Code Smell Visualization

Kimberly Dextras-Romagnino, Nikolaos Tsantalis

Department of Computer Science and Software Engineering

Why Do We Need Visualization?

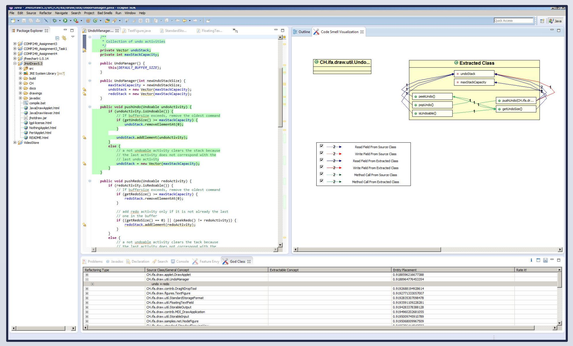
Research and practice have shown that the cost of performing maintenance activities highly depends on the underlying design quality of the software systems. In the past, several techniques have been developed for the detection of design problems as a means to support the improvement of design quality in software systems. However, most of these techniques lack the ability to communicate the detected problems to the developers in a comprehensible and effective way. This is one of the reasons justifying the slow and hesitant adoption of preventive maintenance (i.e., maintenance activities aiming to improve future maintainability) as a practice in the software industry.

In this poster, we demonstrate two code smell visualizations for the **Feature Envy** and **God Class** design problems. The visualizations have been integrated in JDeodorant Eclipse plug-in, a code smell detection and refactoring tool.

How Does it Work?

After installing JDeodorant from the website (see below) , you must select the God Class or Feature Envy option from the “Bad Smells” tab. After selecting which class or package you wish to refactor, click on the “Identify Bad Smells” button. A list of possible refactoring suggestions will be displayed in a table at the bottom of your screen.

The “Code Smell Visualization” Tab opens when you double-click on one of the refactoring suggestions. This tab is easily moveable to anywhere you want in Eclipse. This allows you to view the visualization and the actual code at the same time which makes for a better understanding of what is happening. This is shown in the diagram below:



To learn more about Jdeodorant and to see the Code Smell Visualization for yourself, please visit and download Jdeodorant at the following website :

http://www.jdeodorant.com/

jDeodorant
when quality matters the most...

ACKNOWLEDGEMENTS

NSERC CRSNG

FEATURE ENVY

Feature Envy occurs when a method references another class through methods and fields more often than it references its own class.

Solution: Move the method to the class that it is most envious of, passing any parameters the new method requires

The Visualization

The Visualization shows the **Source Class** on the left which originally contains the method to be extracted (shown with a white background). On the right, is the **Target Class** where the method is going to be moved to. In the middle, is the (potentially) **Moved Method**.

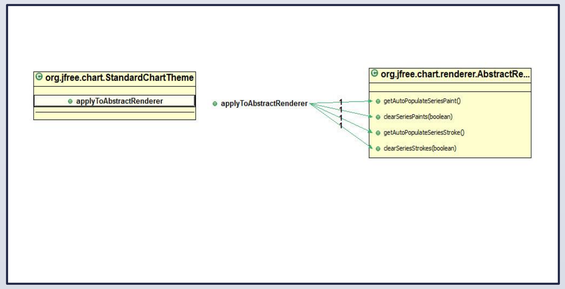
The **Moved Method** has connections to all the methods and fields it references. The number on the connections indicates the number of times this method or field was accessed. There is a legend to indicate the different types of connections

How does it help?

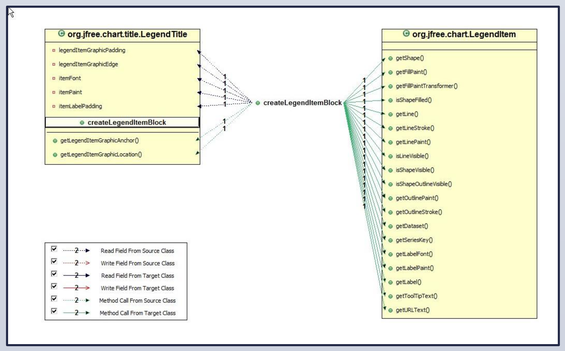
If the **Moved Method** has the majority of its connections to the **Target Class** in comparison to the **Source Class**, then it is a good refactoring suggestion.

Examples

The following example illustrates an excellent refactoring suggestion: the **Moved method** makes no reference to anything in the **Source Class** but calls four different methods from the **Target Class**



The following example is also a good refactoring suggestion: The majority of the calls made by the **Moved Method** are to methods in the **Target Class.**



GOD CLASS

God Class usually violates the single responsibility principle and controls a large number of objects implementing different functionalities.

Solution: Extract all the methods and fields which are related to a specific functionality into a separate Class

The Visualization

The Visualization shows the **Source Class** on the left which originally contained all of the extracted methods and fields. On the right is the **Extracted Class** which now contains all of the extracted methods and fields.

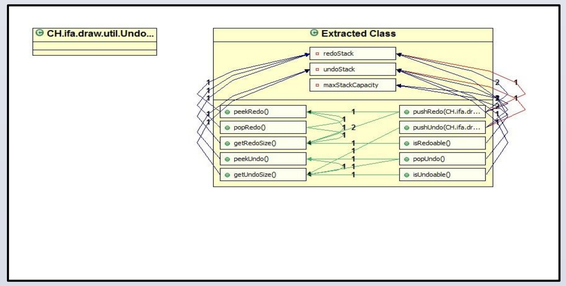
The extracted methods have connections to all the methods and fields they reference. The number on the connections indicates the number of times this method or field was accessed. There is a legend to indicate the different types of connections.

How does it help?

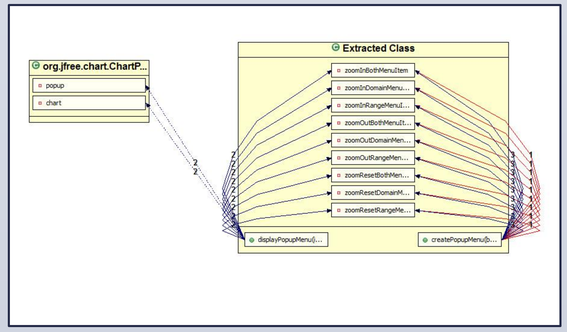
If the extracted methods have the majority of its connections to the **Extracted Class** in comparison to the **Source Class**, then it is a good refactoring suggestion.

Examples

The following example illustrates an excellent refactoring suggestion: the methods and fields in the **Extracted Class** are tightly connected and make no reference to any field or method in the **Source Class**.



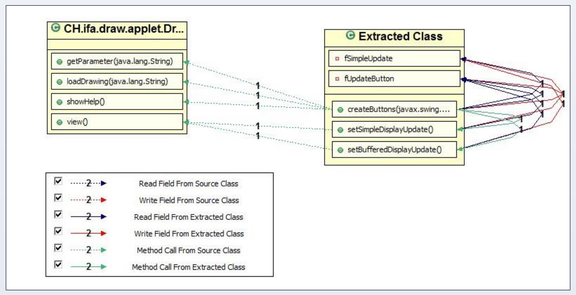
The following example is also a good refactoring suggestion: The majority of the calls made by the extracted methods are to methods and fields in the **Extracted Class.**

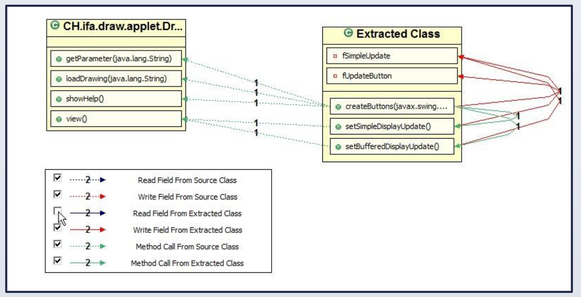


FEATURES

- Select Connections

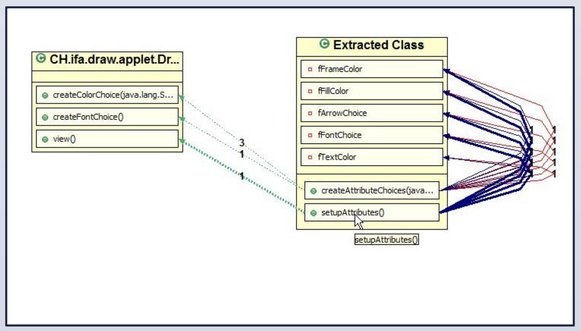
By using the checkbox, you can make the different types of connections appear and disappear allowing for a clearer picture





- Highlight Connections

By moving the mouse pointer over a method or field, you can highlight the outgoing connections from that specific entity



- Tooltips

A tooltip displaying the full name of the method or field appears when you move the mouse pointer over it