Aspect-Oriented Refactoring of Graph Visualization Tool: CHISIO

Esat Belviranlı, Celal Çığır, Alptuğ Dilek

Department of Computer Engineering, Bilkent University
06580, Bilkent, Ankara

{belviran, cigir, alptug}@cs.bilkent.edu.tr

Abstract

Graphs are data models and widely used in many areas from networking to biology to computer science. Visualization, interactive editing ability and layout of graphs are critical issues when analyzing the underlying relational information. There are many commercial and non-commercial graph visualization tools. However, overall support for compound or hierarchically organized graph representations is very limited. CHISIO is an open-source editing and layout framework for compound graphs. CHISIO is developed as a free, easy-to-use and powerful academic graph visualization tool, supporting various automatic layout algorithms. However, since CHISIO was designed and implemented with respect to the object-oriented software approach, cross-cutting concerns were not considered during any phases of the development of the software. In this paper, we propose an aspect-oriented refactoring of CHISIO in which cross-cutting concerns are implemented as aspects.

Keywords

Information visualization, graph editing, software system, graph editor, compound graphs, Aspect-oriented programming, separation of concerns.

1. Introduction

Graphs are simply a set of vertices or nodes plus a set of edges that connect these nodes to each other. They are useful data structures to capture the relations (represented by edges) between the objects (represented by nodes). From network to data flow charts, from bioinformatics to computer science, many real world problems can be modeled and simulated by using this simple but powerful data structure.

Compound graphs are child graphs which can contain inner nodes and edges. They are special type of nodes meaning that: a node can contain nested child nodes inside and also a node can have incident edges on it. By embedding a compound node into another compound node, multiple nested compound graphs can be constructed.

Graph visualization is a research area for producing visual representations of any data using graphs by deriving (or using) the topological information lying under the model. Understandability of these visual drawings must be high for easy interpretation and analysis. Geometrical information of the produced graph holds important measures for better graph visualization. Locations, sizes and relative positions of the nodes are part of the geometrical information and they should be adjusted either manually or automatically in order to produce an understandable and clear graph. This operation is called 'graph layout'. Many complex graphs can be laid out in seconds using automatic graph layouts.

Each application has its own type of data representation style. Therefore, different layout styles should be applied for different applications. For instance, hierarchical layout produces best result for family tree applications. This is due to the fact that ancestry information can easily be represented by the levels generated by the hierarchical layout.

This study is focused on refactoring CHISIO, which was designed and implemented as a general purpose graph visualization and editing tool for proper creation, layout and modification of graphs. It was developed using object oriented software development approaches. In addition to existing capabilities of existing graph visualization software, CHISIO includes compound graph visualization support among with different styles of layouts that can work on compound graphs.

Although CHISIO is a mature software that strictly follows commonly accepted OOSD approaches, we have faced with extensibility, reusability and maintainability problems while adding domain specific extensions to CHISIO. For instance, in trying to extend CHISIO to model biological pathways, we needed to extend not only the graph model in CHISIO; but also many other classes in other packages. Moreover, some of the new features (such as Layout Profiling) we plan to add to CHISIO consist of inherently crosscutting-concerns. We discovered that most of these problems can be solved with aspect oriented approach. As a result, we decided to refactor CHISIO using aspect oriented programming techniques.

In the rest of this paper, we first explain the object oriented design of CHISIO. Then we address the cross cutting concerns and identify the aspects for these concerns. Later on, we give more details on how these aspects are implemented by including point-cuts and advices. We finally discuss the issues we had and what we have learned during our experience with AOP.

The remainder of this paper is organized as follows: Section 2 describes the OO design of CHISIO problem case. Section 3 identifies cross cutting concerns in the software and mentions corresponding aspects. Section 4 deals with Aspect-Oriented design and implementation issues we had during development. Related work can be found in Section 5. Further discussion is placed in section 6. We finally provide our conclusion in section 7.

2. Object-Oriented Design Overview of CHISIO

CHISIO is one of the major graph visualization and editing tools which supports compound graphs and various layouts. It is a software tool which is developed strictly following object oriented software development criteria. Figure 1, illustrates a running screenshot of CHISIO where compound graphs and supported layout styles can be observed.

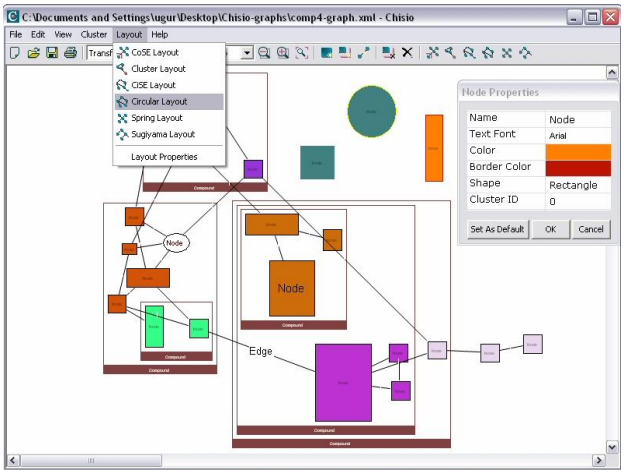

Figure 1. Overview of CHISIO

CHISIO is a comprehensive software which reuses industry-grade frameworks and libraries. It is mainly built on Model-View-Controller architecture which is a very well known structure especially used by visualization software. The design of CHISIO is depends on an extendible model that is developed using several OO design patterns. This section will give information about the architecture and the design of CHISIO in order to give a clearer idea about the software before discussing about the aspect oriented refactoring we have performed.

2.1. The Architecture of CHISIO

CHISIO, in general, is based on the Model-View-Controller architecture. This architecture is required by GEF [14] on which CHISIO is mainly dependent for graph editing and drawing operations. MVC architecture suggests separation of visualization and the data model and the interaction between these components is handled by a third module named controller. This ternary relation provides modification of the data model without any need for change in the user interface. The synchronization between model and view is carried out by controller.

In MVC, *model* is the concept to be represented; visualization of the model is the *view*. *Controller* is the bridge reflecting *model* changes to the *view*.

- **Model:** Model holds the data that is to be displayed. It also assesses the structure of the data and provides update mechanisms for modifications to data. Model does neither know how the data is visually represented nor responsible for updating the view. In CHISIO, model part is the graph structure. CHISIO model will be described in detail in Section 2.2.
- **View:** The view is the user interface which is used for visually presenting the model. Similar to the model, view does not also know about the underlying data structure and logic. Changes to the view are consequences of model updates. *Figure*s in GEF are the correspondents of the view in CHISIO.
- **Controller:** Controller is responsible for creating and updating the view according to model changes. Whenever a change in the data occurs, controllers are notified in order to redraw the view accordingly. *EditPart*s are the controllers in GEF.

Figure 2 illustrates how MVC pattern is applied in GEF.

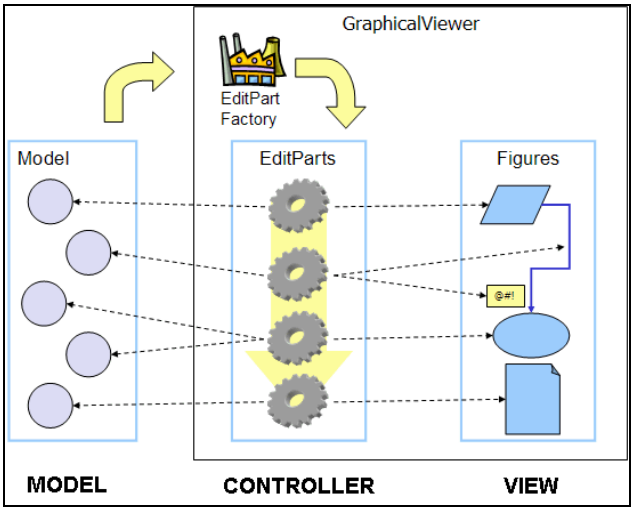

Figure 2. Model View Controller pattern applied to GEF.

2.2. Design Details of CHISIO

In CHISIO, graphs are composed of nodes, edges and compound nodes. Each of these three types extends from the type graph object.

Nodes are customizable building blocks of the graph where a user can change several properties such as background color, border color and font size. Users also are able to assign a cluster number to each node for further use in clustered layouts.

Edges have source and target nodes and each node holds a list of incident edges on it. Edges also have customizable properties.

Compound nodes are special type of nodes which can hold a list of child nodes. Since they can act as simple nodes, they can be nested into each other resulting in a composition relation in the model hierarchy as the situation can be observed in Figure 3.

Common properties such as text and color are held in the graph object. Each graph object is responsible for informing the changes to its associated controller via a property change listener interface. When the controller receives the property change event, it notifies the view to update itself.

Besides figure updating, controller has other responsibilities such as determining capabilities of the associated model. For example, installing an XYLayoutPolicy to the EditPart(controller) corresponding to a compound node will enable insertion of child nodes into specific x and y coordinates.

Whenever a user wants to add a custom graph object where some restrictions should exist on editing, the appropriate policy should be created and installed to the corresponding controller.

CHISIO is able to load graphs in several widely known graph formats such as GraphML[10] and also save edited graphs in the same format.

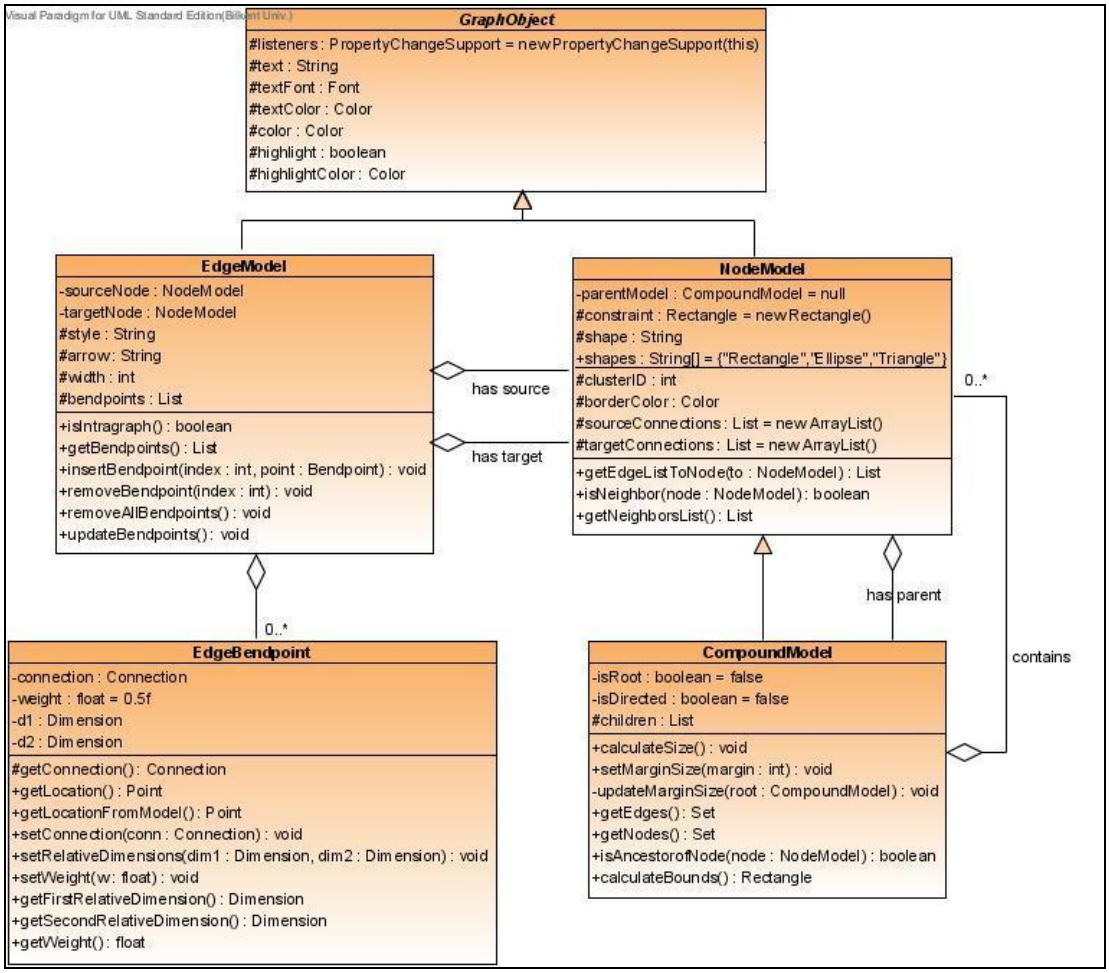

Figure 3. A UML Class Diagram illustrating the compound graph model used in CHISIO.

2.3. Applied Design Patterns

In order to simplify most operations and reduce coupling among software components, some design patterns were applied during the design of CHISIO. The new code introduced during this aspect oriented review is also designed and implemented in the same manner. These are as follows:

2.3.1. Model-View-Controller (MVC) Pattern

This pattern is explained in detail in Section 2.1.

2.3.2. The Composite Pattern

"Composite pattern allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite pattern is to compose objects in to tree structures to represent part-whole hierarchies. Composite pattern lets clients treat individual objects and compositions uniformly" [15].

CHISIO model is inherently based on a composite structure. Compound nodes, hold nodes as their children, which ends up the model to have nested compound nodes when they are used recursively. This structure can easily be observed in Figure 4.

2.3.3. Command Pattern

"The Command design pattern encapsulates the concept of the command into an object. The issuer holds a reference to the command object rather than to the recipient. The issuer sends the command to the command object by executing a specific method on it. The command object is then responsible for dispatching the command to a specific recipient to get the job done." [16].

Commands are one of the core structures of GEF. All operations on *model* are performed via commands. They are executed on a command stack which provides the ability for undo and redo. GEF provides hooks for creating and attaching required commands when some predefined cases occur. In CHISIO such commands are implemented according to custom needs of the application.

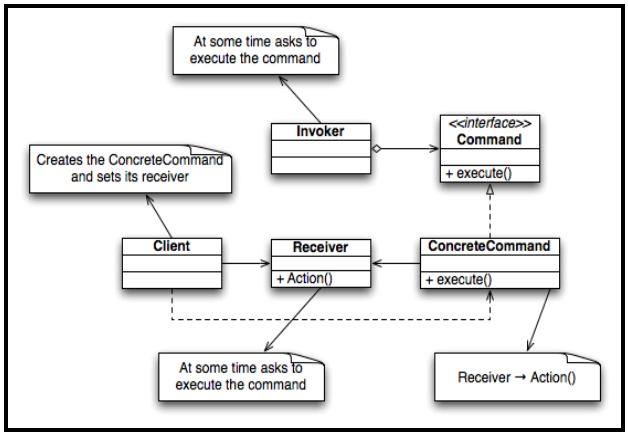

Figure 4. Command Design Pattern.

Once all undo and redo methods of the commands are implemented properly, undoing last operation is simply popping the latest command on the command stack and vice versa for redo.

2.3.4. Singleton Pattern

Singleton pattern is used where only one single instance of a class is needed or required during execution of a program. Constructor of the class should be public and the single instance of the class might be accessed via a static method which performs a lazy initialization.

Many dialogs in CHISIO can be grouped according to their similarity where the only difference is the texts they display. There is no need to create separate instances for such groups of dialogs. Changing related text is enough for expressing the intended message and getting the response.

For example, the new dialog introduced for setting command execution constraints is a singleton object and the same dialog instance is used during the execution of the program.

3. Aspect Identification

When we analyzed the OO design of the system, we have realized that some concerns were crosscutting through different modules. We were not able to localize them in a single component even by means of applying design patterns. This is because, these concerns were inherently crosscutting and scattered among several modules. Moreover, we realized that we need to introduce new cross-cutting concerns when we try to model specific domains such as a networking tool, in which not every node type can be incident to every other node type, if we stick to OO design. Yet, as mentioned in the introduction part, layout profiling feature that is thought to be integrated into the system is inherently crosscutting.

Crosscutting concerns which we have recognized in our system can be addressed with the following aspects:

As production aspects, we have identified the following:

- Command Execution Constraints
- Model Property Settings

Whereas development aspects we have identified are as listed below:

- Layout Profiling
- Action Tracking

3.1. Production Aspects

The production aspects we determined are thought to solve some of the existing understandability and extendibility problems CHISIO contains.

3.1.1. Command Execution Constraints

In OO implementation of CHISIO, there are generic node, edge and compound types. However, when a developer decides to extend CHISIO, specific graph object types must be introduced. It is generally the case that, a specific edge type can only attach to some specific node types or a specific node can only be child of some specific compound types. For instance, in order to model a biological pathway as a graph, new node types (such as Protein, DNA, RNA, etc.), new edge types (such as Activation, Protein-Protein-Interaction, Inhibition, etc.), new compound types (such as Abstraction, Compartment, etc.) must be introduced via extending the current graph model. However, you also need to extend from other classes of CHISIO other than the model package.

The connection of a node with an edge or an insertion of a node to a compound in CHISIO is achieved via executing instances of the class called "*Command*". Since CHISIO is a general graph visualization tool, those commands do not consider any domain specific constraints. In order to introduce those constraints many commands in the software must be extended and their execution methods must be overridden. However, this situation results in scattering of those constraints into many classes. In order to tackle this problem, we introduce a new aspect called CommandConstraints. So, when new graph object types with different constraints (node-edge attachment constraint, or compound-node containment constraint) are to be introduced, the Command classes are no longer to be extended. The only change will occur in the specified aspect.

Figure 5 illustrates the class hierarchy that should have been adapted if an only OO approach was chosen. Thanks to the newly introduced aspect, there is no need to override execute methods of different methods. Thus there will be no change in the existing class hierarchy of the commands.

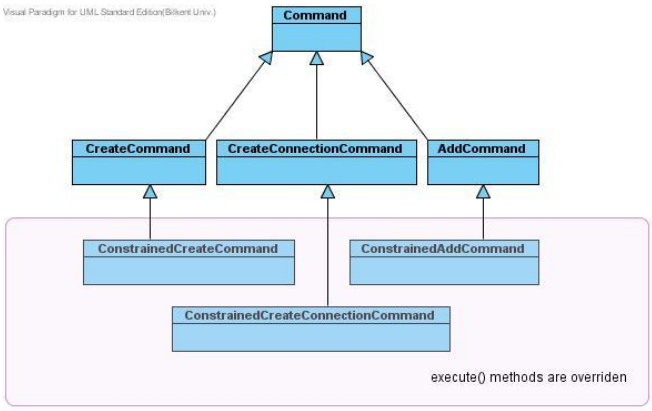

Figure 5. Prevented OO Command Hierarchy.

3.1.2. Model Property Settings

As mentioned above, CHISIO is built upon GEF, which is a framework based on MVC pattern. Any changes in any properties of the model object must be propagated to the controller. However, this results in scattering of the controller notification concern in the entire graph model classes. When a new property is to be introduced to a graph object class, the notification code must also be written for the new field. In order to tackle this problem, we decided to introduce a new aspect called *ModelPropertySettings*. In fact, this is not just a solution for CHISIO, but also a general aspect oriented solution for the separation of notification concern from the actual modeling concern. The graph model classes in CHISIO now do not include code related with the notification of controller part. If a new field, for which the controllers must be notified when a change in the field occurs, a new point-cut must be included to the aspect if it is not already captured.

3.2. Development Aspects

Development aspects handle the crosscutting-concerns that are faced with during the development of software. They are not to be included in the final product. However, developers should pay more attention to these aspects since the cross-cutting concerns they solve, make the software difficult to understand and modify. During the software development phase, those concerns lead to consume considerable amount of time, which is a very valuable resource for developers.

3.2.1. Layout Profiling

CHISIO is great software that provides integration of new layout algorithms easily by extending the graph structure supported. It also supports different layout algorithms such as CoSE, CiSE, Spring, Circular, Sugiyama, etc. layout algorithms. While developing layout algorithms, developer needs to fulfill many requirements such as decreasing method call count and detecting and solving method hotspots. Thus layout profiling should be included in the developer version of this tool. Unfortunately, current implementation of CHISIO lacks this capability. Implementing the profiling will give us the chance to detect and correct bottlenecks of the layout algorithms.

3.2.2. Action Tracking

Nearly, all graph model changes in CHISIO occur as results of the user interactions. Those interactions are called 'actions' in GEF terminology. After an action is triggered, the control flows through internals of the framework. As a result, it is really difficult for a newbie developer to understand and follow the mechanism of GEF. Therefore, we decided to implement tracking functionality for the actions to ease the debugging and testing of the system.

4. Aspect-Oriented Programming

For implementing the identified aspects, we used Eclipse plug-in of AspectJ for most of the aspects. However, we implemented one of the aspects by using JBoss AOP library in order to benefit from the dynamic AOP feature (addition and removal of aspects in the runtime dynamically).

4.1. Command Execution Constraints

In order to add domain specific constraints to CHISIO Commands, we need to capture the execution of the commands for which constraint checks must be performed. Thanks to the OO design of the command package in CHISIO, we can capture the execution of the methods by writing a somewhat general point-cut. We use around advice for that point-cut, in order not to execute the command in case a possible constraint check fails. In order to test the usefulness of this aspect, we introduced new edge, node and compound node types, which extend from the base graph model of CHISIO. As explained in Command Execution Constraints section, there is no need to introduce new command types. We implemented a singleton dialog as depicted in Figure 6 to add/remove constraints (rules) and to enable/disable the constraint checking in the runtime via using hot deployment of JBoss.

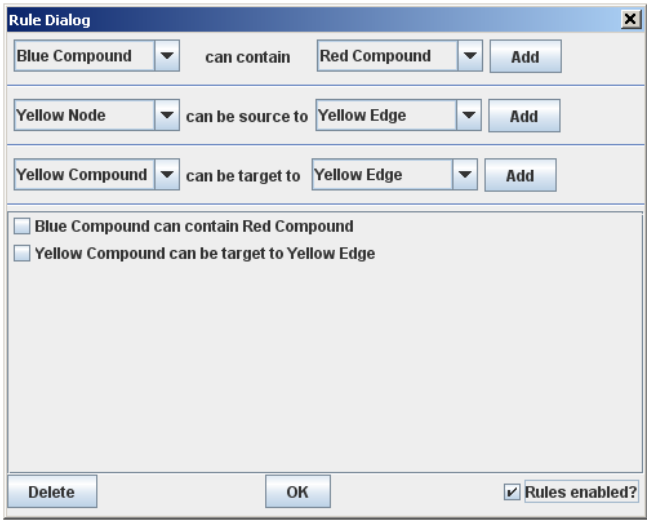

Figure 6. Rule Creation Dialog

Here is the pointcut for this aspect written in AspectJ and JBoss respectively:

```
pointcut commandConst(Command command):
   target(command) &&
   (execution(void ChsCompoundCommand.execute()) ||
   execution(void CreateCommand.execute()) ||
   execution(void CreateConnectionCommand.execute()));
```

Figure 7. Command constraint pointcut for AspectJ

```
execution(public void *.ChsCompoundCommand->execute())
OR execution(public void *.CreateCommand->execute()) OR
execution(public void *.CreateConnectionCommand-
>execute())
```

Figure 8. Command constraint pointcut for JBoss

The interceptor class called CommandInterceptor, which is given in Figure 12, performs the constraint checks with respect to the rules submitted via rules dialog. If the constraint check fails, then the command is simply ignored and not executed.

The code related to hot deployment feature of JBoss (dynamic advice binding) is given in Figure 13 and Figure 14.

4.2. Model Property Settings

Writing the model property setting as an aspect was not as trivial as the other aspects. The reason is that, there needs to be a mechanism to determine when to notify the controllers when a property is set in the model part of the software. Simply, capturing the set methods of all the fields does not solve the problem, because there are set methods for which the controllers must not be notified. Moreover, there are some methods where we need to notify the controllers even if they are not set methods for

any property. For instance, the method to add a child node to the children list of a compound is not a set method.

However, the most general case a controller should be notified is within a set method for a field. Thus we determined a general point-cut to capture the set methods and specific point-cuts for other cases. The notification mechanism in CHISIO requires an identifier string for the notification of different properties. As a result, we need to use different notification strings for each different property. However finding that notification string for different properties is not that trivial. We developed a mechanism relying on Java's reflection API. If when a property named "sample" is set within the "setSample" method, there must be a final static String named "P_SAMPLE" in the class containing the "sample" property. If the programmers follow this pattern in adding new fields, the controllers will automatically be notified when the field is set in the set method. In order to find the "P_SAMPLE" notification string, when "sample" field is set; we decided to create a hashmap between each field and the corresponding notification string. Thus we added another point-cut to capture this relation when a graph model class is loaded. We use after advices for all the point-cuts to notify the controllers. Point-cuts and advices for this aspect are given in Figures 15-18.

4.3. Layout Profiling Point-cut

All layout algorithms in CHISIO extend from an abstract class named "AbstractLayout" and there is the method "doLayout()" which starts the layout for all types. Thus in profiling, we capture the control flow of the method mentioned. We use before and after advices for that point-cut in order to calculate the total time spent within a method and the number of times a method is executed. The point-cut is given in Figure 20.

In this point-cut, we exclude all the methods outside layout package because they are out of scope. What is good about this aspect is that it is trivial to apply the profiling aspect to other parts of the code, just by changing it.

In order to measure elapsed time properly, we have used both before and after advices for this point-cut. After a call of a method, we finalize the timer for the method and save its duration. These advices are depicted in Figures 21-23.

We create a tree node for each method and update the tree constantly in order to reflect latest duration and percentage settings. In order to prevent redundancy the nodes corresponding to same methods which has the exactly same hierarchies are represented by the same node, hence making the profiling tree more meaningful. Figure 9 represents a screenshot taken during profiling.

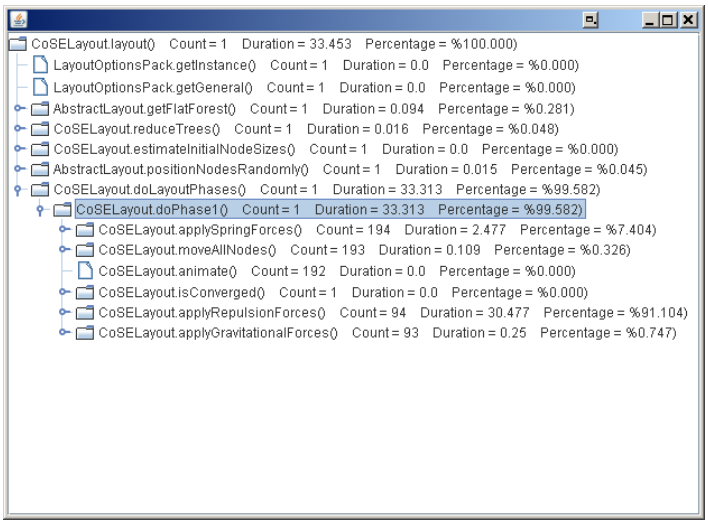

Figure 9. Layout Profiling Screenshot

4.4. Action Tracing Point-cut

Action tracking aspect is more like a specialized logger which keeps track of the executed methods of two parent classes, Command and EditPolicy, and also their child classes. Logs are generated and printed on a separate console in debug mode. Whenever user performs an interactive operation, related method calls in the associated Command and EditPolicy derivative classes are displayed.

Point-cuts and advices for this aspect are given in Figure 19.

Figure 10 illustrates a sample screenshot of the log produce after inserting a new node into the graph:

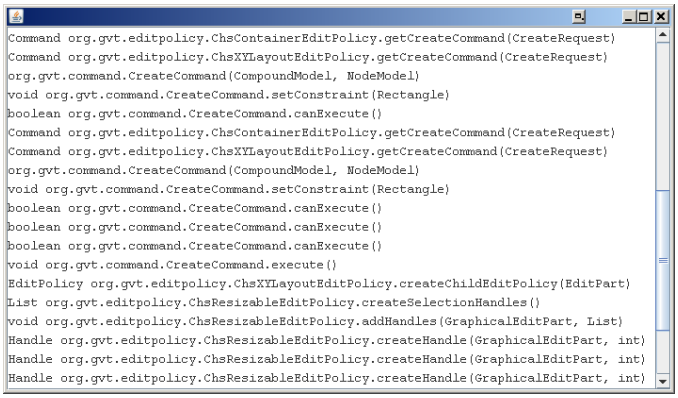

Figure 10. Action Tracking Screenshot

5. The Related Work

There are many different commercial and non-commercial graph visualization tools. Some of them, generally the ones with better capabilities and user interfaces are commercial. There are also good ones even though they are non-commercial. Tom Sawyer's TSV [1], yWorks' yFiles [2] and ILOG's JViews Diagrammer [3] are well-known examples for commercial tools. Prefuse[10], uDraw [4], Graphviz [5], VGJ [6] and Graphlet [8] are some of many non-commercial tools.

In order for a graph visualization tool to be considered as useful, it has to support some basic features. Some of them are ability to create, delete, move, resize graph objects, zooming, highlighting, scaling functionalities, save/load capability, clustering of nodes and user interface to view properties of graph objects. Although CHISIO is an academic tool, its features and capabilities are comparable to most non-commercial and some commercial tools. Figure 11, shows a comparison of CHISIO with the other visualization tools including both the commercial and non-commercial ones.

	Move/Resize/ Create/Delete	Zoom/ Scale	Compound Support	Highlighting	Save/ Load	Clustering/ Grouping	Object Inspector
TSV	+	+	+	+	+	+	+
JViews	+	+	+	+	+	+	+
yEd	+	+	+	-	+	+	+
aiSee	+	+	+	-	+	-	+
Graphviz	+	+	-	-	+	-	+
Prefuse	+	+	+	-	+	-	-
Gravisto	+	+	-	-	+	-	+
JGraphEd	+	+	-	-	+	-	+
VGJ	+	+	-	-	+	+	+
GDToolkit	+	+	-	-	+	-	+
uDraw	+	+	-	-	+	-	+
VCG	-	+	-	-	+	-	-
CHISIO	+	+	+	+	+	+	+

Figure 11. Comparison Table

However, none of these tools used AOP techniques as far as we know. CHISIO may differ from these tools since AOP technique is applied in the development cycle.

Besides, UML class drawing tools can be considered as a similar study since UML can be considered as graph visualization tool [9]. In this study, AOP technique is applied and some improvements made as an outcome of AOP.

However, the concerns in developing a UML drawing tool and a general graph drawing and layout tool seem to differ in many ways. Thus there is not an aspect oriented design of a general graph drawing tool to which we can make comparisons.

6. Discussions & Lessons Learned

During the aspect oriented refactoring of CHISIO, we gained experience about the benefits of aspect oriented software engineering. However, we also tackled with problems resulting from the fact that, CHISIO was not originally designed with the cross-cutting concerns in mind. For instance, aspectizing the cross-cutting concern of notification of the controllers by the model classes was really painful and it is not as clean and solid as the other aspects. If we were to design CHISIO from scratch with an aspect oriented approach, we might have ended up with a simpler and easy to implement aspect. Moreover, we realized the fact that, there are inherently cross-cutting concerns in a general graph visualization tool, such as checking the node-edge attachment and compound-node containment constraints or profiling the layout.

In improving CHISIO via aspect oriented software approach, we had the opportunity to use and compare two different libraries in terms of usability and functionality. We implemented most of the aspects mentioned in section 3 with AspectJ, whereas we preferred using JBoss AOP for the command execution constraints aspect.

We integrated AspectJ plug-in of Eclipse to the development environment, which provided ease of use when compared to JBoss AOP. The cross-cut references and the advice markers, which are useful in checking the code segments captured by a certain advice, provided by the plug-in increase the testing and debugging opportunity for aspect oriented programming. Since AspectJ treats the point-cuts and advices as first level abstractions, you get any possible syntax error at the time of compilation. This functionality is sure to shorten the implementation phase of the software development cycle especially for the big projects. On the other hand, for JBoss AOP you have to execute the code in order to get any possible syntax error related with your point-cuts, since you write them either in a separate XML file or as string parameters within Java code.

The biggest advantage of using JBoss AOP is the ability to add/remove advices dynamically in the runtime. For integrating command execution constraints to CHISIO, JBoss AOP seemed as a better candidate thanks to having this feature. If the constraints are to be removed dynamically at any time, one can just disable the advice via the provided user interface. After that, the interceptor

code is no longer called and this might become an important performance gain over AspectJ. This is due to the fact that, there is no way to prevent AspectJ to run/avoid certain aspect with respect to some property. The advice code is run every time regardless of anything as long as the point-cut is captured.

7. Conclusion and Future Work

We have reviewed CHISIO, a compound graph visualization and layout tool and framework, with respect to the aspect oriented approach. Since cross-cutting concerns were not explicitly considered during the development phase, we have separated the cross-cutting concerns in CHISIO as much as we can and we integrated the layout profiling feature as an aspect. By the end of this study, CHISIO has become easier to extend, maintain and reuse.

We used AspectJ as the aspect oriented programming language. Modeling and implementation of cross-cutting concerns as aspects in AspectJ is easy. This way, we have seen that AOSD reduces complexity of the software system by modularizing the cross-cutting concerns. The resulting system is less coupled and highly cohesive. This positive improvement makes the software system easily understandable, adaptable and manageable. Therefore, the overall quality of the software has been increased.

Due to the time constraints, we ended up with just two development and two production aspects. Thus, possible future work should be identification of further cross-cutting concerns existing in the software and handling those concerns in the same manner. One such concern is the enabling/disabling of "Highlighting" mechanism. Highlighting is not supported internally by GEF framework and it was implemented as a separate layer on top of the graph layer. Since it is not handled internally by the framework, it must be disabled before or enabled after some operations such as running the layout or performing animation. Thus the code related with highlight layer handling is scattered in different classes of different packages. Further cross-cutting concerns should be found with further study of the existing system as stated.

8. Acknowledgements

We would like to thank Assist. Prof. Bedir Tekinerdoğan for organizing the TAOSD workshop and giving us the opportunity to publish our work in this paper.

9. References

[1] Graph Layout Toolkit and Graph Editor Toolkit User's Guide and Reference Manual. Tom Sawyer Software, Oakland, CA, USA, 1992-2002.
http://www.tomsawyer.com.

[2] yFiles User's Guide. yWorks GmbH, D-72076 Tbingen, Germany, 2002. http://www.yworks.com.

[3] JViews User's Guide. ILOG SA, 94253 Gentilly Cedex, France, 2002.
http://www.ilog.com.

[4] uDraw(Graph), 2005.
http://www.informatik.uni-bremen.de/uDrawGraph.

[5] Graphviz - Graph Visualization Software.
http://www.graphviz.org.

[6] VGJ, Visualizing Graphs with Java. Auburn University, Auburn, Alabama.
http://www.infosun.fmi.uni-passau.de/Graphlet

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides.
Design Patterns Elements of Reusable Object Oriented
Software. Addison-Wesley, 1995..

[8] Graphlet, A toolkit for graph editors and graph algorithms. University of
Passau, Passau, Germany.
http://www.infosun.fmi.uni-passau.de/Graphlet

[9] Y. Dedeoğlu, H. Sözer, M. Tekkalmaz, A. Uluçınar. *Aspect-Oriented UML Class Diagram Drawing Tool*,1st Turkish AOSD Workshop Preceedings, TAOSD 2003.

[10] The GraphML File Format
http://graphml.graphdrawing.org/

[11] The Prefuse Visualization Toolkit.
http://prefuse.org/

[12] The AspectJ(TM) Programming Guide.
http://www.eclipse.org/aspectj/doc/released/progguide/

[13] JBoss AOP
http://www.jboss.org/jbossaop/

[14] Eclipse Graphical Editing Framework (GEF)
http://www.eclipse.org/gef/

[15] Composite Pattern
http://en.wikipedia.org/wiki/Composite_pattern

[16] Command Design Pattern
http://www.jboss.org/jbossaop/

A. Appendix

```
package org.gvt.command;

import java.util.Iterator;

public class CommandInterceptor implements Interceptor{
    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return null;
    }
     * This method performs edge validation (an edge can be incident to the
    private boolean validateEdgeNodeRelation(Object command)

     * This method performs node validation (a node can be contained by the
    private boolean validateNodeCompoundRelation(Object command)

    public Object invoke(Invocation invocation) throws Throwable
    {
        try
        {
            MethodInvocation mi = (MethodInvocation)invocation;
            Object command = mi.getTargetObject();

            boolean discard = false;

            if (command instanceof CreateConnectionCommand)
            {
                discard = this.validateEdgeNodeRelation(command);
            }
            else
            {
                discard = this.validateNodeCompoundRelation(command);
            }

            if (!discard)
            {
                return invocation.invokeNext();
            }
            else
            {
                JOptionPane.showMessageDialog(null, "This is against the rules");
            }

            return null;
      }
      finally
      {
      }
    }
}
```

Figure 12. Interceptor for Command Execution Constraints in JBoss

```
/**
 * The private constructor that makes necessary initializations
 */
private RuleDialog()
{
    this.canAttachAsSourceList = new ArrayList<String>();
    this.canAttachAsTargetList = new ArrayList<String>();
    this.canContainList = new ArrayList<String>();
    this.createContents();
    try {
        this.binding = new AdviceBinding(
                "execution(public void *.ChsCompoundCommand->execute())" +
                " OR execution(public void *.CreateCommand->execute())" +
                " OR execution(public void *." +
                    "CreateConnectionCommand->execute())",
                null);
        this.binding.addInterceptor(CommandInterceptor.class);
    } catch (ParseException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        System.exit(1);
    }
    this.setResizable(false);
}
```

Figure 13. Advice binding initialization for command execution constraint interceptor

```
/**
 * This is the method handling the events occurring as a result clicking various buttons
 * in the dialog.
 *
 * @param event, the action event occurred
 */
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == this.addButton1)
    {
        this.addContainmentRule(
            this.compoundComboBox.getSelectedItem().toString() +
            CONTAIN +
            this.allNodeComboBox3.getSelectedItem().toString());
    }
    else if (event.getSource() == this.addButton2)
    {
        this.addAttachAsSourceRule(
            this.allNodeComboBox1.getSelectedItem().toString() +
            SOURCE +
            this.edgeComboBox1.getSelectedItem().toString());
    }
    else if (event.getSource() == this.addButton3)
    {
        this.addAttachAsTargetRule(
            this.allNodeComboBox2.getSelectedItem().toString() +
            TARGET +
            this.edgeComboBox2.getSelectedItem().toString());
    }
    else if (event.getSource() == this.okButton)
    {
        this.setVisible(false);
    }
    else if (event.getSource() == this.deleteButton)
    {
        this.deleteRule();
    }
    else if (event.getSource() == this.constraintEnabler)
    {
        if (this.constraintEnabler.isSelected())
        {
            AspectManager.instance().addBinding(this.binding);
        }
        else
        {
            AspectManager.instance().removeBinding(this.binding.getName());
        }
    }
}
```

Figure 14. Dynamic advice binding for command execution constraint interceptor

```
/**
 * The pointcut to be used for obtaining information in automating the notification for
 * setXXX methods.
 */
pointcut notificationChecker():
    staticinitialization(GraphObject+);

/**
 * The pointcut to be used for actually notifying the listeners after a property is set
 * to a new value within a setXXX method.
 *
 * @param graphObject, the target graphObject
 * @param object, the new value set to the property in action
 */
pointcut propertySet(GraphObject graphObject, Object object):
    target(graphObject) &&
    set(* GraphObject+.*) &&
    args(object)&&
    withincode(void GraphObject+.set*(*));

/**
 * The pointcut to be used for notifying the listeners after an edge is added to or
 * removed from a node.
 *
pointcut edgeAdditionRemoval(NodeModel nodeModel):
    target(nodeModel) &&
    (execution(void NodeModel.add*Connection(EdgeModel)) ||
        execution(void NodeModel.remove*Connection(EdgeModel)));

/**
 * The pointcut to be used for notifying the listeners after a node is added to or
 * removed from a compound node.
 *
 * @param compoundModel, the compound node to/from which a node is added/removed
 * @param child, the child node to be added or removed
 */
pointcut childNodeAdditionRemoval(CompoundModel compoundModel, Object child):
    target(compoundModel) &&
    args(child) &&
    execution(void CompoundModel.*Child(Object));

 * The pointcut to be used for notifying the listeners after any change occurs regarding
pointcut bendPointAdditionRemoval(EdgeModel edgeModel):
    target(edgeModel) &&
    execution(void EdgeModel.*Bendpoint*(..));
```

Figure 15. Model Property Settings Pointcuts

```
/**
 * This is the notification checker advice. This advice is run when a graph model class
 * is statically loaded. It traverses through all fields and finds the notification
 * strings and maps them with the actual fields.
 */
after():notificationChecker()
{
    Class currentClass = thisJoinPointStaticPart.getSignature().getDeclaringType();
    Field[] fields = currentClass.getDeclaredFields();
    HashMap<String, String> fieldMap = new HashMap<String, String>();

    for (int i = 0; i < fields.length; i++)
    {
        String fieldName = fields[i].getName();
        fieldMap.put(fieldName.toUpperCase(), fieldName);
    }

    Iterator<String> iter = fieldMap.values().iterator();

    while(iter.hasNext())
    {
        String fieldName = iter.next();

        // If fieldName starts with "P_" and is all upper case, then this
        // is a notification field. We must enforce a property with the same name
        // without "P_" to exist in the same class.
        if (fieldName.indexOf("P_") == 0)
        {
            String theFieldNameToExist = fieldMap.get(fieldName.substring(2));

            if (theFieldNameToExist != null)
            {
                Method[] methods = currentClass.getDeclaredMethods();
                boolean cont = methods.length > 0;
                int counter = 0;
                String theMethodNameToExist = "set" +
                    theFieldNameToExist.substring(0, 1).toUpperCase() +
                    theFieldNameToExist.substring(1);

                while (cont)
                {
                    cont = !(theMethodNameToExist.equals(methods[counter].getName()));

                    if (cont)
                    {
                        counter++;

                        if (counter == methods.length)
                        {
                            cont = false;
                        }
                    }
                }

                if (counter == methods.length)
                {
                    // TODO: Uygun bir mesajla hata belirticez

                }

                this.fieldNotificationMap.put(theFieldNameToExist, fieldName);
                this.classFieldMap.put(fieldName, currentClass);
            }
        }
    }
}
```

Figure 16. Model Property Settings advices Part 1

```
    /**
 * This advice is executed when a property for which the listeners must be notified is
 * set to a new value within a setXXX method.
 *
 * @param graphObject, the target graphObject
 * @param object, the new value of the property in action
 */
after(GraphObject graphObject, Object object):propertySet(graphObject, object)
{
    String fieldName = thisJoinPointStaticPart.getSignature().getName();
    String notificationFieldName = this.fieldNotificationMap.get(fieldName);

    if (notificationFieldName != null)
    {
        Class currentClass = this.classFieldMap.get(notificationFieldName);

        try
        {
            graphObject.firePropertyChange(
                (String)currentClass.
                    getDeclaredField(notificationFieldName).get(graphObject),
                null,
                object);
        }
        catch(Exception e)
        {
        }
    }
}

/**
 * This is the advice that is executed after an edge is added to or removed from a node
 * model.
 *
 * @param nodeModel, the node model in action
 */
after(NodeModel nodeModel):edgeAdditionRemoval(nodeModel)
{
    String methodName = thisJoinPointStaticPart.getSignature().getName();

    if (methodName.indexOf("Source") >= 0)
    {
        nodeModel.firePropertyChange(NodeModel.P_CONNX_SOURCE, null, null);
    }
    else
    {
        nodeModel.firePropertyChange(NodeModel.P_CONNX_TARGET, null, null);
    }
}
```

Figure 17. Model Property Settings advices Part 2

```
/**
 * This is the advice that is executed after a node is added to or removed from a compou
 * node model.
 *
 * @param compoundModel, the compound model in action
 * @param child, the child node in action
 */
after(CompoundModel compoundModel, Object child):childNodeAdditionRemoval(
        compoundModel, child)
{
    String methodName = thisJoinPoint.getSignature().getName();

    if (methodName.indexOf("add") >= 0)
    {
        compoundModel.firePropertyChange(CompoundModel.P_CHILDREN, -1, child);
    }
    else
    {
        compoundModel.firePropertyChange(CompoundModel.P_CHILDREN, child, null);
    }
}

/**
 * This advice is run when any change for an edge regarding the edge bend points occur.
 *
 * @param edgeModel, the edge model in action
 */
after(EdgeModel edgeModel):bendPointAdditionRemoval(edgeModel)
{
    edgeModel.firePropertyChange(EdgeModel.P_BENDPOINT, null, null);
}
```

Figure 18. Model Property Settings advices Part 3

```
package org.gvt.action;

import org.aspectj.lang.Signature;


public aspect ActionTracker
{
    /**
     * Pointcut for catching edit policy related methods.
     */
    pointcut editPolicyMethods() : execution (* org.eclipse.gef.EditPolicy+.*(..) );

    /**
     * Pointcut for catching Command related methods.
     */
    pointcut commandMethods() : execution (* org.eclipse.gef.commands.Command+.*(..) );

    /**
     * Pointcut for catching Command constructors
     */
    pointcut commandConstructionMethods() : execution (org.eclipse.gef.commands.Command+.new(..));

    /**
     * Advice for logging edit policy related methods.
     */
    before() : editPolicyMethods()
    {
        this.log(thisJoinPoint.getSignature());
    }

    /**
     * Advice for logging command related methods.
     */
    before() : commandMethods()
    {
        this.log(thisJoinPoint.getSignature());
    }

    /**
     * Advice for logging command construction methods.
     */
    before() : commandConstructionMethods()
    {
        this.log(thisJoinPoint.getSignature());
    }

    private void log(Signature signature)
    {
        ActionTrackerDialog.getInstance().queue(signature);
    }
}
```

Figure 19. Action Tracking Aspect

```
 * Pointcut for catching runLayout method executions.
pointcut runLayoutMethod():
    execution(* org.gvt.layout.AbstractLayout.runLayout(..));

 * Pointcut for catching all method executions in the control flow of layout()
pointcut allLayoutMethodCalls():
    execution(* org.gvt.layout.*.*(..)) &&
    !runLayoutMethod() &&
    cflow(execution(* org.gvt.layout.AbstractLayout+.layout(..)));
```

Figure 20. Layout Profiling Ponitcuts

```
    * This advice is executed whenever the layout button is pressed.
   before(): runLayoutMethod()
   {
       // Launch profiling dialog in a new thread
       new Thread(new Runnable()
       {
           public void run()
           {
               ProfilingDialog.getInstance().setAutoRefresh(true);
               ProfilingDialog.getInstance().show();
           }
       }).start();
   }

    * This advice is executed after layout finishes. It stops refreshing the tree
   after(): runLayoutMethod()
   {
       ProfilingDialog.getInstance().setAutoRefresh(false);
   }
```

Figure 21. Layout Profiling Aspects Part 1

```
/**
 * This advice is executed before a layout method is called.
 */
before(): allLayoutMethodCalls()
{
    Signature methodSignature = thisJoinPointStaticPart.getSignature();

    // Save the current node with the current hierarchy information as the key.
    // This information will be used for merging the nodes that corresponds
    // to same method and having the same call hierarcy

    CURRENT.append(methodSignature.toShortString());
    //String methodCallHierarchy = this.deriveMethodCallHiearchy(methodSignature);

    // Check, if the method is called in the same hierarchy before.
    DefaultMutableTreeNode currentTreeNode = this.methodLevelData.get(CURRENT.toString());

    MethodData data;

    // Special case handling for layout() method.
    if (this.methodExecutionStack.isEmpty())
    {
        currentTreeNode = (DefaultMutableTreeNode)this.treeModel.getRoot();

        data = new MethodData(methodSignature);

        currentTreeNode.setUserObject(data);
    }
    // No reuse available, create the node and method data objects
    else if (currentTreeNode == null)
    {
        data = new MethodData(methodSignature);

        currentTreeNode = new DefaultMutableTreeNode(data);
        DefaultMutableTreeNode parentNode = this.methodExecutionStack.peek();
        parentNode.add(currentTreeNode);

        // for dynamic update, notify tree model for the newly inserted nodes.

        int[] indicies = new int[1];
        indicies[0] = parentNode.getChildCount()-1;
        ((DefaultTreeModel)this.treeModel).nodesWereInserted(parentNode, indicies);

        // Save current node for further possible reuse.
        this.methodLevelData.put(CURRENT.toString(), currentTreeNode);
    }
    // Reusing previously created tree node.
    else
    {
        data = (MethodData) currentTreeNode.getUserObject();
    }

    // Save method start time.
    data.start = System.currentTimeMillis();

    // Increment call count
    data.callCount++;

    // Push this method to stack, because it is currently executing
    this.methodExecutionStack.push(currentTreeNode);
}
```

Figure 22. Layout Profiling Aspects Part 2

```
/**
 * This advice is executed after a layout method is called.
 */
after(): allLayoutMethodCalls()
{
    // Pop the topmost node, its execution finished.
    DefaultMutableTreeNode currentTreeNode = this.methodExecutionStack.pop();

    MethodData methodData = (MethodData) currentTreeNode.getUserObject();

    // Save elapsed duration
    methodData.duration += (System.currentTimeMillis() - methodData.start);

    // Reset timer
    methodData.start = 0;

    String thisMethod = methodData.getMethodSignature().toShortString();
    int index = CURRENT.lastIndexOf(thisMethod);
    CURRENT.replace(index, CURRENT.length(), "");
}
```

Figure 23. Layout Profiling Aspects Part 3