code style, for example the number of classes or interfaces in a file, or to get an idea where the majority of the methods and fields are concentrated. The scale is the same as above: purple means the maximum amount of data and light blue no data available.

Figure 12 shows an example in which the colors represent the number of classes in the project. Here, there are a lot of files that contains only one class and only a few files that contains more then one class.

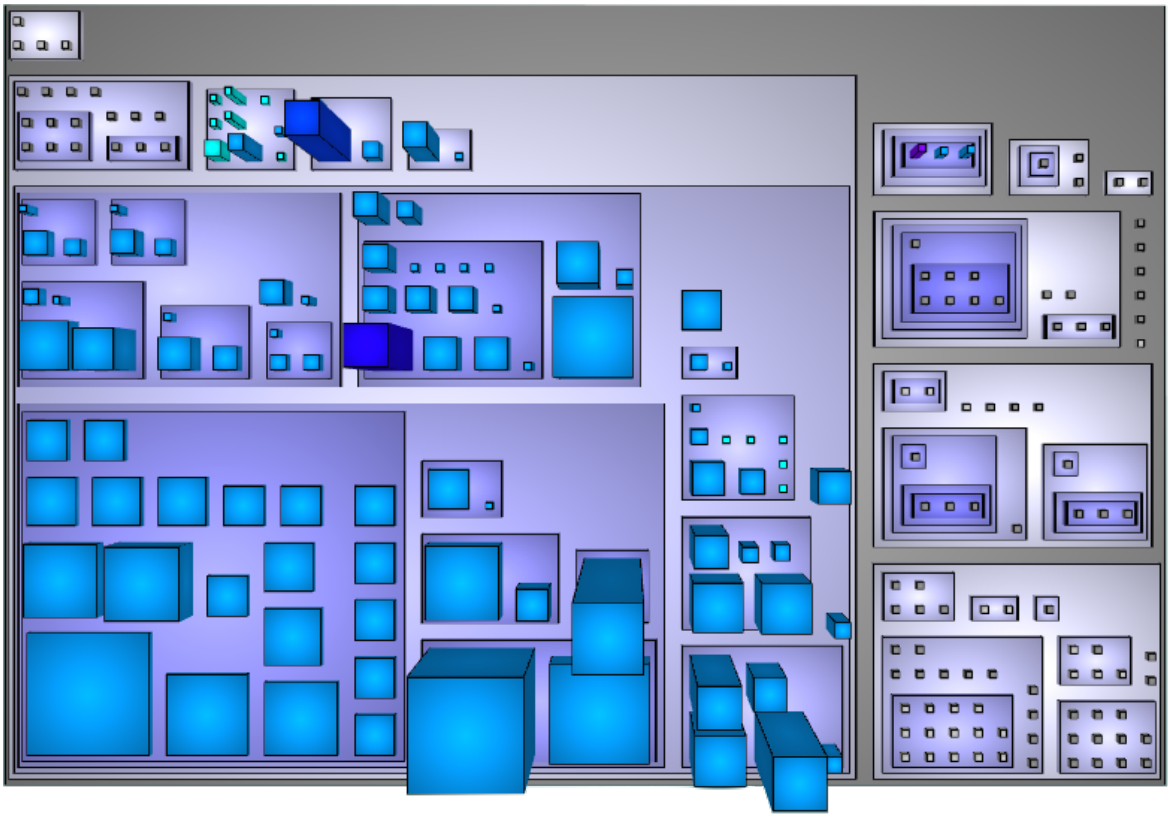


Figure 12. Absolute number of class

3.5 System architecture

The project is divided in to two different parts. The StORMeD importer that imports into the database all the useful data from the StORMeD Dataset and the visualizer that is the part that allows the developers to navigate, interact and modify the city. The former part is written in Scala and latter in Java using the Play Framework. The application is web based.

3.5.1 StORMeD importer

StORMeD importer is nothing else than a visitor of a JSON file that represent a discussion. The JSON file contains an H-AST of the discussion whit all the information. Since we need only a small subset of this information, we extract the useful one, such as method calls with the number of parameters and the imports, and we store it in the database. Since this operation is time-consuming, we adopt a multi-thread solution that reduces drastically the time used to analyze all the files. Only to give an idea, we keep approximately several hours on a server with 56 Intel 2.10 Ghz Xeon processor and 300 GB of RAM.

The script that deals with this is written in Scala combined with a database query and access library called *Slick*[4].

3.5.2 Visualizer

The visualizer is the core of the system. Since it is a web application, it consists of a back-end and a front-end part. All the computation is done on the server side since the amount of data is large; we also choose a strategy to precompute the main metrics and store them as JSON files. The process is represented in Figure 13.

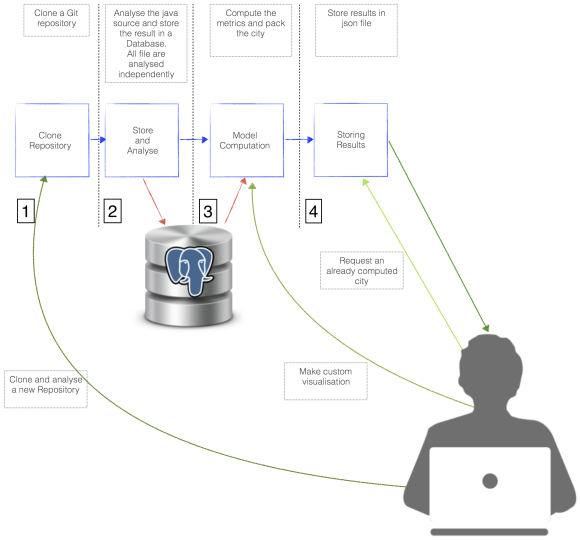


Figure 13. Process Pipeline

The process is as follows:

1. Upload the repository. We use a git subversion system. User have to provide the URL of their git repository and then insert their account information.

2. Files analysis. We traverses the directory of the repository and model an AST for each java file. This process writes into the database all the file characteristics such as classes, methods interfaces and fields(represented as red arrows).

3. Generate the city. All the actions employed at this stage are functions applied to each node in the tree. The first pass through the tree is the size computation, that gives to each node the value for width, height, and color. Since the project structure is a tree, we maintain this structure during all the process. To improve the performance during the visualization, we generate one city for each different metric computation. The packing algorithm is done as if each package reside in the origin of the axes. Only during the rendering time we move the package around the scene.

[4]http://slick.lightbend.com/

4. Store the result in a JSON file. This file contains a tree whit all the information of the render. To obtain extra information like the list of the discussions or the file contents we need to execute some database queries(all possible user interaction are show with green arrows).

We give to the user also the opportunity to make his own city, by deciding through a predefined list of metrics which assign to each dimension. This process repeat exactly the same passages done after the AST. Note that we also cache the result for future requests. The front end part gives to the user a way to interact whit the system. The most interesting part is the rendering of the JSON: the root of the city is draws at the origin and at each recursion we assign a new origin in which the package should be drawn. We also implement a small query system that gives a way to search file or package in the city. By double clicking on a building, it appears a pop-up with the java code and we highlight the keyword to made the code readable. Also, there is a list of discussion views. All the back end is written using *Play Framework*[5] and Java. For the front end we use *bootstrap*[6] and ES6 with *webpack*[7] plus the babel parser *Babylon*. We use *Babylon.js*[8] as 3d engine that works on top of *WebGl*. Since we are on a browser, we can not have more than 10k of boxes.

[5]https://www.playframework.com/
[6]http://getbootstrap.com/
[7]https://webpack.github.io/
[8]http://www.babylonjs.com/

4 Use cases

4.1 Introduction

In this section we are going to analyse two projects by using our tool. The analysis of this project is split in two parts. The former part concern the structure, in which we are looking for code identity harmony (See Section 3.1.2). In the latter part we analyse the information coverage.

For the former part we can not say that a particular design is wrong, we could only give a monitor to the developers to check some port and understand if it is correct. A virtual demo is available on `http://rio.inf.usi.ch:51001/`: it is possible navigate and play with these two projects.

4.2 Use case I: Tomcat

The Apache Tomcat[9] software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Apache Tomcat software is developed in an open and participatory environment. The Apache Tomcat project is intended to be a collaboration of the best-of-breed developers from around the world.

4.2.1 Code related analysis

Figures 14 and 15 depict Tomcat's code related information. The buildings represent the java files post on top of his package. The height of a building is the number of methods and the width is the number of fields. The color represents the number of interfaces (Figure 15) and the number of classes (Figure 14). Generally, we have an equal distribution of class and interface for each file, there are only a few disharmony.

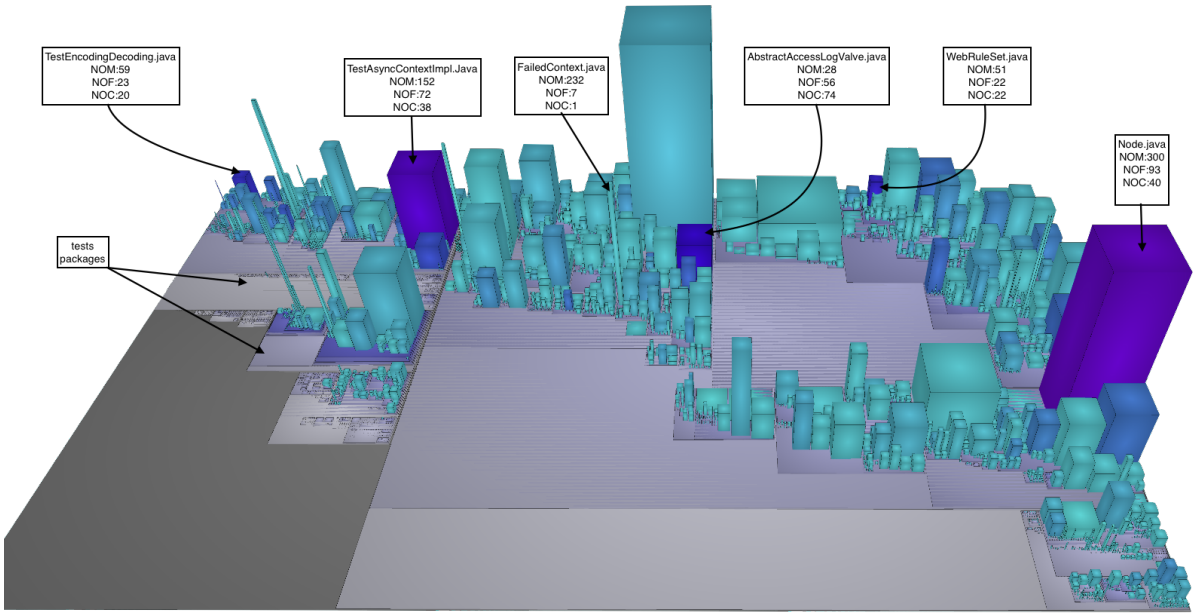


Figure 14. Tomcat: The color represent the classes distribution

In Figure 14 there are some classes that have a high concentration of inner classes. The file has a huge number of private static classes. This is not a problem for the Java Code Conventions [10]: there is not more than one public class and all the other inner classes are inside the public class. The names and the characteristics are shown in the figure. Two of them are test classes, the other three are not. The test classes are fine. The other three classes could have a high level of coupling that is a hint to check the design. Regarding the interface, we have some files that contain more than one. In this case, we have not a big number of interfaces, so it could be a design choose and not a problem. Now we take a look at the methods and fields of a class. As we can expect, the class *Node.java* has a lot of methods but at the same time it has a huge number of classes as showed before. It is a good candidate for a God Class. *StandardContex.java* has the potential to be a God Class either, since it has only three classes and a huge number of methods and fields. In both classes, we can find in a high level of coupling.

[9] http://tomcat.apache.org/

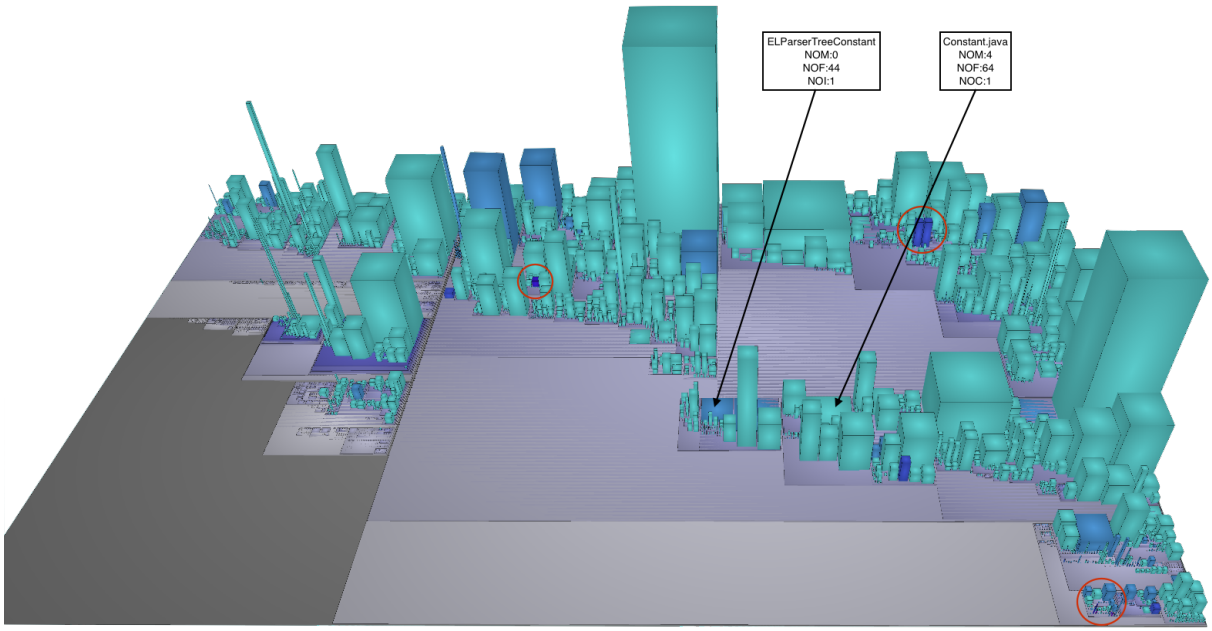


Figure 15. Tomcat: The color represent the interfaces distribution

There are a few buildings that look like a Brain Class. We do not care about test classes since they are, by definition, a list of methods. The first one is *FailledContext.java*, that has a huge amount of methods and no too much fields. In Figure 14 there are other classes of this type. The Data Classes are not too much, one is *Constant.java* and there are others shown in Figure 15.

4.2.2 Corollary information analysis

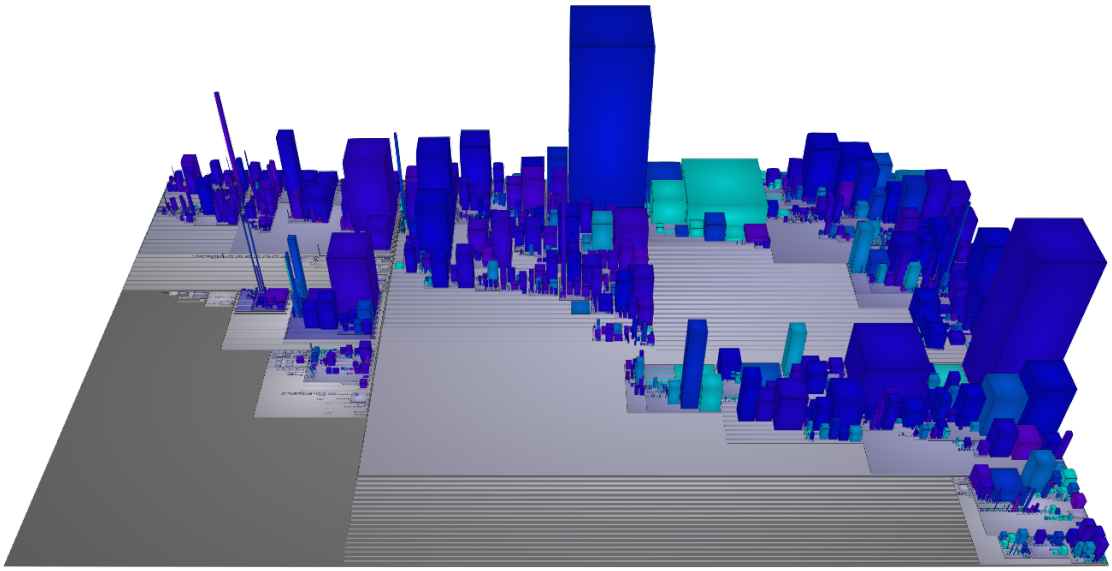


Figure 16. Tomcat: The color represent the interfaces distribution

Figure **??** depicts the information related to the code of the Tomcat repository. The buildings represents the java files post on top of their packages. The height of a building is the number of methods and the width is the number of fields. The color represents the number of discussions over method calls (Figure 16), the amount of java documentations over method declarations (Figure 17) and the information coverage (Figure 18).

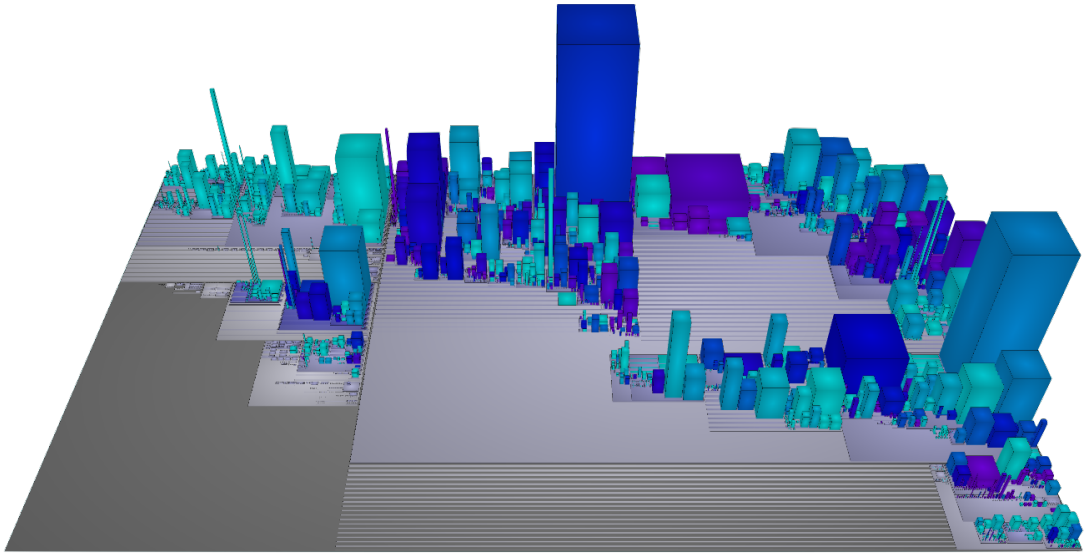


Figure 17. Tomcat: The color represent the Java Documentation distribution

Let's start analyze the java documentation: there are classes completely documented and others that have not documentation at all. The tests are completely not documented and some of the classes that have more methods, have a lower percentage of documentation. This part should be better documented. Instead, the discussion coverage is pretty good. The color of the city in average is dark blue and there are a lot of building purple this means a lot of discussions related to the code. Now that we have the result of both metrics we can merge it together and we have the Figure 18. Thanks to the discussions found online and the code documentations, the the system has a homogeneous information coverage.

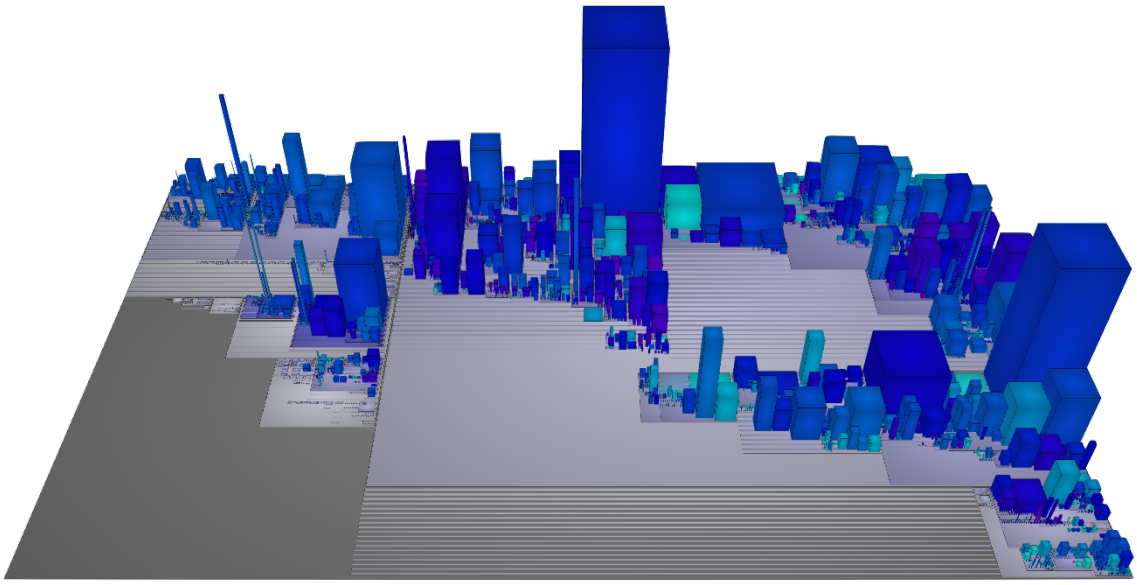


Figure 18. Tomcat: Discussion and Java Documentaton

4.3 Use case II: JGit

JGit[10] is an implementation of the git version control system for java. We analyse the system in the same way as for Tomcat. We decided to use this system since is also used in Cub8. It is an example of an open source product and it is also part of the Eclipse IDE[11].

4.3.1 Code related analysis

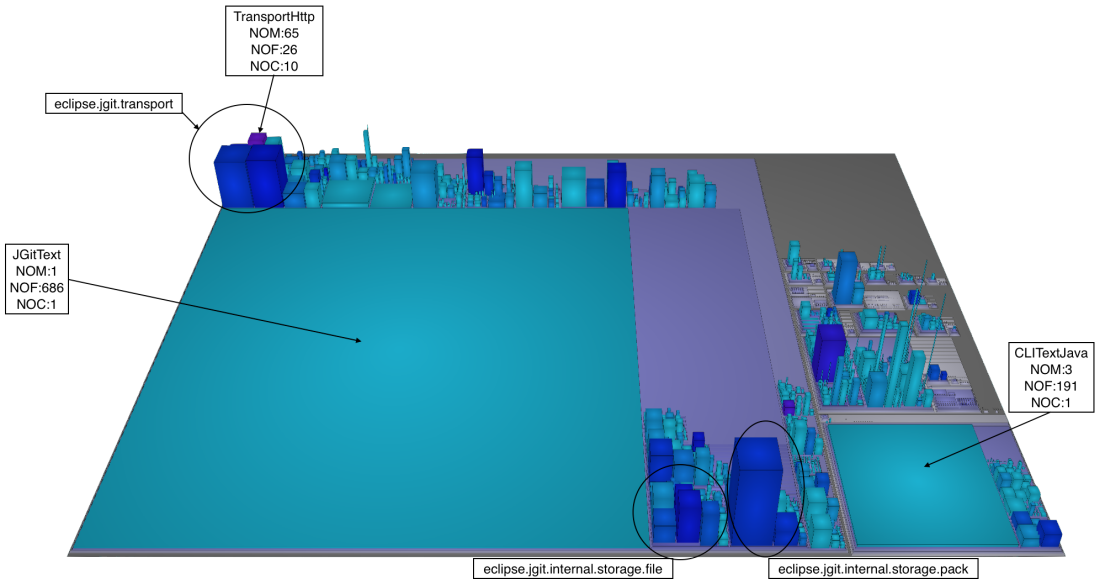


Figure 19. JGit: The color represent the classes distribution

Figures 19 and 20 depicts JGit's code related information. The height of a building is the number of methods and the width is the number of fields. The color, in Figure 19, represents the number of classes, in Figure 20 the number of java interfaces.

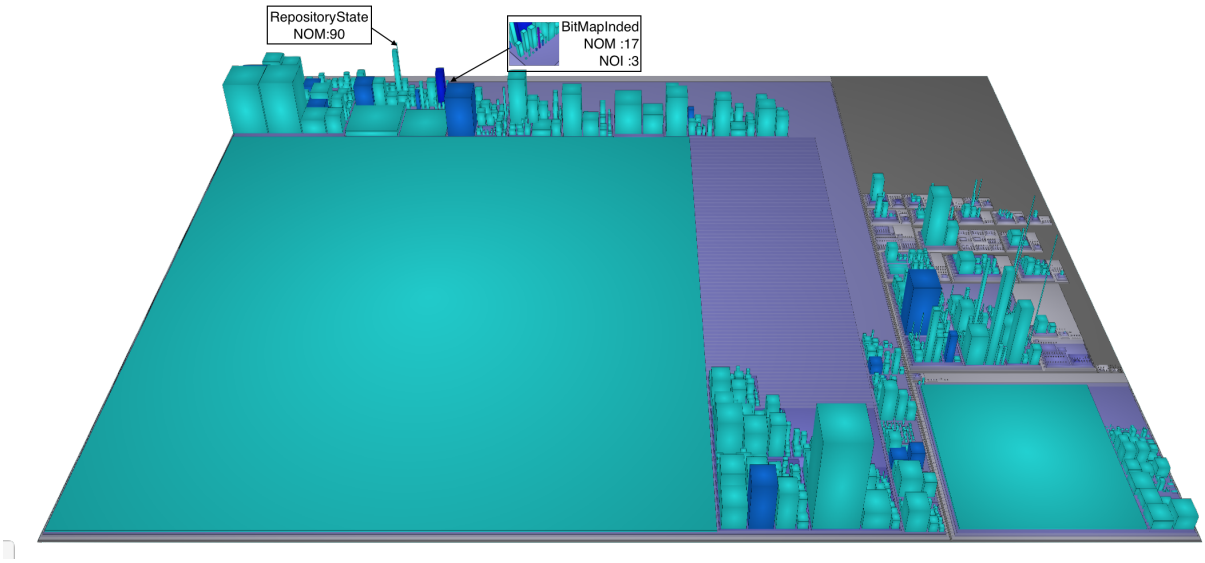


Figure 20. JGit: The color represent the interfaces distribution.

The classes view (Figure 19), shows that there are some files that have more than one classes. It os important to note that the maximum amount of classes for a file in this project is 10. The coupling degree is low. The interface

[10]https://eclipse.org/jgit/
[11]https://eclipse.org/

distribution appears to be well spread, there are only few occurrences of multiple interfaces on the same file. We can see a God Class call *PackWriter.java* that has 48 fields and 121 methods. There are also two big Data Class: *CLIText.java* and *JGitText.java*. At last but not least there is a Brain Class call *RepositoryState.java* that has 90 methods.

4.3.2 Corollary information analysis

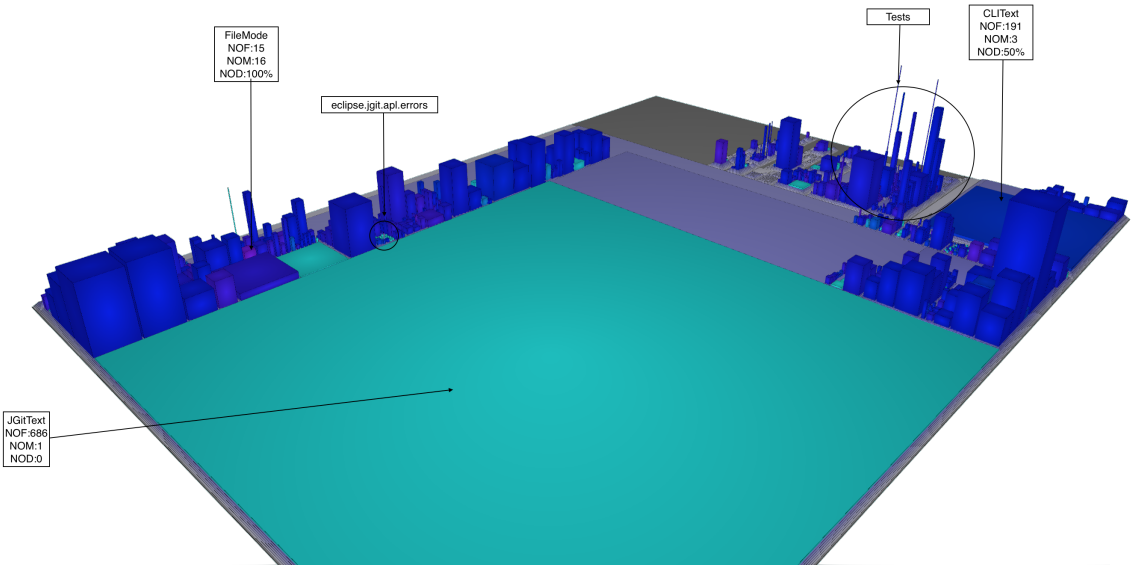


Figure 21. JGit: Discussions express in percentage respect to the number of imports and method calls

Figures 21,22 and 23 depict Jgit's code corollary information. The height of a building is mapped to the number of methods and the width is mapped to the number of fields. In Figure 21 ,the color represents the number of discussions over methods, in Figure 22 the number of java documentation over methods and in Figure 23 the information coverage.

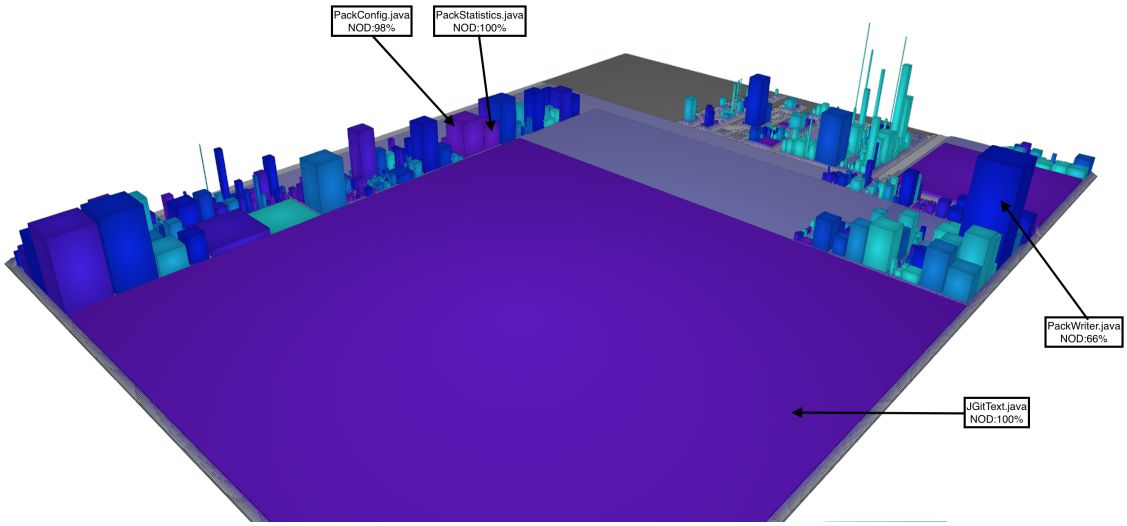


Figure 22. JGit: The color represent the Java Documentation express in percentage respect to the number of methods

Starting by Figure 21 we see that the discussions available are, on average, around the 50%. There are no discussions for JGitText, it has only one method that, the only thing that the method does it is to call another method inside the package; therefore, there are not any discussions available. There are also few classes with a 100% of discussions coverage. The java doc view (Figure 22) shows that there are a lot of building without documentation. Some of them are tests, but a huge number are not and therefore a newcomer has to allocate an higher effort.

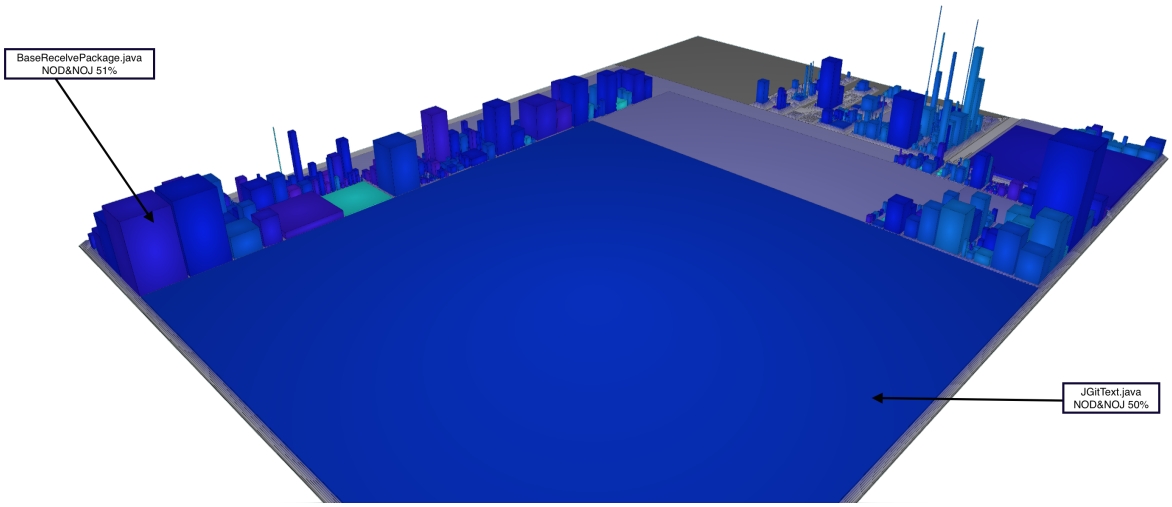


Figure 23. JGit: Discussion and Java Documentation express in percentage

Figure 23 shows a mix of both information. In general, we achieve a good level of information coverage of the entire system. There are two interesting points in this view: the former is visually the more evident, the mix of JavaDoc and discussions on *JGitText.java* result in a bluish, this mean that we have around the 50% of knowledge on it. The latter, the package *jgit.interna.storage.file* that has not too much documentation, it has a lot of discussions and therefore the information coverage is augmented.

5 Conclusion

We presented Cub8, a novel approach to extract information not strictly related to the code useful to improve the simplicity of understanding the system. It is the combination of an augmented visualization system using a 3d city metaphor with information that are retrieve from the web. We showed the possible information that it can visualize, it allows to analyze the information coverage as forum discussions or Java documentation individually, or it can combine multiple view to give to the developers a global idea about the system. Thanks to the possibility to change the metrics dynamically, we allow the developers to retrieve also some code disharmony. The user cases give to the developer an idea about how the system works and how is possible to highlight different problems of the system.

References

[1] A. Arwan, S. Rochimah, and R. J. Akbar. Source code retrieval on stackoverflow using lda. In *Information and Communication Technology (ICoICT), 2015 3rd International Conference on*, pages 295–299, May 2015.

[2] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*, VISSYM'04, pages 261–266, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[3] S. M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for informed decision making; from code to components. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 765–772, New York, NY, USA, 2002. ACM.

[4] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, June 1989.

[5] R. P. Gabriel. *Patterns of software*, volume 62. Oxford University Press New York, 1996.

[6] T. Guimaraes. Managing application program maintenance expenditures. *Commun. ACM*, 26(10):739–746, Oct. 1983.

[7] C. Knight and M. Munro. Virtual but visible software. In *Information Visualization, 2000. Proceedings. IEEE International Conference on*, pages 198–205, 2000.

[8] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[9] A. V. Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.

[10] Oracle. Java code conventions.

[11] L. Ponzanelli, A. Mocci, A. Bacchelli, and M. Lanza. Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software*, pages 343–352, Oct 2014.

[12] L. Ponzanelli, A. Mocci, and M. Lanza. Stormed: Stack overflow ready made data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, page to appear. ACM Press, 2015.

[13] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. ACM.

[14] C. Treude and M. P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 392–403, New York, NY, USA, 2016. ACM.

[15] Y. Tymchuk, A. Mocci, and M. Lanza. Code review: Veni, vidi, vici. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 151–160, March 2015.

[16] R. Wettel and M. Lanza. Program comprehension through software habitability. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 231–240, June 2007.

[17] R. Wettel and M. Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, June 2007.

[18] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *2008 15th Working Conference on Reverse Engineering*, pages 219–228, Oct 2008.

[19] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 155–164, New York, NY, USA, 2008. ACM.