Phase Analysis of JHotDraw

# Design and implementation of the analyzer

The design process of the analyzer started by choosing the best tool for the requirements and goals that needed to be satisfied. The chosen tool was the AspectJ plugin for Eclipse. The plugin was developed by Eclipse and is an extension that provides the possibility of creating aspect-oriented projects. The easy installation and flawless integration also influenced the choice of AspectJ. The project template was created using this extension. The structure of the template enabled the analysis process from the development environment.

The analyzer needed to output multiple CSV files as a result of recording the execution of a generic drawing application. The need for logging was clear from the start of the project. The aspect oriented method enables the developers to inject code in parts of the running code that match a certain pattern.

Using aspect oriented methods for logging is a proven technique. It allows for a dynamic and relatively easy logging configurations. The hard part of the implementation of the logger is getting the pointcut patterns right. The three files that are created when running the tool are called: depth – this file contains the name of the executing method and the current stack depth, numberOfCalls - this file contains the name of each method and counter representing the number of times the method was called during the application runtime, the last file is named numberOfCallsFromParent and contains a list of string-integer pairs where the string is composed by the concatenation of two method names and the number represent the number of times the first method called the second method while the application was tracked. The format of these output files is CSV (Comma Separated Values). This format can be easily imported in Excel. After importing in Excel I created three line chart templates that were used to generate the charts displaying the trace information. The creation of templates was justified by the amount of times this generation needed to be executed. The visual aid was useful during the analysis phase.

The CSV files were created using a simple FileWriter. The writer writes the entities line by line by adding a comma between the value pair and appending an end line character after each pair. The file containing the stack depth for each method call is modified after each method call in order to save memory. The update needs to be called after the call depth information is available. The files containing the number of calls by parent and the number of calls for each method are created once the analyzed application has stopped. To achieve this I needed to hold all the information in memory. This is doable for most medium application since the number of methods wasn’t that high. The chosen storage method was HashMap. The map key for the first structure was the pair of method names (caller - callee) separated by a hyphen. The map key for the second structure was the name of the method. Every time a method is called both structures need to be updated. If the map does not contain the key of the current entity we insert it using 1 as the value. If the map already contains the key I increment the counter and insert it in the map. The stop event can be detected listening for the method called stop.

In order to catch each method call we needed to define the event in the method lifecycle that we needed to listen for. The most important locations in the application execution that were considered were the steps before the execution of a method and the steps after the execution of a method.

The solution implementation consist of an aspect file. The file is named TrackExecution. The aspect contains handlers for each of the important events.

The aspect oriented method looks at the pattern defined by the programmer and injects code in the application that will be executed. This allowed me to insert my code that tracks and records the details of the execution.

The number of fields declared in the aspect were kept at a minimum. The most important attributes are the names of the log files, the maps containing the data that can be written only when the execution ends and a static variable tracking the call depth.

The name of the execution method can retrieved in the before and after sections by calling the methods of the thisJointPointStaticPart available in this context. This enabled me to obtain the method signature. For the creation of the log file contain the number of calls by parent for each method I needed to find the caller for each execution step. This was done by reading the name of the previous method from the stack of the current thread.

# Description of the rendering process

The tool needed to analyze the execution of the JHotDraw application when a user draws two circles and connects them using the connection tool. This scenario was recreated multiple times to characterize the process and to provide some insight into the architecture of the application.

The result of this scenario was the output of three files. The first contains the name of the executing method and the current stack depth, the second file contains the name of each method and counter representing the number of times the method was called during the application runtime, the last file contains a list of string-integer pairs where the string is composed by the concatenation of two method names and the number represent the number of times the first method called the second method while the application was tracked.

Analyzing the log file containing the number of calls for each method we can observer that the most frequent methods called are contained in the Java core, usually the ones implemented directly in the native code in the JVM. A good example for this is the hashCode function that is called 2700+ times. The most frequently called method are: get, getTarget, hashCode, propertyChange and getEditor.

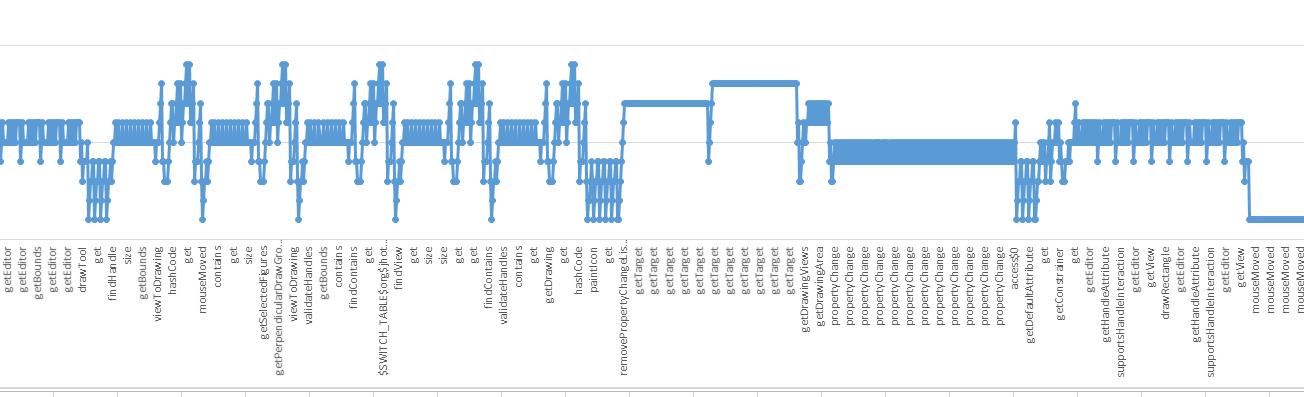
The high number of getter and method calls indicate that the application was written respecting the encapsulation principle, where the class attributes are declared as private and are retrieved and modified using the getter and setter methods defined for each of these attributes. This paradigm aims to reduce coupling and helps hide implementation details each class from other classes.

The calling frequency for the methods getView, getActiveView and draw suggest a model view controller architectural pattern. This pattern is very common in application that provide a graphical user interface. The user modifies the model object through the view and uses the controller to refresh the view. This technique provides a clean segregation of responsibilities. It keeps the data separate from the graphical components and application logic.

The methods representative for the observer pattern are hard to find, they are called only a few times. This mean that application was created without the use of this pattern or the methods were renamed.

The analysis of the logs clearly indicate that the tracking of the user’s mouse movement is achieved with the mouseMoved method that is a mouse listener that handle the event of the mouse moving within the frame of the application.

The connection tool is rendered by using the above mention function to track the user movement and calls the methods updateCursor and draw.



The above section is an extract from the rendering of the connection tool. The method findHandle is used to the start and end points for the connection. The validateHandle method is used to make sure the handle is correct. The method getTarget is used multiple times to bubble to the root of the target element and the call of propertyChange updates the target and the drawRectangle method takes care of the rendering.

When selecting the connection tool feature a start point is created and the next click fired in an existing element will create an end point. The final step of the rendering is drawing the line between the centers of the two elements. Other common methods called during this process: addNode, getNodeCount, isConnectable. The method isConnectable is extensively called to check if the user has click on the next figure for the connection to be complete.

# Trace graphs