Visualizing Code Bad Smells

Maen Hammad[1], Sabah Alsofriya[2]
Department of Software Engineering, The Hashemite University
Zarqa, Jordan

***Abstract*—Software visualization is an effective way to support human comprehension to large software systems. In software maintenance, most of the time is spent on understanding code in order to change it. This paper presents a visualization approach to help maintainers to locate and understand code bad smells. Software maintainers need to locate and understand these bad smells in order to remove them via code refactoring. Object oriented code elements are visualized as well as their bad smells if they exist. The proposed visualization shows classes as building and bad smell as letter avatars based on the initials of the names of bad smells. These avatars are shown as warning signs on the buildings. A framework is proposed to automatically analyze code to identify bad smells and to generate the proposed visualizations. The evaluation of the proposed visualizations showed they reduce the comprehension time needed to understand bad smells.**

Keywords—Software visualization; program comprehension; data modeling; bad smells

I. Introduction

Code bad smells are symptoms of poor design and implementation choices [1]. These bad smells have negative impact on the maintainability of the code. Badly written code is hard to understand, test and change. As a result, the changes of bugs increase. So, maintainers have to locate these smells in the code in order to remove them. Code smells are removed by a process called refactoring [1]. It is the process of rewriting the code to improve its internal structure without changing its external behavior.

The problem is how to identify these bad smells and locating code elements affected by these smells. Most of bad smells detecting tools reports results as formatted text. Developers have to go back to the source code and check the identified smell. They need to understand the cause of the smell in order to remove it. Understanding the smell with its causes in the code is essential to the refactoring process. The research question that we are trying to address is; how to represent or model code smells within its static code environment?

Program comprehension is essential to software maintenance activities. Most maintenance cost is spent on understanding the current status of the code and the system in general. Maintainers consume time and effort when interacting with large scale projects in order to understand them. Visualization is an effective way to ease the interaction process with code and hence support comprehension tasks. Our premise is that visualizing code smells with structural code elements supports maintenance activities by reducing comprehension time.

In this paper, we propose a visualization technique to model bad smells as well as their locations in the structural code environment. Classes are modeled as buildings. Each building consists of a number of floors that match the number of methods in the class. The number of class attributes, LOC for each method and its parameters are also visualized in the buildings. Bad smells are visualized as signs of letter avatars on the buildings. Each bad smell is modeled by a different avatar based on the initials of the smell's name. A framework is proposed to automatically analyze object oriented source code and to generate the proposed visualizations.

This paper presents our early work towards realizing the proposed framework as a complete visualization tool. We also illustrate the proposed visualizations and we show how they can be useful for program comprehension tasks. The main two research contributions of this paper are:

- Easy to understand visualizations to locate and identify bad smells in code.
- A framework to automatically analyze source code to generate the proposed visualizations.

This paper is organized as follows. Section 2 summarizes the main related research in the area. Section 3 presents the proposed visualizations. The proposed framework is detailed in Section 4. The evaluation of the proposed visualizations is presented in Section 5 followed by our conclusions and future work.

II. Related Work

There are many researches in the software visualization area that hard to cover in this paper. We focus on the most related work to ours that can be categorized into visualizing static code structure and visualizing bad smells.

A. *Visualizing Static Code Structure*

Ducasse and Lanza [2][3] presented a novel visualization for classes named class blueprint. It visualizes the internal structure of classes to support class understanding. A well-known 3D visualization approach, that model software as cities, is presented in [4][5][6]. The visualization maps the information about the source code in meaningful ways related to real cities. Another visualization approach for architecture and metrics of software systems as 3D software cities is presented in [7]. Panas et al. [8] proposes a 3D visualization metaphor to model software production cost as real cities. A reverse engineering environment called Rigi is presented in [9] to analyze, interactively explore, summarize, and document large projects. Marcus et al. [10] presented the sv3D framework for software visualization. The visualization is focused on source code and testing levels. Langelier et al. [11] proposed a visualization framework to supports and visualizes quality analysis of large software systems. Fittkau et al. [12] presented a live visualization approach to monitor traces for large software landscapes. In [13], Fittkau et al. presented ExplorViz to visualize the hierarchical abstractions of large software. The goal is supporting programming comprehension tasks. Merino et al. [14] introduced an interactive software visualization tool called CityVR. The tool implements the city metaphor technique using virtual reality to support comprehension tasks.

B. *Visualizing Code Smells*

The focus of this discussion is on visualizing bad smells not detecting them. Parnin and Goorg [15] presented visualizations to inspect bad coding patterns to assist developer finding relevant methods to inspect. Parnin et al. [16] proposed a catalogue of visualizations to assist reviewers to identify bad smells. They implemented a visualization tool called NOSEPRINTS. Murphy-Hill and Black [17] presented a bad smell detector called Stench Blossom. It detects and visualizes bad smells using the ambient view. The smell is shown with the source code. Mumtaz et al. [18] analyzed multivariate software metrics that link two visualization techniques, Parallel coordinates' plots and RadViz, for detecting outliers that may indicate for bad smells. Steinbeck [19] presented a visualization technique that consists of several Treemaps as a circle in order to integrate more bad smells visualizations. Carneiro et al. [20] presented a multiple views for code concern properties. These showed how these views support code smell detection.

In Summary, most of the related works in this area visualize either code elements or bad smells. We distinguished by modeling both; code and bad smells in the code with meaningful.

III. The Proposed Visualizations

The proposed visualization models classes as buildings with the following characteristics:

- Each building consists of a number of floors equals to the number of methods in the class. The first floor is not counted. The doors of the building are shown in this floor.
- The height of each floor represents the number of lines of code (LOC) in that method.
- The number of windows in each floor equals to the number of parameters for that method.
- The number of doors, shown in the first floor, of each building equals to the number of the data fields in that class.

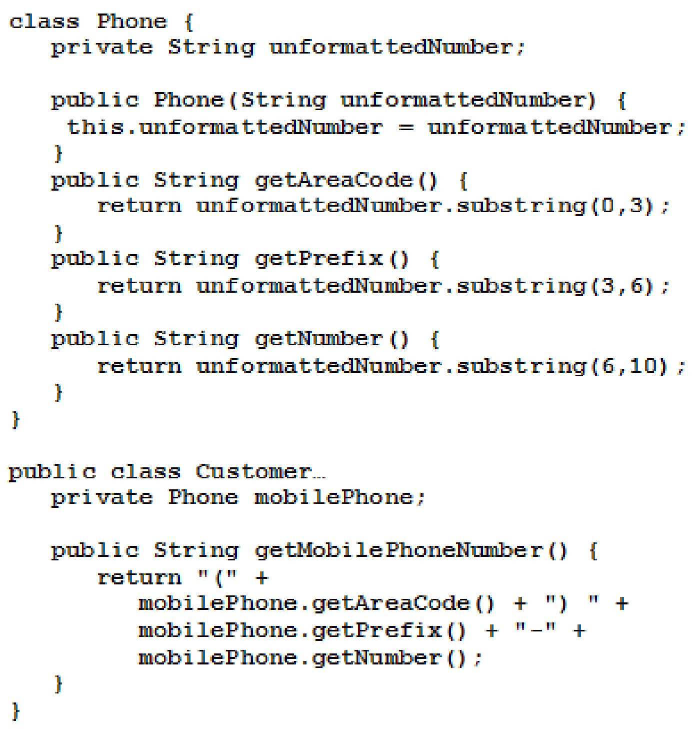
The bad smells are visualized as red signs on the buildings with letter avatars based on the initials with the following characteristics:

- Each avatar represents one code bad smell.
- Method related bad smells appears as avatars in the corresponding floor of that method.
- Class related bad smells appears as avatars on the roof of the building for that class

For example, Fig. 1 shows the source code for two classes; Phone and Customer from (from https://elearning.industriallogic.com). Phone class has four methods and one data field. The Customer class has one data field and one method. The two classes are modeled and visualized in Fig. 2. To illustrate the proposed visualizations, we generated the visualizations using the SketchUp (www.sketchup.com) tool. We used it to generate the visualizations in this paper based on the descriptions that we proposed.

The visualized building for the Phone class in Fig. 2 has four floors that correspond to the four methods in the class. The name of each method is shown in its corresponding floor. The first floor is not counted. We consider it level zero. It is always shown for all classes even if they have no methods. Buildings' doors are shown in level zero with the name of the class. The number of doors corresponds to the number of class's attributes. For the Phone class, one door is shown which models the single attribute of the class; unformattedNumber. The same applies for the visualized building for the Customer class. The class has one method (getMobilePhoneNumber) and one attribute (mobilePhone). It is modeled as a building with one floor and one door. The height of the first floor is five units with no windows. The floor models the getMobilePhoneNumber method that has five LOC and zero parameters.

The heights of all four floors are equal since all methods have two LOCs. The first floor has one window which means the first method has one parameter. Other methods have no parameter modeled by zero windows.



```
class Phone {
   private String unformattedNumber;

   public Phone(String unformattedNumber) {
    this.unformattedNumber = unformattedNumber;
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6,10);
   }
}

public class Customer…
   private Phone mobilePhone;

   public String getMobilePhoneNumber() {
      return "(" +
         mobilePhone.getAreaCode() + ") " +
         mobilePhone.getPrefix() + "-" +
         mobilePhone.getNumber();
   }
}
```

Fig. 1. Phone and Customer Classes.

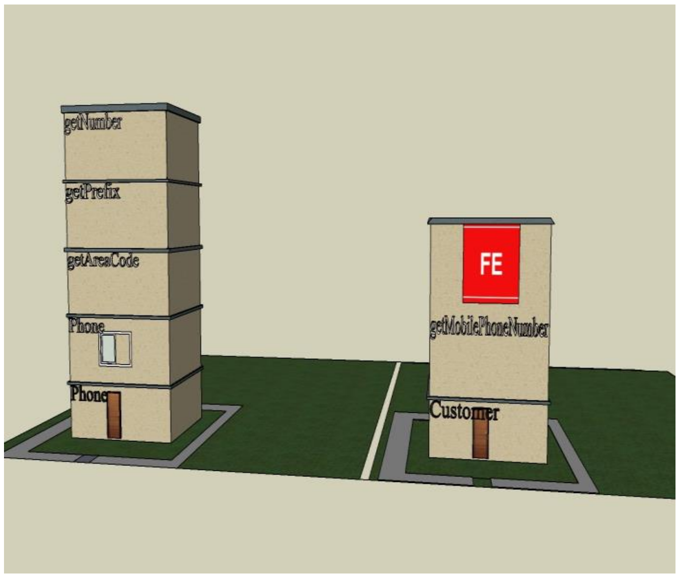


Fig. 2. The Visualization of the Phone and Customer Classes in Fig. 1.

Data class smell results when a class has data with only setters and getters. When a method is more interested in the data of other class, this method suffers from the feature envy smell. A method that has large number of parameter has a long parameters smell. Finally, long method bad smell results when the method has too many responsibilities and performs different tasks. Shotgun surgery results when a small change occur, many other changes have to be made in many classes and methods. The avatars that model these five bad smells are shown in Fig. 3. We tried to make the design of the avatars reflects the meaning of each bad smell. We used the initials of the bad smells names. The background is red to reflect the warning status of the smell. The letter avatars was designed using a tool from (http://google-avatar.herokuapp.com/)

Using initials has two main advantages. The first one is the readability and clarity. They are easy to read and distinguished among each other. This is essential in case large number of buildings with many smells is visualized. The second advantage is the supporting of adding more smells to be visualized. Letter avatars are easy to design and visualize. So, they support the extensibility of the approach to include more smells.

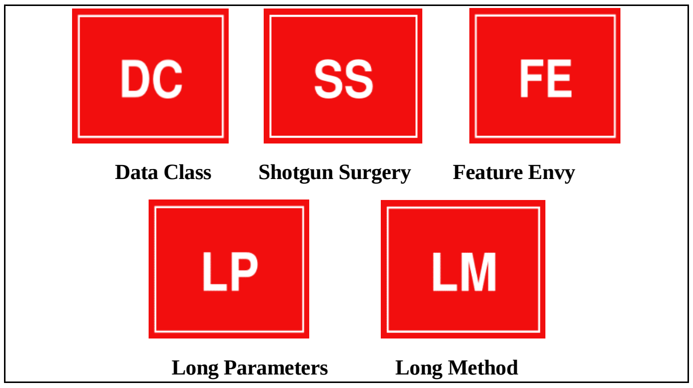


Fig. 3. The Proposed Avatars that used to Model the Bad Smells.

Method related avatars are visualized on the floors while class related smells are placed on the roofs. In Fig. 2, the Feature Envy avatar is shown on the floor that models the getMobilePhoneNumber method in the customer class. This is because this method is more interested in the data field of the Phone class and hence has the Feature Envy smell.

Fig. 4 visualizes the code shown in Fig. 5. The code in Fig. 5 shows a Java code example for a shotgun surgery smell (from http://javaonfly.blogspot.com). The visualized building shown in Fig. 4 models the class Account. It consists of four floors and three doors. The number of windows in each floor models the number of parameters for each method. The shotgun surgery avatar is shown on the roof of the building. The smell resulted from the account balance validation condition in the methods. In case this validation is updated, all methods have to be updated.

Fig. 6 shows two more visualizations examples for two different classes. The data class smell avatar is shown on the roof of the first building since it is a class related bad smell. The avatars of long parameter and long method smells are visualized on the second floor of the second building. This building has a very tall floor with many windows. This floor corresponds to the method that has the identified method related bad smells. The two avatars of these two smells are shown together.

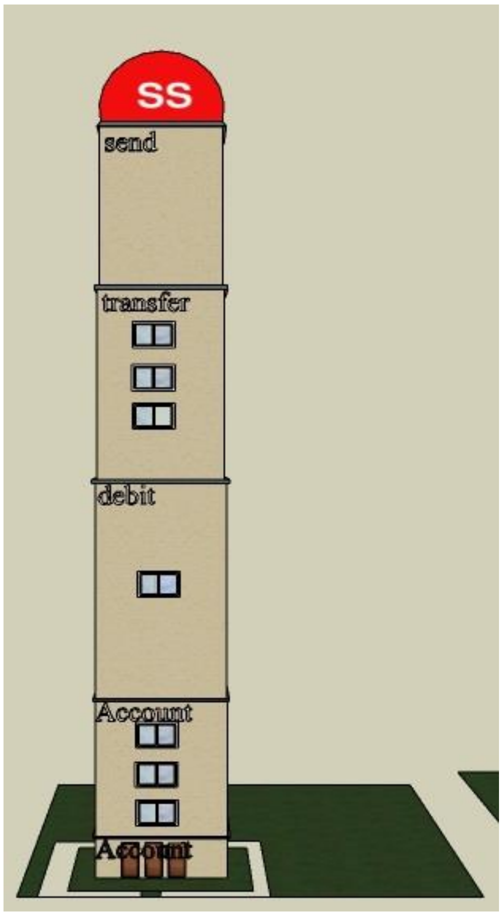


Fig. 4. The Visualization of the Account Class Shown in Fig. 5.

```
public class Account {
       private String type;
       private String accountNumber;
       private int amount;

public Account(String type, String accountNumber,
int amount){
              this.amount=amount;
              this.type=type;
              this.accountNumber=accountNumber;
       }
public void debit(int debit) throws Exception{
         if(amount <= 500){
          throw new Exception("Mininum balance
shuold be over 500");
              }
         amount = amount-debit;
         System.out.println("Now amount is" +
amount);
       }

public void transfer(Account from,Account to,int
cerditAmount) throws Exception{
           if(from.amount <= 500){
            throw new Exception("Mininum balance
shuold be over 500");
              }
              to.amount = amount+cerditAmount;
       }
public void sendWarningMessage(){
        if(amount <= 500){
         System.out.println("amount should be over
500");
              }
       }
}
```

Fig. 5. A Class with Shotgun Surgery Bad Smell.

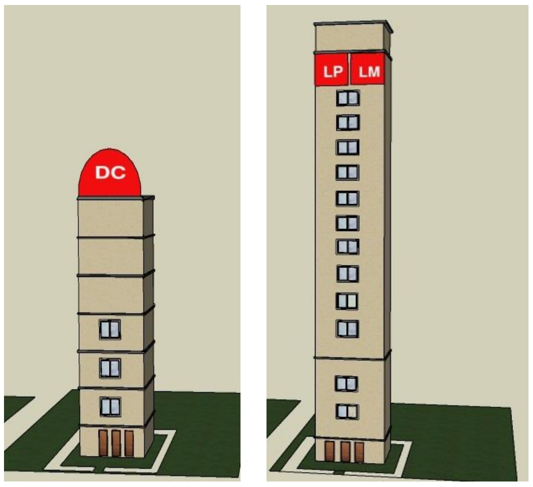


Fig. 6. Two Visualizations for two different Classes with Bad Smells.

It is important to mention that the floors of the building may have different heights because the LOC of the class methods are not necessarily equal. Also, the number of windows in all floors varies because it is based on the number of parameters for each method. The visualization of the LOC and the number of parameters for methods helps in preventing bad smells in advance. By browsing buildings, developers can quickly locate methods with potential long methods and long parameter smells. These methods have tall floors and many windows. These methods need to be carefully changed to avoid smells.

IV. PROPOSED FRAMEWORK

In this section, the proposed automated process for generating the views is detailed. This automated process can be realized by the proposed framework shown in Fig. 7. It shows the block diagram for the main components of the proposed framework. The process starts with the source code as an input to the framework. Then, the code is analyzed to identify bad smells. The same code is also parsed to extract the code elements that will be visualized. The identified bad smells with their locations in the extracted code elements are used to generate the data model. In the next step, the generated data model is used by a visualization tool to generate the proposed visualizations. The following subsections detail the components of the proposed framework.

A. Code Elements Extractor

This component is responsible for parsing the source code and extracting the needed code elements. The input could be a source file or a package of files. Each source file is transformed to the XML representation srcML [21]. This representation tags each code element with its syntactic information. The srcML representation can be automatically generated by using its tool that is available from (http://www.srcml.org/).

A set of XPath queries are applied on srcML to extract all needed code elements. The first extracted elements are classes. Then for each class all its methods and its data fields are extracted. Finally, the attributes of each method are extracted. Each extracted code element is given a unique label. The given label indicates the location of the code element. For example, the label of a data field in a specific class within a specific package is written as; package (Name) .class (Name) .field (Name). Another example, the label of a method with its (LOC) value is written as: package (Name). class (Name). method (Name). parameters (Names). LOC (number). All extracted labels are sent to the Data Model Generator component.

B. Bad Smell Identification

Code bad smells are identified using any specialized tool. There are many tools in the literature that can be utilized. More than one tool can be used in this component to identify a variety of bad smells. JDeodorant [22][23] and JFly [24] are our target tools to be used in this component. JDeodorant is an Eclipse plug-in tool that identifies code smells with refactoring suggestions. The bad smells identified by the tool are; Feature Envy, Type Checking, Long Method, God Class and Duplicated Code. JFly is also an Eclipse plug-in tool that detects bad smells from code changes as well as static code. The tool keeps track on code changes to identify nine bad smells. The identified code smells are; Inappropriate Intimacy, Data Class, Middle Man, Message Chain, Long Parameter, Lazy Class, Brain Method, Speculative Generality and Temporary Field.

The identified bad smells are forwarded to the Data Model Generator. Each bad smell is tagged with unique label. Each label represents the name of the bad smell and the location of the affected code element. For Example, the identified feature envy bad smell in a specific class within a specific package is labeled with the following label; smell (FeatureEnvy). Package (Name). class (Name). In case a long parameters code bad smell is detected in one method, its label is written as follows; smell (LongPara). Package (Name). class (Name). method (Name). parameters (Names).

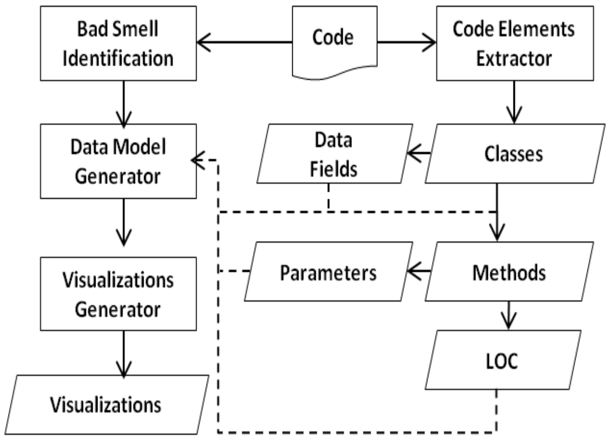


Fig. 7. The Proposed Framework for Generating the Proposed Visualizations.

C. Data Model Generator

Two types of labels are sent to the Data Model Generator component. The first type represents the extracted code elements and the other type represents the identified bad smells in code elements. The labels of code elements are used to generate the data model of the buildings. Then, the labels of bad smells are used to determine the avatars and their locations on the building.

The data model is generated automatically and mainly contains the following information:

- Locations and sizes of buildings that will be rendered on the screen. The base size of the buildings varies based on their numbers and the size of the screen.
- Specifications of buildings; the number of floors, the height of each floor, the number of floor's windows and the number of doors.
- Specifications of the avatars; type, number and location on the building.

The specifications are stored as data meta-model in a flexible XML format. The goal is to ease the rendering process using any visualization tool. For example, Fig. 8 shows a snapshot from the XML representation for the specification of one building. Each building has its own tag that corresponds to a single class. Within the building, more information is stored about its contents. This information includes the location on the screen, number of doors and the specification of each floor. Also, the avatar of the identified bad smell is stored as a tag. It is important to note that the width of the building is determined based on the number of class attributes. For example, a building that models a class with ten attributes has longer width to visualize ten adjacent doors than a building with only one door.

D. Visualizations Generator

Finally, the proposed visualizations are ready to be rendered. This component is responsible for rendering the generated data model on the screen. A tool can be developed using any programming language to draw the specifications stored in the data model. A specialized visualization tool can also be utilized to generate the visualizations.

```
<building>
      <location><x>200</x>  <y>220</y>
       <width>50</width>
      </location>
      <doors> 1 </doors>
       <floors>
       <floor> <height> 25</height>
        <windows> 10 < /windows>
       <avatar>long method</avatar> </floor>
       <floor> <height> 3 </height>
         <windows> 0 < /windows>
          <avatar>NONE</avatar> </floor>
       </floors>
</building>
```

Fig. 8. A Snapshot for the XML Representation of One Building.

We are working on developing a visualization tool to generate the views from the data model. The tool will be able to generate 3D visualizations for buildings. Zooming, localization and browsing are essential features that are under consideration. Developers will have the ability to search and locate a specific building that corresponds to a specific class. Also, they will be able to zoom in or out the buildings. The browsing feature helps developers to navigate through buildings in 3D environment. These features help developers to understand and handle large number of buildings that model large scale systems with many classes.

V. Evaluation

We need to evaluate the usefulness of the proposed visualizations in supporting comprehension tasks. So, we performed a controlled pilot experiment on software engineering undergraduate students. The goal of the experiment is to check how the proposed visualizations help in quickly understand and locate bad smells in code. The steps of the experiments were as follows:

1) Five java classes were carefully selected and implemented to have intentionally bad smells. The five bad smells that are under consideration in this paper where distributed over the five classes.

2) The data model of the classes was generated to model the classes based on the proposed visualizations.

3) The visualizations were generated using SketchUp and based on the data model of the classes.

4) Four software engineering students who are familiar with code bad smells were divided equally into two groups. All students have very good GPA rating.

5) The first group was given the five classes with textual report about the bad smells and their locations in these five classes.

6) The second group was given the same five classes and their visualizations with their bad smells.

7) Each student was asked to write down why the dedicated bad smell has occurred.

8) The answers of the two grouped were compared and the average completion time.

Table I shows the completion time in minutes for all four students on the two groups. The average completion time for the first group is 4.9 minutes. The first student needed five minutes to complete the task while the second student needed four minutes and 48 second (4.8 minutes). The second group, who used the visualizations, achieved better average time which is three minutes.

The comparison results between the two groups showed also that both groups answered the questions correctly. But the average completion time was different. The group who used the visualizations completed the task with about 40% less average time than the other group.

After completing the experiment, we also, asked the four students if they find the visualizations useful in understanding code smells. Three out the four students found it useful. The fourth student found the design of the avatars is not useful in modeling the bad smells.

TABLE I. THE COMPLETION TIME FOR THE SUBJECTS

Group	Student	Time in Minutes	Average
Without Visualizations	S1	5	4.9
	S2	4.8	
With Visualizations	S3	2.5	3
	S4	3.5	

VI. CONCLUSIONS AND FUTURE WORK

Software visualization support program comprehension tasks for maintainers. Useful visualizations have been proposed to help developers locate and identify bad smells. These visualizations show object oriented code elements with the types of identified bad smells in these code elements. A framework is presented to automatically analyze source code and generate the proposed visualizations. The framework is automated and can be extended to include more smells and visualize more code elements. The evaluation of the proposed visualizations showed their positive impact on understanding bad smells and their causes.

Our future work aims to completely implement the framework and realize it as a plug-in tool in an IDE as Eclipse. More avatars will be considered to cover more bad smells. We are also working on visualizing more code elements as relationships among classes, data types and methods invocations.

REFERENCES

[1] M.Fowler, Refactoring improving the design of existing code .Addison-Wesley, 1999.

[2] Ducasse, Stéphane, and Michele Lanza. "The class blueprint: visually supporting the understanding of glasses." IEEE Transactions on Software Engineering 31, no. 1 (2005): 75-90.

[3] Lanza, Michele, and Stéphane Ducasse. "A categorization of classes based on the visualization of their internal structure: the class blueprint." ACM SIGPLAN Notices 36, no. 11 (2001), pp. 300-311.

[4] Wettel, Richard, and Michele Lanza. "Visualizing software systems as cities." In 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 92-99. 2007.

[5] Wettel, Richard, and Michele Lanza. "Codecity: 3d visualization of large-scale software." In Companion of the 30th international conference on Software engineering, pp. 921-922. 2008.

[6] Wettel, Richard, Michele Lanza, and Romain Robbes. "Software systems as cities: A controlled experiment." In 2011 33rd International Conference on Software Engineering (ICSE'11), pp. 551-560. 2011.

[7] Alam, Sazzadul, and Philippe Dugerdil. "Evospaces visualization tool: Exploring software architecture in 3d." In 14th Working Conference on Reverse Engineering (WCRE 2007), pp. 269-270. 2007.

[8] Panas, Thomas, Rebecca Berrigan, and John Grundy. "A 3d metaphor for software production visualization." In Proceedings on Seventh International Conference on Information Visualization (IV 2003), pp. 314-319. 2003.

[9] Kienle, Holger M., and Hausi A. Müller. "Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation." Science of Computer Programming 75, no. 4 (2010), pp. 247-263.

[10] Marcus, Andrian, Louis Feng, and Jonathan I. Maletic. "3D representations for software visualization." In Proceedings of the 2003 ACM symposium on Software visualization, p. 27. 2003.

[11] Langelier, Guillaume, Houari Sahraoui, and Pierre Poulin. "Visualization-based analysis of quality for large-scale software systems." In Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering (ASE'05), pp. 214-223, 2005.

[12] Fittkau, Florian, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: The ExplorViz approach." In 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp. 1-4. 2013.

[13] Fittkau, Florian, Alexander Krause, and Wilhelm Hasselbring. "Software landscape and application visualization for system comprehension with ExplorViz." Information and software technology 87 (2017).pp. 259-277.

[14] Merino, Leonel, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. "CityVR: Gameful software visualization." In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 633-637, 2017.

[15] Parnin, Chris, and Carsten Görg. "Lightweight visualizations for inspecting code smells." In Proceedings of the 2006 ACM symposium on Software visualization, pp. 171-172, 2006.

[16] Parnin, Chris, Carsten Görg, and Ogechi Nnadi. "A catalogue of lightweight visualizations to support code smell inspection." In Proceedings of the 4th ACM symposium on Software visualization, pp. 77-86. 2008.

[17] Murphy-Hill, Emerson, and Andrew P. Black. "An interactive ambient visualization for code smells." In Proceedings of the 5th international symposium on Software visualization, pp. 5-14. 2010.

[18] H. Mumtaz, F. Beck and D. Weiskopf, "Detecting Bad Smells in Software Systems with Linked Multivariate Visualizations," In 2018 IEEE Working Conference on Software Visualization (VISSOFT'18), pp. 12-20, 2018.

[19] Steinbeck, Marcel. "An arc-based approach for visualization of code smells." In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17), pp. 397-401. 2017.

[20] Carneiro, Glauco de F., Marcos Silva, Leandra Mara, Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, and Manoel Mendonca. "Identifying code smells with multiple concern views." In 2010 Brazilian Symposium on Software Engineering, pp. 128-137, 2010.

[21] Collard, M. L. , Kagdi H. H., Maletic, J. I., "An XML-based lightweight C++ fact extractor," Proc. of 11th IEEE International Workshop on Program Comprehension (IWPC'03), pp. 134-143, 2003.

[22] Fokaefs, Marios, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. "JDeodorant: identification and application of extract class refactorings." In 2011 33rd International Conference on Software Engineering (ICSE'11), pp. 1037-1039, 2011.

[23] Mazinanian, Davood, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. "JDeodorant: clone refactoring." In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp. 613-616. 2016.

[24] Maen Hammad, Asma Labadi, "Automatic Detection of Bad Smells from Code Changes", International Review on Computers and Software, Vol. 11, No. 11, pp. 1016-1027, 2016