Hogeschool Utrecht

System documentation

Architecture Graphics Service

Team Architecture Graphics 2012
Tom de Bruijn
Johan van der Slikke
Guido van Tricht
Patrick van der Willik

Team Architecture Graphics 2013
Stan Verhoeckx
Dennis Bullee
Tim Bazuin
Ruwan Kellaert
Mark Wallenburg

1-8-2013

1 Inhoudsopgave

2 Graphics Service System documentation

2.1 Introduction

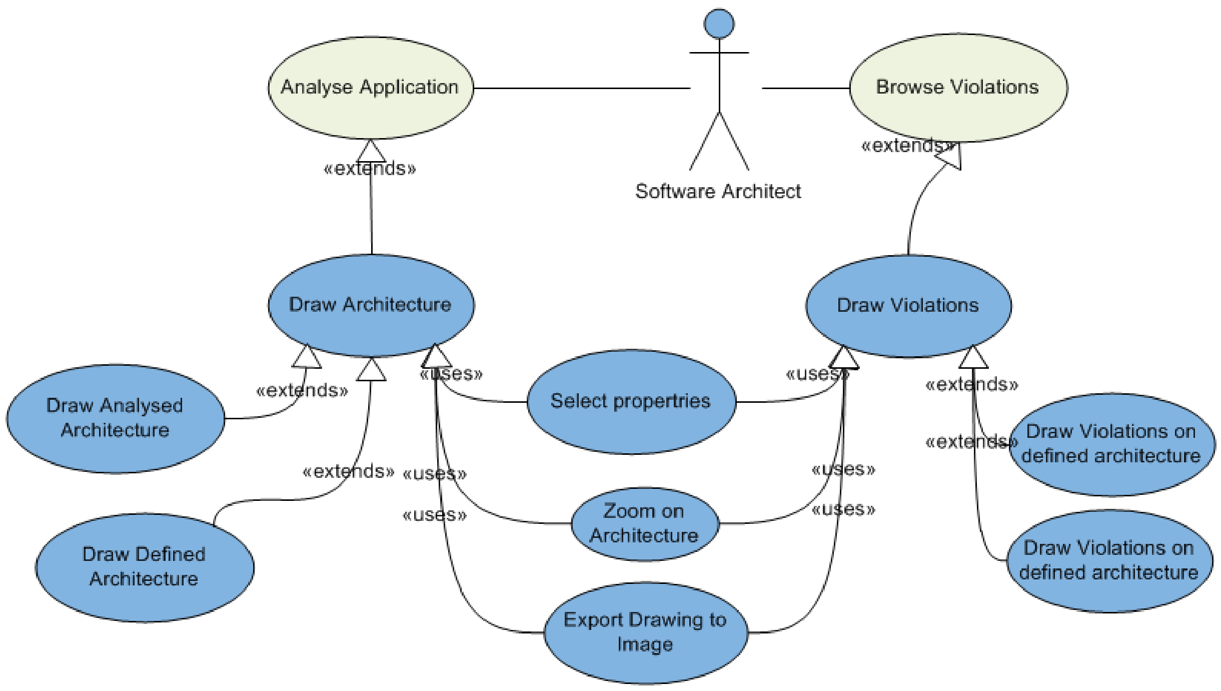
The purpose of the Graphics Service within the HUSACCT application is to visualize the defined and analysed models. With this module it’s possible to navigate through the architecture, visualise dependencies and relations between modules and visualize violations on the defined rules.

The Graphics Service communicates with the Analyse Service, Define Service and Validate Service to gather the information it needs in order to generate the models. Internally the Graphics Service uses the *JHotDraw* library to draw the generated drawings.

The Graphics Service is separated in several sub-components and layers. These sub-components will be explained in chapter 2. The Presentation and Task layers are the most important two layers within the Graphics Service. The Presentation layer is explained in chapter 3 and will describe the inner workings of *JHotDraw* and the setup of how the service utilities it. Chapter 4 describes the Task layer. This layer contains the logic behind all of the features. Abstraction is only used for the “export to image” features and will briefly be explained in chapter 5.

2.2 Functionality

2.2.1 Use case model



2.2.2 Use case descriptions

2.2.2.1 *Use case: Draw Architecture*

Nr.	U1
Version Nr.	1.1
Author	Guido van Tricht, Tim Bazuin
Priority	Must (MoSCoW)
Use case	Draw Architecture
Actors	User (A1)
Summary	A user opens the analyzed architecture, in which a package diagram can be viewed and class diagram can be displayed.
Precondition	A user has analyzed the source code
Primary condition	A.1.1 : A user clicks on Analyse. A.1.2 : A user clicks on Analysed Architecture Diagram. S.2.1 : The system asks for the analyzed architecture. S.2.2: The system shows the highest level of the analyzed architecture.
Post condition	There is a graphical representation of the architecture shown on the screen.

2.2.2.2 *Use case: Draw Violations on analyzed Architecture*

Nr.	U2
Version Nr.	1.1
Author	Guido van Tricht, Tim Bazuin
Priority	Must (MoSCoW)
Use case	Draw Violations on analyzed architecture
Actors	User (A1)
Summary	A user has analyzed and validated the source code. The violations based on the architecture can now be shown on screen.
Precondition	A user has defined an architecture, created rules for it, analyzed the source code and validated the code.
Primary condition	A.1.1 : A user clicks on Draw Violations. S.2.1 : The system shows the analyzed architecture with violations, which are shown with colors.
Post condition	There is a graphical representation of the architecture shown on the screen with violations.

2.2.2.3 *Use case: Draw Violations on defined architecture*

Nr.	U3
Version Nr.	1.0
Author	Guido van Tricht
Priority	Must (MoSCoW)
Use case	Draw Violations on defined architecture
Actors	User (A1)
Summary	A user has defined an architecture, created rules on it, analyzed the source code and validated the code. A user can now see a graphical representation of the defined architecture with the violations on top of it.
Precondition	A user has defined an architecture, created rules on it, analyzed the source code and validated the code.
Primary condition	A.1.1 : A user clicks on Draw Violations. S.2.1 : The system shows the defined architecture with violations, which are shown with colors.
Post condition	There is a graphical representation of the architecture shown on the screen with violations.

2.2.2.4 *Use case: Zoom on Architecture*

Nr.	U3	
Version Nr.	1.0	
Author	Tom de Bruijn	
Priority	Should (MoSCoW)	
Use case	Zoom on Architecture	
Actors	User (A1)	
Summary	A user has opened the architecture and can now zoom between the different levels of the defined or analyzed architecture.	
Precondition	A user has defined an architecture or analyzed the source code.	
Primary condition	A.1.1 : A user clicks on a package in the shown architecture. S.2.1 : The system zooms in on the selected package, showing the child architecture. A.3.1: A user clicks on a class in the shown architecture. S.4.1: The system zooms in on the selected class.	
Alternative scenarios	AS1	[There are violations shown in the architecture] A.1.1 : A user clicks on a module in the shown architecture. S.2.1 : The system zooms in on the selected module, showing the child architecture with the violations on top of it.
	AS2	S.2.2 [There is another package underneath] There are also shown packages next to the classes. A user can now also zoom in on the package shown. (A.1.1)
Post condition	There is a graphical representation of the selected architecture shown on the screen with or without violations.	

2.2.2.5 *Use case: Select properties*

Nr.	U4
Version Nr.	1.1
Author	Tom de Bruijn, Tim Bazuin
Priority	Could (MoSCoW)
Use case	Select properties
Actors	User (A1)
Summary	A user can select a figure in the shown architecture. Once selected, the properties panel will show the properties of the selected figure.
Precondition	A user has defined an architecture or analyzed the source and has opened the graphics view.
Primary condition	A.1.1 : A user clicks on a module in the shown architecture. S.2.1 : The system asks for the properties of the selected module and shows these properties in the properties panel.
Post condition	There is an detailed description shown about the selected module in the properties panel.

2.2.2.6 *Use case: Export To Image*

Nr.	U5	
Version Nr.	1.0	
Author	Tom de Bruijn	
Priority	Could (MoSCoW)	
Use case	Export To Image	
Actors	User (A1)	
Summary	A user saves the shown architecture as an image.	
Precondition	A user has defined an architecture or analyzed the source and has opened the graphics view.	
Primary condition	A.1.1 : A user clicks on the export button. S.2.1 : The system shows a file browser in which a user can select a location where the image should be saved. A.3.1: A user selects a location and defines a file name. S.4.1: The system exports the shown architecture as an image in the defined location.	
Alternative scenarios	AS1	[The file name is already in use] S.4.2 : The system shows a message, telling a user that the name is already in use and asks a user is he/she wants to overwrite the file. A.5.1: A user selects “Ja/Yes”. S.6.1: The system overwrites the file.
	AS2	[The file name is already in use] S.4.2 : The system shows a message, telling a user that the name is already in use and asks a user is he/she wants to overwrite the file. A.5.1: A user selects “Nee/No”. S.6.1: The system does not overwrite the file and shows the file browser again.
Post condition	An image of the shown architecture is saved in the defined location.	

2.3 Decisions and justification

Decision	Justification
We are going to use the Java library *JHotDraw*. With this library we will be able to draw the architecture of Java based source code as for other programming languages.	NF1
We are going to use the Java library *JHotDraw*. This library can be used in Windows, Linux and Mac.	NF2
We use the Factory Pattern for an abstract solution for the usage of the different DTOs provided by the other teams.	NF1
We will use *JInternalFrames* to show our graphics view. This decision was made by the Control team.	NF3
We've separated presentation and task logic into two separate layers. The task layer is does not contain direct knowledge of which technologies (*JHotDraw*) used in the presentation layer.	NF2
Logic of the file system for exporting the images of a drawing is present in our service (abstraction layer) so that control does not have to make a separate use case for this feature.	
We've added calls to the graphics interface to allow the direct call to show violations in a Drawing. Thisallowsforless user interaction.	NF4
We've separated calls to retrieve the drawing *JInternalFrames* and to draw them. This way we decrease the amount of requests in case the drawing has already been drawn.	NF3
Data is requested from the other services on a level by level basis. This way we only receive information we are using in the drawing and we do not contain any logic of the structure of the analysed architecture of an application.	NF3
Data such as dependencies and violations are retrieved by multiple calls to the analyse and validate services for each combination of the DTOs. This way we do not contain any logic of (*Analysed)ModuleDTOs to DependencyDTOs* and *ViolationDTOs*.	
Although not used decorators are present in the presentation layer for figures. This allows for more dynamic behavior to be added later on in development. Decorators allow certain behavior or graphical additions to be added.	NF1
We've created separate controllers for the analyse and define services. The logic that controls the drawing mechanism is present in the *DrawingController*. The separate controllers handle only the communication with the analyse, define and validate services.	
We manage all types of DTOs and what they represent (figures) in a *FigureMap*. This contains *HashMaps* that links figures to DTOs for easy retrieval of DTOs without putting the logic in the *DrawingController*.	NF1
A *UserInputListener* is implemented into the controllers to handle requests from the presentation layer. This allows for other usages as well in the future such as a context sensitive right-click mouse menu.	
Other services (analyse, define and validate) do not inform us of any changes in their components thus we implemented a refresh option to allow users to redraw the drawings based on the new data.	NF4
In order to position the drawn figures in the drawing we make use of *LayoutStrategies*. We aim to implement two different kinds of *LayoutStrategies*. One that will position the figures on a table basis (columns and rows) and one that will take into account the shortest routes to other elements for dependency and violation lines.	
We are using threading in order to keep the application responsive during CPU intensive work	NF4 / NF7

We’re providing multi-input methods for commands in order to provide a user with a rich user experience	NF4

2.4 Software partitioning

The Graphics service has a service class called *GraphicsServiceImpl* which implements the interface *IGraphicsService*. There are methods for both the analysed and defined diagrams so they can operate independently from one another.

The control service expects a *JInternalFrame* back from the *getAnalysedArchitectureGUI* and *getDefinedArchitectureGUI*. For the workspace save and load methods an XML library is used.

The *draw[Analysed/Defined]Architecture* methods draw the root of these diagrams by requesting the information from the services. The service manages the other views of the service in the service itself.

The *draw[Analysed/Defined]ArchitectureWithViolations* methods have the same functionality as the *draw[Analysed/Defined]Architecture* methods, but they activate the violation lines on startup. This is currently not used by the control group which is why the service has its own UI button to activate this.

The *[get/load]WorkspaceData* functions are used to retrieve and load workspace data, respectively.

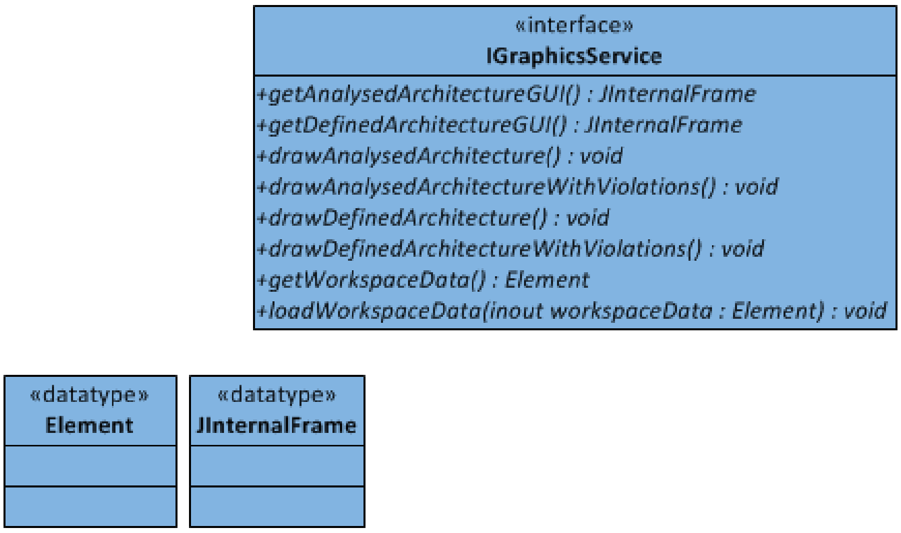

«interface»
IGraphicsService
+getAnalysedArchitectureGUI() : JInternalFrame
+getDefinedArchitectureGUI() : JInternalFrame
+drawAnalysedArchitecture() : void
+drawAnalysedArchitectureWithViolations() : void
+drawDefinedArchitecture() : void
+drawDefinedArchitectureWithViolations() : void
+getWorkspaceData() : Element
+loadWorkspaceData(inout workspaceData : Element) : void
«datatype»
Element
«datatype»
JInternalFrame

Diagram 1. Service Definition

As explained in the introduction, the Graphics Service (here shown as *ArchitectureGraphicsService*) uses the Analyse, Define and Validate Service to collect the data necessary to draw the models.

The Control Service is used for selecting the language of the application, and to tell other services when to update their view based on the changes made in the Graphics Service.

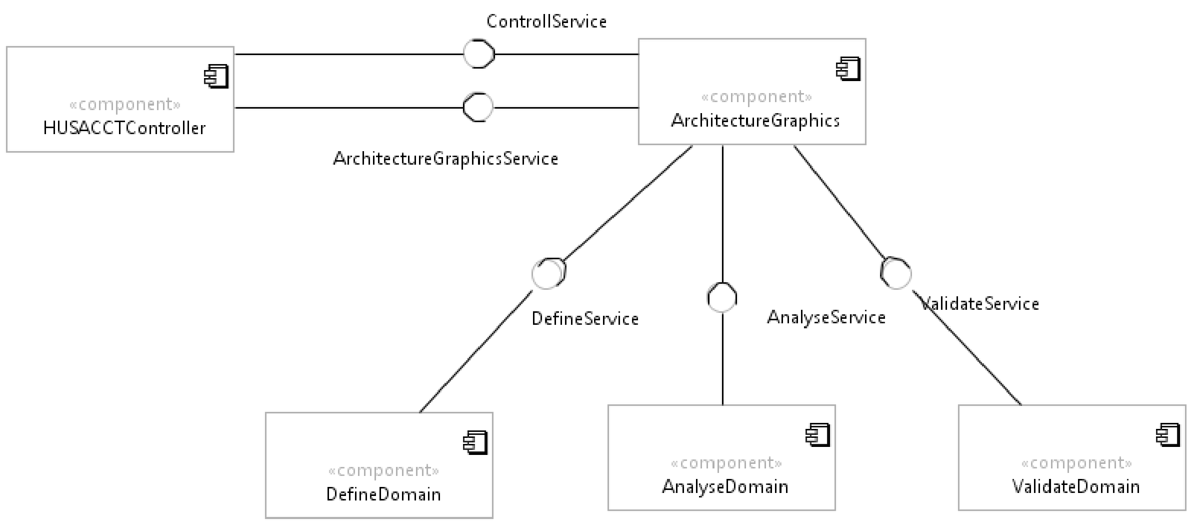


Diagram 2 Services used

2.5 Physical Layer Model

This physical layer model was created as a derivative of the project –wide structure of services. This project wide setup separates the service into three layers: presentation, task and domain. An abstraction layer could be added in the future.

For the Graphics service the setup is quite different from the other services. There is nothing to analyse, no data to save, no checks to execute. The service is simple in terms of what it needs to know.

The setup for the Graphics service got rid of the domain layer and created an *util* layer to be used by every layer. There was no actual information to be saved so no domain layer was needed. The only reason the abstraction layer exists in the service currently is for the “export to image” feature.

The *util* layer can be used by both layers as it contains logic about the setup, the *FigureMap*, the *UserInputActionListener* and enumerators. It also contains the threading setup which is used by the task layer, but isn’t actually a part of it.

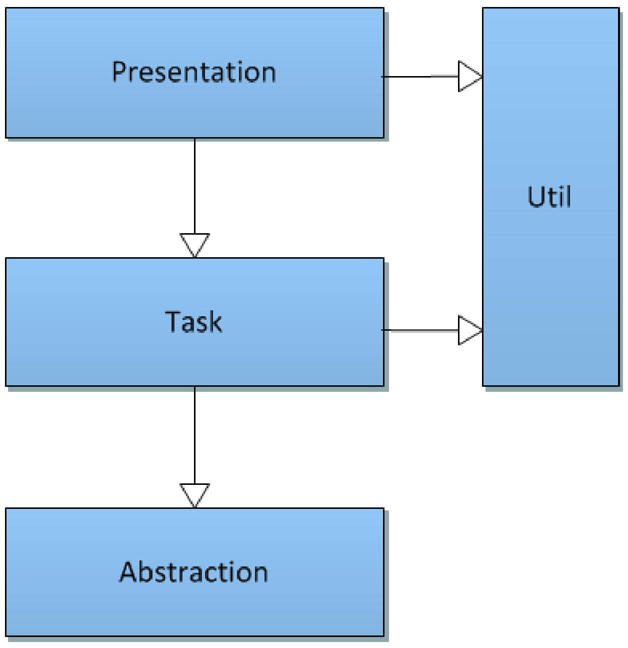


Diagram 3. Physical Layer Model

2.6 Presentation layer

The Presentation Layer is tasked with converting the generated drawing into a visual representation and to display this to the user. Once shown on screen the Presentation layer also allows a user to interact with the drawing through the use of widgets such as buttons, dialogs and context menus. The presentation layer is divided into two components: the *JHotDraw* component and the user interface widgets.

Communication from the Presentation layer to the Task layer and handling user events that are related to these events is done through the use of an Observer pattern. This is the *UserInputListener* interface.

The figures created in the presentation layer are created according to the UML 2.x standard. The figures only show the objects themselves and not their contents; variables and methods. If an object with a type that is unfamiliar to the service is found it will create a default Module figure which is a simple square figure.

2.7 JHotDraw drawing

The choice to use *JHotDraw* to generate the drawings was made after some short preliminary research and the construction of a simple prototype. During the Inception phase of the first (2012) HUSACCT project Christian Köppe informed us that *JHotDraw* could be a very good framework to use within the project because it was built using many of the design patterns from the Gang of Four. A search for alternative libraries led to two Eclipse projects.

Based on our initial success with *JHotDraw* and the fact that these two Eclipse libraries appeared very complex we decided to go with *JHotDraw*. Later on in the development however we were confronted with the major issue that *JHotDraw* suffers from: the lack of documentation. *JHotDraw*'s documentation consists of the *JavaDoc* generated from the source code. The documentation is quite extensive about the used patterns and how these patterns are applied but it does not explain how to use *JHotDraw* or how certain components within *JHotDraw* work.

Due to the lack of documentation we had to spend a serious amount of time reverse engineering *JHotDraw* to find out how it worked or how to replace certain objects with our own implementation.

2.7.1 Setup JHotDraw

The three most important components within *JHotDraw* that work together to construct a drawing are the *DrawingView*, Drawing, Figure interfaces. Below is a class diagram showing *JHotDraw* and below that how these interfaces are used within the HUSACCT project.

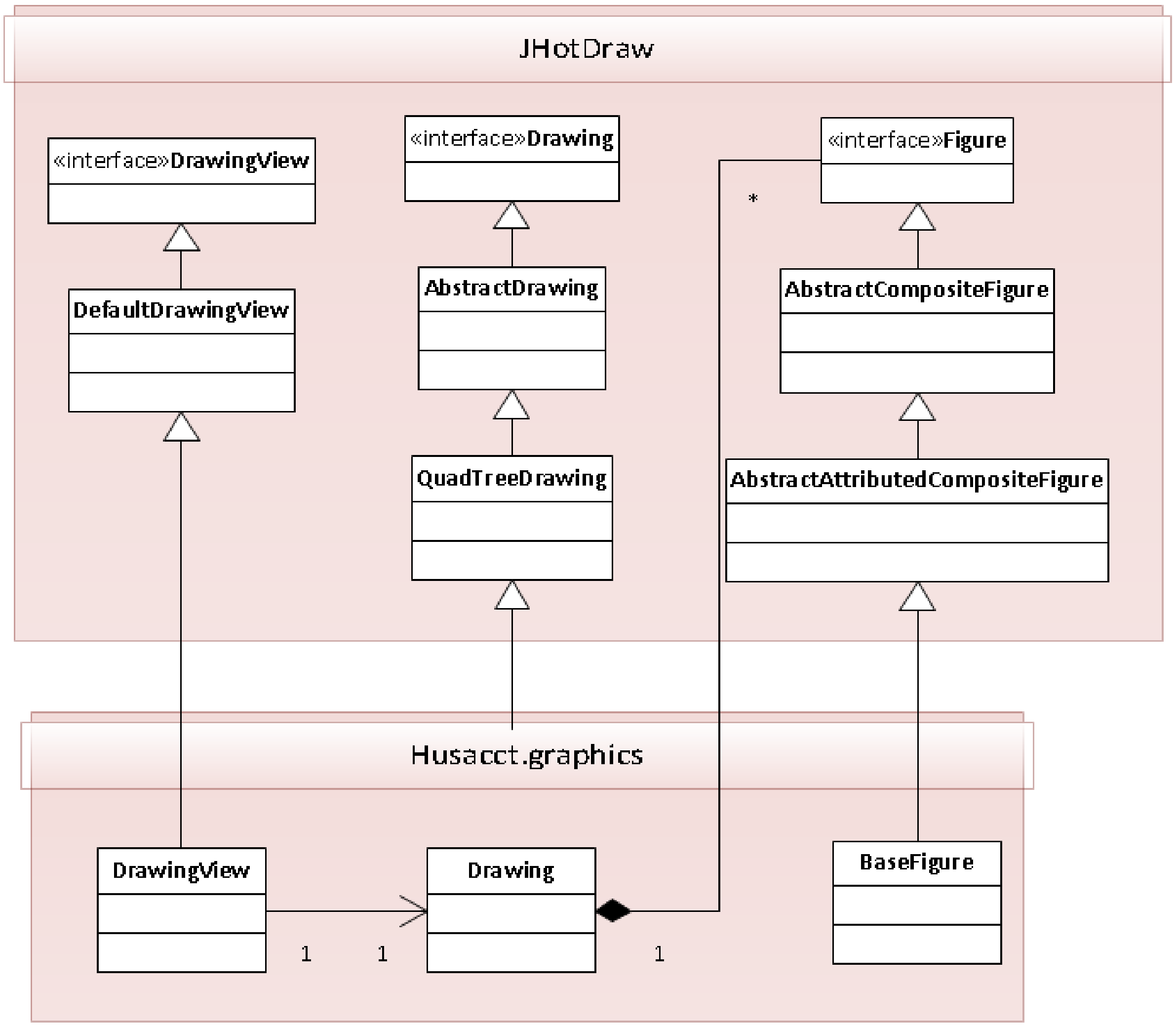


Diagram 4. JHotDraw implementation HUSACCT

JHotDraw adds a few layers of abstraction to promote loose coupling and modularity. In our initial implementation we derived our Drawing class from the *JHotDraw DefaultDrawing*. During larger tests we noticed that updating the drawing becomes slow. By replacing the *DefaultDrawing* with the *QuadTreeDrawing* we solved this problem. Had *JHotDraw* not added those layers of abstraction this would've required significant re-factoring.

In chapter 3.2.2 the further implementation within HUSACCT is explained in detail.

2.7.2 Implementation in HUSACCT

The class diagram below shows how HUSACCT implements the *JHotDraw* framework.

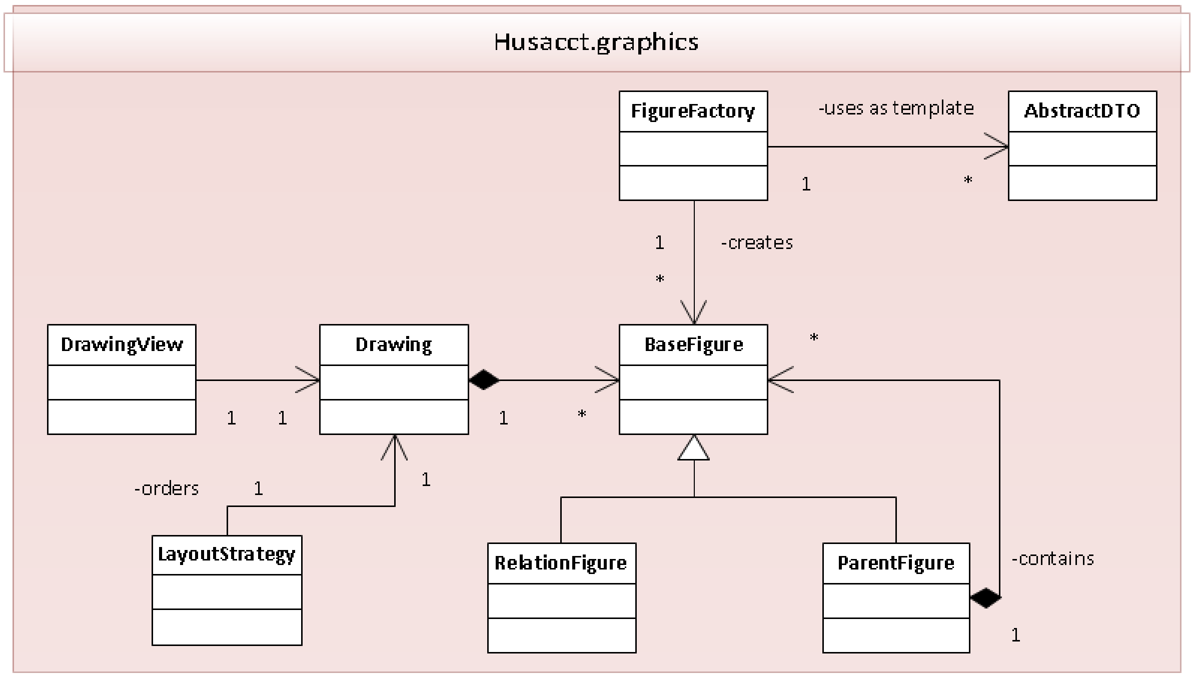


Diagram 5. JHotDraw implementation HUSACCT

The *Drawing* represents the container that holds all figures within the drawing by implementing the *JHotDraw Drawing* interface. We've specifically selected the *QuadTreeDrawing* as base class because the *DefaultDrawing* cannot handle 10 or more drawn objects before becoming slow. The *QuadTreeDrawing* uses a quad-tree to efficiently store and retrieve the drawn objects within a 2D plane. Drawn objects can be added to a Drawing by implementing the *Figure* interface and adding those objects to the Drawing. Within the HUSACCT project we have the base class *BaseFigure* which implements the Figure interface and implements various tasks concerning working with *BaseFigures*.

The Graphics Component provides a set of custom implemented Figures to represent each visual component that can be added to a drawing. Examples are the *ClassFigure*, which represent a class, the *RelationFigure*, which represents a relationship between two other figures/modules and the *ParentFigure*, which acts as a self-sorting container. These classes are left-out of the class diagram to keep it readable.

Creating new Figure instances can be done with the help of the *FigureFactory*. It uses the *factory pattern* to create new objects based on an *AbstractDTO* and returns the *BaseFigure*. The *AbstractDTO* is part of the *common* package and is based on the *data transfer object (DTO) pattern*. Figures are based on *AbstractDTOs* but do not have a direct dependency with them. A Figure does not need to know the details about its DTO to know how to render itself. We have chosen to let the *Controller* classes handle the registration and linking of DTOs to Figures. Controllers use the *FigureMap* to this purpose.

Since Drawings are only collections of Figure objects *JHotDraw* doesn't automatically sort or order drawings. In order to apply a specific layout to a Drawing we have implemented *LayoutStrategy* classes. These classes are based on the *strategy pattern* to order to allow for changeable layout algorithms. The *NoLayoutStrategy* is an empty layout strategy that doesn't apply any layout to the drawing. The *BasicLayoutStrategy* attempts to generate a plain square object tree by placing the figures in an equal number of rows and columns. The *LayeredLayoutStrategy* is an attempt at implementing a smart strategy. The *LayeredLayoutStrategy* is incomplete and should not be shipped for production.

Last but not least is the *DrawingView*. This class takes a Drawing and renders the contents of the drawing and presents it to the end-user. The *DrawingView* is a user-interface that implements the Component interface and also supports handling of user interaction. A user can interact with the *DrawingView* through a context menu. The *DrawingView* makes use of the *observer pattern* to allow other user interface components to listen for user interaction through the *UserInputListener*. The controllers are subscribed to the *DrawingView* to listen for these events.

2.7.3 Line Separation

Methods for separating overlapping lines are implemented using strategies. These strategies reside in *husacct.graphics.presentation.linelayoutstrategies* and are defined by the *ILineSeparationStrategy* interface.

The method *updateLineFigureToContext* in the Drawing class defines the interface. It calls the method *seperateOverlappingLineFigures* with this strategy. This method finds overlapping lines and feeds them to the strategy.

These are the currently available strategies:

- *ConnectorLineSeparationStrategy*
  This strategy separates lines by creating a distance between them by distancing their connectors.
- *ElbowLineSeparationStrategy*
  This strategy separates lines by moving the center points away from each other. These lines will then show as an elbowed line.

2.7.4 Figure Decorators

Figures can have decorators. These are classes defined by the *husacct.graphics.presentation.presentation.decorators.Decorator* interface, which can manipulate a figure before it gets drawn. Decorators can be added to a *BaseFigure* by its *addDecorator* method.

2.8 User interface

The user interface part of the Graphics Service consists of the classes that are responsible with the user interaction and reporting these events back to the Controller for further processing. Most of the logic concerning event handling is done by these classes, generating more specific events to notify the Controllers.

The Controllers are therefore not concerned with handling the low-level Click events. Instead they concern themselves with the high-level action that has been linked with the button, for example zooming or hiding figures.

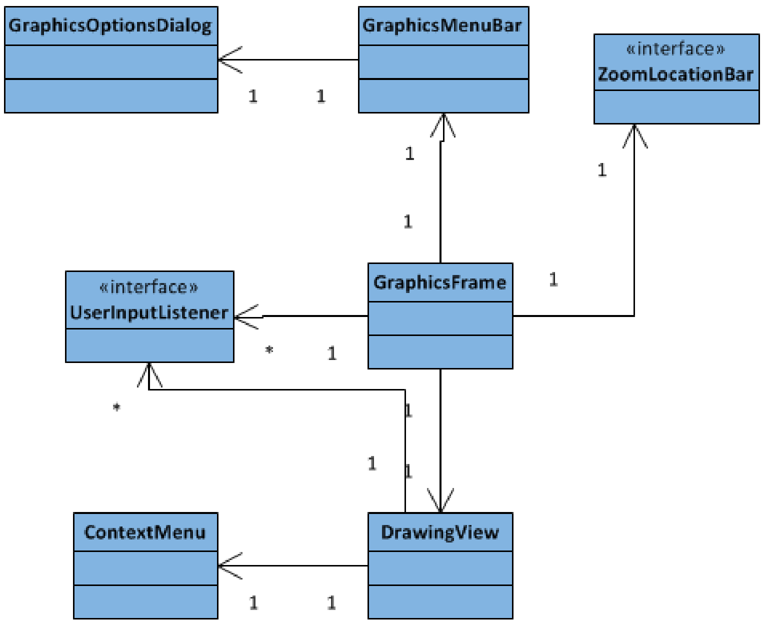


Diagram 6. User interface widgets

The GraphicsFrame is a JInternalFrame inherited class which responsibility it is to contain all user interface widgets and interact with the user. It contains the DrawingView to render the drawing, the ZoomLocationBar to display where within the hierarchy a user is and the GraphicsMenuBar to provide a user with a toolbar to navigate through the architecture or work with the drawing.

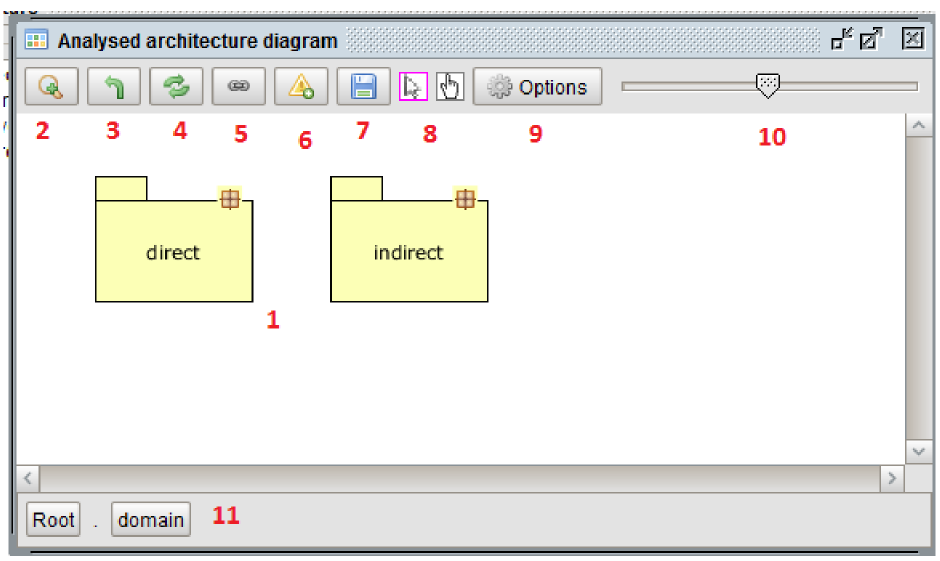


Image 1. Graphics Service dialog

The screenshot above shows the Graphics Service component in action. The highlighted areas are:

2-8:	GraphicsMenuBar
9:	GraphicsMenuBar button which opens the GraphicsOptionsDialog
10:	Scaling slider on the GraphicsMenuBar responsible for scaling the drawing
11:	ZoomLocationBar used for navigating the hierarchy

The GraphicsMenuBar also allows a user to create a modeless dialog. This is the GraphicsOptionsDialog. This dialog provides a collection of the same options, with labels added for clarification.

Right-clicking in the drawing will cause the event handling code inside the DrawingView to display a ContextMenu based on the state of the drawing such as selection and hidden figures. Any actions selected in the ContextMenu will be communicated back to the Controller through the UserInputListener.

The UserInputListener provides the interface through which the Controller will be notified about events such as zooming, scaling or selecting. Both the GraphicsFrame and the DrawingView notify listeners through this interface.

2.9 Tasklayer

The Task layer contains all the logic for the Graphics service main features. They are controlled from the main controller, *DrawingController*. Two controllers extend this abstract controller: AnalysedController and DefinedController which create the physical and logical diagrams.

A lot of features are shared among these two controllers through the *DrawingController*. Most features are triggered through the *UserInputActionListener* on the main controller. The zoom in, zoom out, refresh, toggle dependencies and violations are among these. These and more features are described in the chapter below.

2.9.1 Analysed and Defined Controllers

The Architecture Graphics service is completely dependent of other services. It requests data from other services such as the analyse service for the physical diagram. For the defined logical diagram it uses a combination of the define and analyse services. In both cases it uses the analyse service to retrieve dependencies between physical and logical modules.

To display those diagrams two controllers have been created; the AnalysedController and the DefinedController. These controllers have no knowledge of the actual drawing process; they only retrieve data from the services. A minimal amount of known services is desired to limit the amount of dependencies.

The retrieval of data is done in methods of *UserInputActionListener* such as *moduleZoom*, *moduleZoomOut*, etc. They send the data (DTOs) to the *getAndDrawModulesIn* method which checks if it's a valid path (different per service) and sets the current path before telling the *DrawingController* to draw the requested diagram.

Methods such as *getDependenciesBetween* are only used to retrieve data. Every controller needs these as the *DrawingController* needs to be able to make this call to do it's work.

2.9.1.1 *Adding a new controller*

If another type of graphics ever needs to be supported a couple of things need to be done. First create a new controller that extends the *DrawingController*. Then modify the *GraphicsService* to be able to draw and return a GUI for this new controller. Implement the controller with its methods based on the Analysed and DefinedController. They are "dumb" controllers which only retrieve data and tell the *DrawingController* what to use for the drawing.

The Drawing (JHotDraw) and the JInternalFrame (GraphicsFrame) can be easily reused.

2.9.1.2 *Validate service*

Both controllers are responsible for retrieving all the data required to show violations. This means you can show violations on both a physical and logical level. To allow this to work both controllers use the validate service. When a user tells the service to show violations it will requests the data from the validate service and will go through all the known objects (currently shown figures), retrieving their DTOs (more on that later).

It uses these DTOs to get every combination between them and check with the validate service if any rules are broken between them. The *DrawingController* then uses this data to show the violation lines.

2.9.2 DrawingController

The data received from the services is sent from the specific controller to its inherited methods. These inherited methods originate from the abstract class *DrawingController*. This controller is responsible for telling the presentation layer what to draw without having too much knowledge of it.

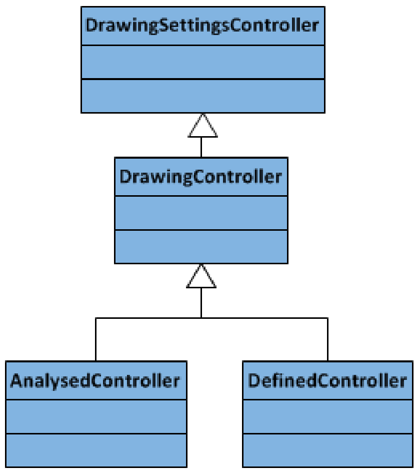


Diagram 7. Controllers

It is responsible for creating figures used in the presentation layer by calling the *FigureFactory*. It then gives these figures to the Drawing in the presentation layer which in turn uses *JHotDraw* to show them in the User Interface.

2.9.3 Dependency lines

After adding the figures to the drawing it will ask the specific controller to retrieve the dependencies between all the figures. It then goes over the same process and asks the *FigureFactory* to create *LineFigures* to show between the concerned figures. To connect the lines to the figures it uses the *FigureConnectionStrategy*, which handles the actual connecting.

2.9.3.1 *Violation lines*

When the option is activated the *DrawingController* will also call the specific controller to retrieve the violation data from the validate service. Again it uses the *FigureFactory* to create the *LineFigures* and the *FigureConnectionStrategy* to connect them to figures.

2.9.4 Zooming

The graphics service supports two kinds of zooming. The most simple version of zooming is double clicking on a figure in the diagram, or selecting a figure and pressing the zoom button in the menu bar or right-click context menu. Zooming is to select a physical or logical module and view the inside of it which can consist of other physical or logical modules. However, the zoom feature supports a second type of zooming called “Multi zoom”.

This multi zoom is triggered when more than one module figure (lines figures are excluded) is selected upon the zoom. This option shows the insides of both levels in one diagram to supply a user with an easy overview of the software.

The logic behind these two options is so different that even in the code these two are separated into different methods. The controller methods are designed only to retrieve data from the services, but the data structure is too different for multi zoom to be combined with single zoom.

Note: When the multi zoom became more feature complete later on in the project some changes were made for support for single zoom in the future.

2.9.5 Single Zoom

The single zoom is called through the *drawSingleLevel* method in the *DrawingController*. It’s called with an array of *AbstactDTOs* which can be either *AnalysedModuleDTOs* or *ModuleDTOs*. From those modules figures are created, etc. for that level.

2.9.6 Multi Zoom

An array is the easiest way to supply details of a single level, but is hardly enough for multi zoom. More details about the DTOs are needed for multi zoom. It needs to know what the parent is of the DTO without actually asking the DTO itself. That will mean the inclusion of logic about how data is saved in the DTO. Instead the service uses the selected figure to deduct that this is the parent of the DTOs that are retrieved of the analyse or define services.

This information, the DTOs and their parents, is then saved in a Java *HashMap<String, ArrayList<AbstractDTO>>*. The string is the name of the parent figure and the *ArrayList* is the *Array* similar to what is used for single zoom. Based on the name a parent figure is created and the children are positioned into this parent figure.

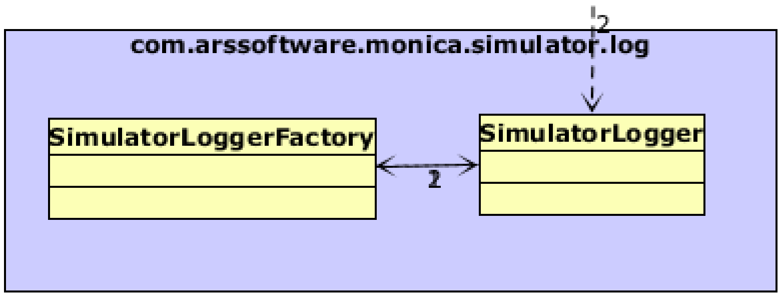


Image 2. Multizoom

2.9.6.1 *Context figures*

When zooming in on an architecture a user can select classes and other non-zoomable modules. Selected modules can be zoomed on (using context zoom), with all other modules shown as context on the next zoom level down. Unfortunately, at this time it will not show the parent of the context figure.

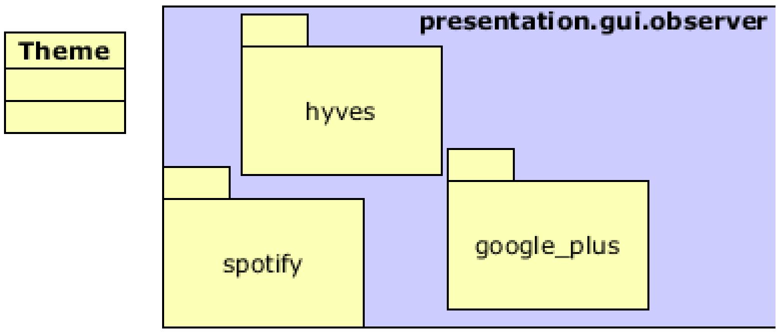


Image 3. Context figure

The Analysed and DefinedControllers manage these context figures. They know at the time of the request which figures selected and which present before the zoom. From this data they can deduce which figures need to be saved. The logic behind this is controller specific as the data is based on a different structure.

2.9.7 Context zoom

Similar to multi zoom, context zoom can be enabled by right clicking the zoom button. When using this zoom mode, all figures that were not selected are kept as they are and the figures that were are expanded as in the multi zoom. The underlying mechanics (code) functions in a way very similar to multizoom, except that all the figures that are "context" (figures that were not zoomed on) get a Boolean set on them (the 'isContext' property of *BaseFigure*) to indicate that they are.

2.9.8 FigureMap

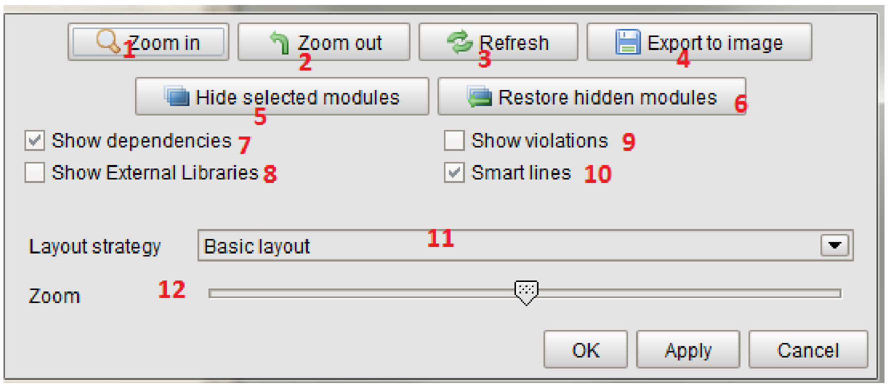
When a figure is added to the drawing the *DrawingController* first maps them to the DTOs received from the services. This is saved in the *util.FigureMap* class in several *ArrayLists*. This information needs to be saved temporarily so it can be used for the zoom in, zoom out and multi zoom actions. It is also used by the *DrawingState* which remembers the positions of the objects for each drawn level.

One can use a number of queries to get the data they need, but the most used is to use a figure to get the corresponding DTO. It also support bulk requests to get all violations, all violated figures and to clear all data in the *FigureMap*. The *FigureMap* is reset upon every request: zoom in, zoom out, refresh and the activation of show/hide dependencies/violations options.

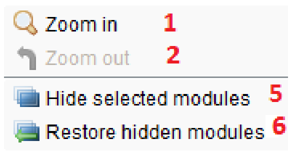
2.9.9 Settings

A user has the ability to turn on or to turn off a select number of features. The settings of these features are stored in the *DrawingSettingsController*, which in turn are saved when a HUSACCT workspace is saved.

These options can be accessed through the options dialog and a context menu, as shown in the accompanying images.



Options Dialog



Context Menu

2.9.9.1 *Show/Hide dependencies (7)*

This is on by default.

The larger an application is the more dependency lines appear. At a certain point the number of lines grows too large and it becomes unreadable. There is little to be done about this, but as more lines have to be drawn the longer you have to wait. If a user just wants to quickly zoom in to a certain level he or she can deactivate this feature and turn it on only when it is required.

This option can also be useful when the show/hide violations option is activated and the dependencies are not required for the view.

2.9.9.2 *Show/Hide violations (9)*

This is off by default.

The more rules are set the more violation lines can appear. At a certain point the number of lines grow too large and the diagram becomes unreadable. As more lines need to be drawn the longer a user has to wait for a diagram to be drawn. In case a user wants to quickly zoom in to a certain level this feature deactivated and turned on only when is required.

This option can be especially useful in combination with the show/hide dependencies option.

2.9.9.3 *Smart lines on/off (10)*

This is on by default.

The more lines need to be drawn on a physical or logical architecture the more unreadable the drawing can be become. The smart lines feature tries to fix this by doing two things. (This works for both dependency lines and violation lines. They account for each other when both are shown.) The feature is located in the Drawing class.

Prevent overlap of lines

LineFigures can go in both directions between two objects/figures. The lines will connect on the exact same points on the figures. It will become difficult to tell which dependency/violation number belongs to which line.

With the smart lines option all lines are checked for these kinds of occurrences. When this is the case their connection points will be separated. The lines will then connect a certain amount of pixels further away from each other.

Line thickness

When a lot of lines are drawn it will become difficult to tell where the biggest connection are present. Smart lines will check every line figure and increase the thickness of the line when it contains a larger amount of dependencies/violation than other checked lines. This process will unfortunately check every connected DTO of the line. This means the loading time will grow exponentially as it will have to check its details against every other line.

2.9.9.4 *Hide Selected/Restore Hidden modules (5/6)*

Test cases with large and complex applications result into a complex unreadable drawing. For this reason a user can select one or more figures and using the context menu (right click) hide the selected figures.

To restore all hidden figures a user can click the “Restore hidden modules” button from the options menu or context menu.

2.9.10 Threading

The drawing process is one of the most time consuming processes within HUSACCT. It relies on several other services, inheriting the processing time of those. In earlier builds the entire HUSACCT application would lock up when a drawing had to be generated. This was unacceptable, so the drawing process was put into a separate thread to prevent HUSACCT from locking up and allowing a user to interact with it while the drawing is generated in the background.

There are three threads:

1. *DrawingSingleLevelThread*
   - Single zoom and line drawing.
2. *DrawingMultiLevelThread*
   - Multi zoom and line drawing.
3. *DrawingLinesThread*
   - Drawing only lines. Used for the toggle of dependency lines and violation lines. The logic that is used in this thread is also used by the other two threads without starting this separate thread.

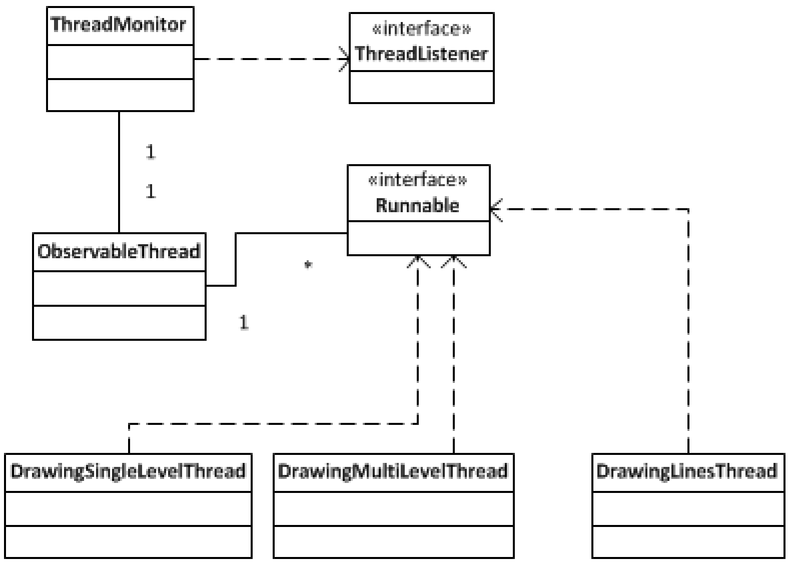


Diagram 8. Threading structure

The UIs of the Architecture Graphics service are disabled while a thread is running. However, there are instances a new request can come into the service whereupon a new thread is started. When more than one drawing thread is running they will conflict with each other and destructively leading to a corrupted image and most likely they will also cause errors, which can lead to crashes. To prevent this, a check has been built in to see if a thread is busy. When this is the case it will disregard the new thread and instead update the UI with an “Out of date” button with a refresh action attached to it.

2.10 Abstraction layer

The abstraction layer is very small as there is very little internally the service depends on. Its only component is the *FileManager* which is used for the “export to image” feature. The logic to save the image of a drawing to a file is located there. The logic to extract an image from a drawing is present in *JHotDraw* itself, which happens using *OutputFormats*.

2.11 Testing

In the HUSACCT project there are *JUnit* tests for the Graphics service which test the functionality that can be automatically tested. The *JHotDraw* components cannot be automatically tested.

3 Black Box Test

3.1 Test Data

3.1.1 Defined architecture

Data:

HUSACCT-BenchmarkApplication - https://github.com/HUSACCT/HUSACCT-BenchmarkApplication/

Layers	**Included packages**
Presentation	Presentation.*
Domain	domain.*
Infrastructure	infrastructure.*

Components in domain: modules: domain infrastructure and presentation

Packages in domain: blog facebook flickr foursquarealternative google_plus gowalla hyves language lastfm linkedin locationbased music netlog orkut pinterest shortcharacter spotify stumbleupon

Classes in flickr: Flickr FlickrPicture Tag

Defined rules:

Presentation is not allowed to use Infrastructure

3.1.2 Analyse architecture zie githubrepositories

Data:

HUSACCT-BenchmarkApplication - https://github.com/HUSACCT/HUSACCT-BenchmarkApplication/

3.2 View Defined Architecture

Precondition: A user has defined an architecture through the define User Interface and the define service.

Actions

1. In the menu bar a user clicks “Defined architecture diagram” (name subject to change.)
2. The Architecture Graphics Frame is shown for the defined architecture. It should show the defined layers.
   Data: presentation, domain, infrastructure.
   a. If these layers are mapped to analysed software units, dependency lines are shown with a number next to them representing the amount of dependencies.
   b. These dependency lines between layers should be thicker or thinner compared to other lines based on the amount of dependencies. More dependencies results in a thicker line.

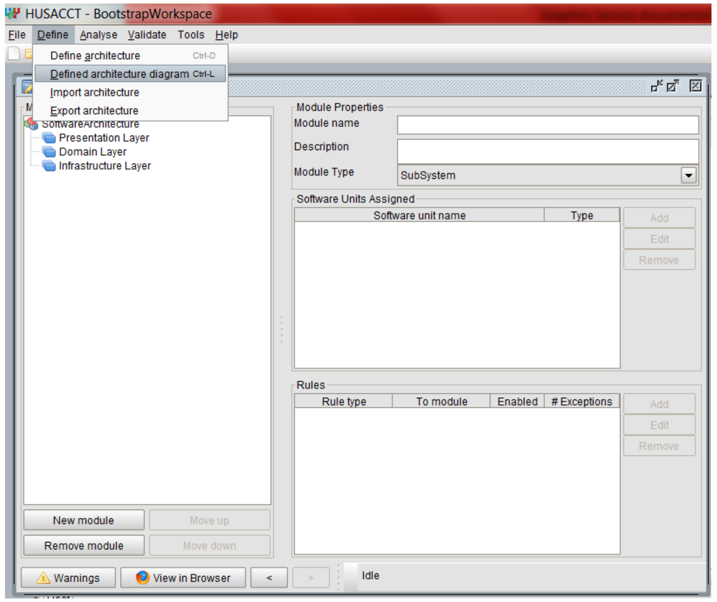


Figure1 Action 1

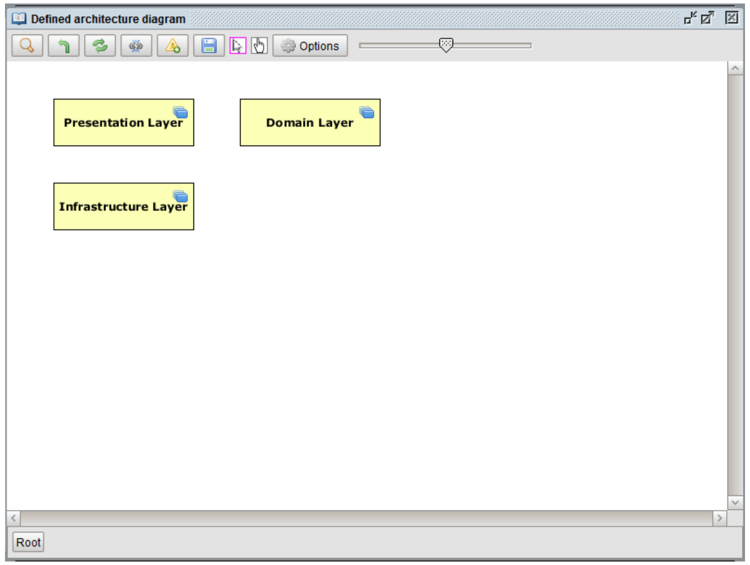


Figure2 Action 2

Expected Result	Realized Result	Satisfying?	Solution
The defined architecture is shown. This must be exactly the same as what has been defined.			

3.3 View Analyzed Architecture

Precondition: A user has analysed the source code of a selected application.

Actions:

1. A user opens the analysed architecture GUI through the menu bar by clicking “Analyse” -> “Analysed architecture diagram” (name subject to change). (Figure 3)
2. The Architecture Graphics Frame opens. (Figure 4) It should contain the analysed architecture.
   Data: husacct: .
   a. Dependency lines are shown with a number next to them representing the amount of dependencies.
   b. Optional feature: These dependency lines between layers should be thicker or thinner compared to other lines based on the amount of dependencies. More dependencies results in a thicker line.

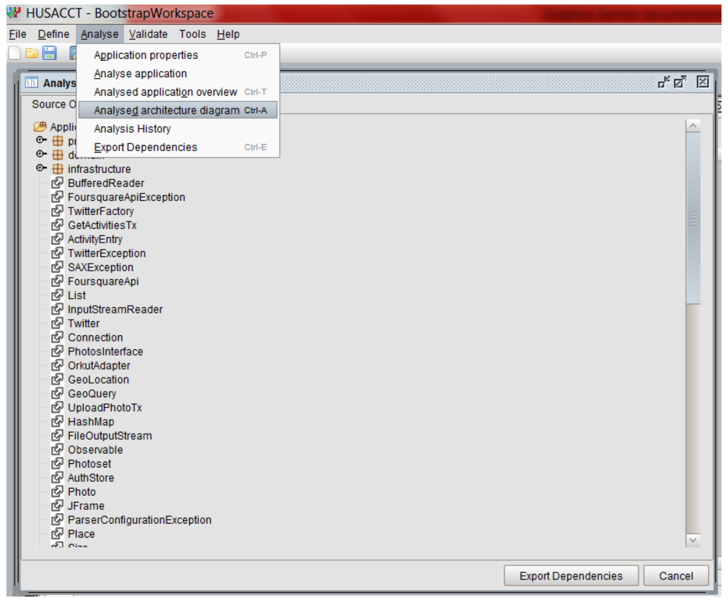


Figure 3 Action 1

Expected Result	Realized Result	Satisfying?	Solution
The defined architecture is shown. This must be exactly the same as the architecture analyzed from the source code.			

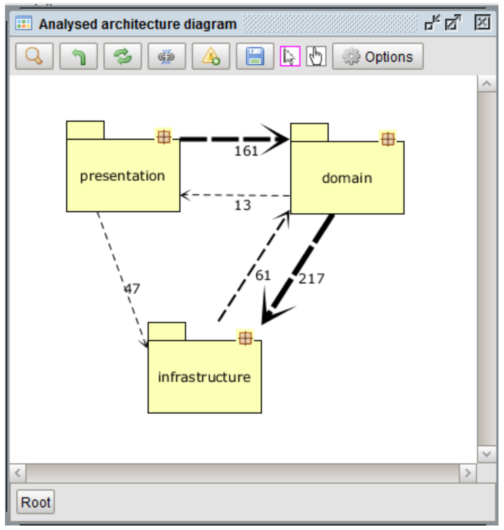


Figure 4 Action 1

3.4 Zoom on Defined Architecture

Precondition: A user has defined an architecture and has opened the logical Architecture Graphics frame.

Actions:

1. Layer zoom:
    a. A user double clicks on the layer [Data: Domain].
    b. The system should clear the drawing's existing figures, but this is not visible to the user.
    c. The system draws the following in the empty drawing:
        i. Data: modules: domain infrastructure and presentation
2. Component zoom:
    a. A user double clicks on the component [Data: domain].
    b. The system should clear the drawing's existing figures, but this is not visible to the user.
    c. The system draws the following in the empty drawing:
        i. Data: Packages: blog facebook flickr foursquarealternative google_plus gowalla hyves language lastfm linkedin locationbased music netlog orkut pinterest shortcharacter spotify stumbleupon
3. Package zoom:
    a. A user double clicks on the package [Data: flickr].
    b. The system should clear the drawing's existing figures, but this is not visible to the user.
    c. The system draws the following in the empty drawing:
        i. Data: Class: Flickr FlickrPicture Tag

Action type	Expected Result	Realized Result	Satisfying?	Solution
Layer zoom	The child architecture from the Layer is shown.			-
Component zoom	The child architecture from the Component is shown.			-
Package zoom	The child architecture from the Package is shown.			-

3.5 Zoom on Analysed Architecture

Precondition: A user has analysed an application and has opened the analysed Architecture Graphics frame.

Actions:

1. Package zoom:
    a. A user double clicks on the package [Data: domain]. (Figure 4)
    b. The system should clear the drawing's existing figures, but this is not visible to the user.
    c. The system draws the following in the empty drawing:
        i. Data: Packages: blog facebook flickr foursquarealternative google_plus gowalla hyves language lastfm linkedin locationbased music netlog orkut pinterest shortcharacter spotify stumbleupon

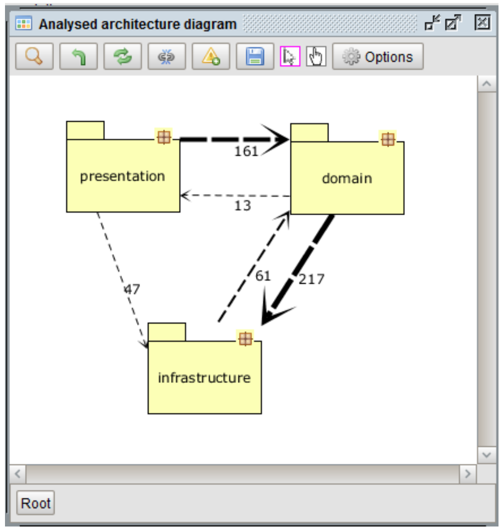


Figure 4 Action 1

Action type	Expected Result	Realized Result	Satisfying?	Solution
Package zoom	The child architecture from the Package is shown.			-

3.6 Show Violations on Defined Architecture

Precondition: A user has defined an architecture through the define User Interface and the define service, and has opened the Defined Architecture Graphics.

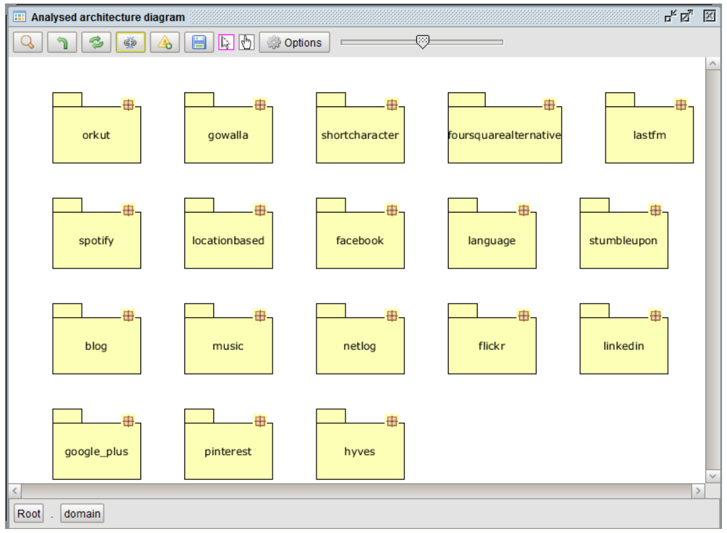


Figure4 Action 1

Actions

1. In the graphics menu bar a user clicks “Show violations” (name subject to change.)
2. The violated modules and dependencies are shown in a color.
    a. Violations in modules and dependencies are shown in a color based on the severity of a violation.(These colors can be defined by the user)
        i. Yellow: Low
        ii. Orange: Medium
        iii. Red: High
    b. The system draws the following violations:
        i. Data: violations detected and shown between the two modules. (see rules)* They are present on every zoom level.

Expected Result	Realized Result	Satisfying?	Solution
The violations of the shown modules are shown.			

3.7 Show Violations on Analysed Architecture

<u>Precondition:</u> A user has analysed an architecture through the analyse User Interface and the analyse service, and has opened the Analysed Architecture Graphics.

<u>Actions</u>

1. In the graphics menu bar a user clicks “Show violations” (name subject to change.)
2. The violated modules and dependencies are shown in a color.
    a. Violations in modules and dependencies are shown in a color based on the severity of a violation.(These colors can be defined by the user)
        i. Yellow: Low
        ii. Orange: Medium
        iii. Red: High
    b. The system draws the following violations:
        i. Data: violations between presentations and infrastructure. They are present on various physical paths are set and they are shown on every zoom level. *

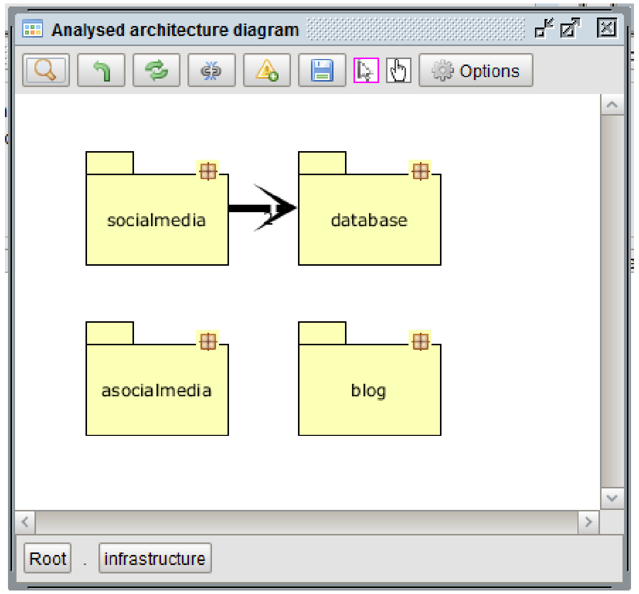


Figure5 Action 1

Expected Result	Realized Result	Satisfying?	Solution
The violations of the shown modules are shown.			

3.8 Show properties of selected figure

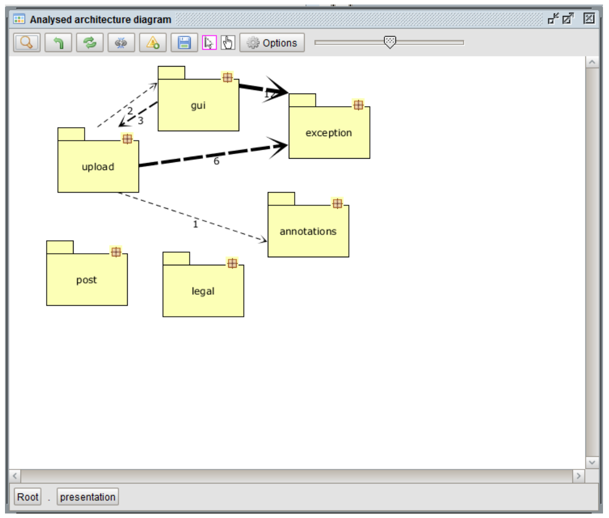


Figure6 Action 1 /\ Figure8 Action 1 \/

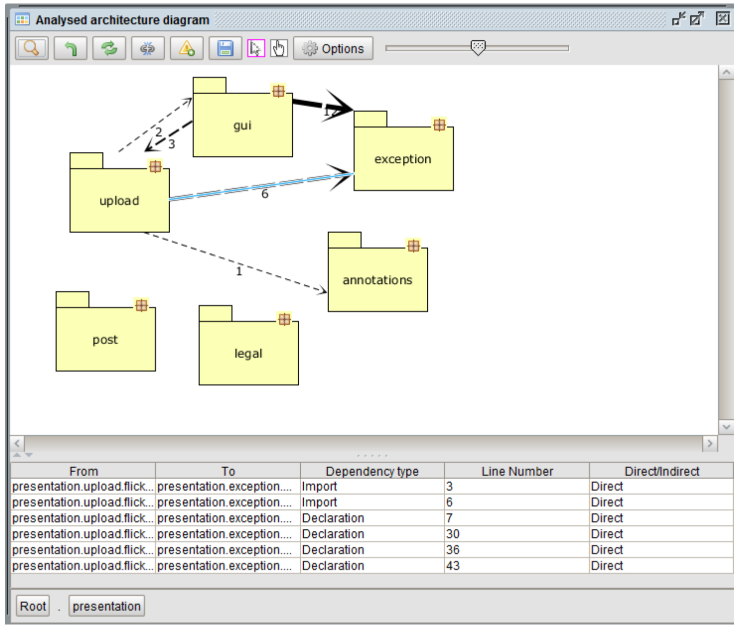
<u>Precondition:</u> A user has analysed or defined an architecture through the analyse or define User Interface and the analyse or define service, and has opened the Analysed or Defined Architecture Graphics.

<u>Actions</u>

1. A user selects a figure [Data: domain].
   a. Figure is a physical figure.
   b. Figure is a logical figure.
2. The properties view is shown
   a. When the violations are shown, these violations are shown in the

      properties of a module [Data: more than one violation is visible in the properties pane].
   b. When the dependencies are shown in the properties of a dependency [Data: more than one dependency is visible in the properties pane].

Expected Result	Realized Result	Satisfying?	Solution
The dependencies of the physical selected module are shown.			
The dependencies of the logical selected module are shown.			
The violations of the physical selected module are shown.			
The violations of the logical selected module are shown.			



From	To	Dependency type	Line Number	Direct/Indirect
presentation.upload.flick...	presentation.exception....	Import	3	Direct
presentation.upload.flick...	presentation.exception....	Import	6	Direct
presentation.upload.flick...	presentation.exception....	Declaration	7	Direct
presentation.upload.flick...	presentation.exception....	Declaration	30	Direct
presentation.upload.flick...	presentation.exception....	Declaration	36	Direct
presentation.upload.flick...	presentation.exception....	Declaration	43	Direct

3.9 Export to image

Precondition: A user has analysed or defined an architecture through the analyse or define User Interface and the analyse or define service, and has opened the Analysed or Defined Architecture Graphics.

Actions

1. In the graphics menu bar a user clicks "Export to image" (name subject to change.)
2. The Export to Image Frame is shown for the shown architecture. It should show the documents folder.
3. A user selects an folder to save the file image in, defines a "File Name" and clicks "Save" [Data: ~/Desktop/export-graphics.png].

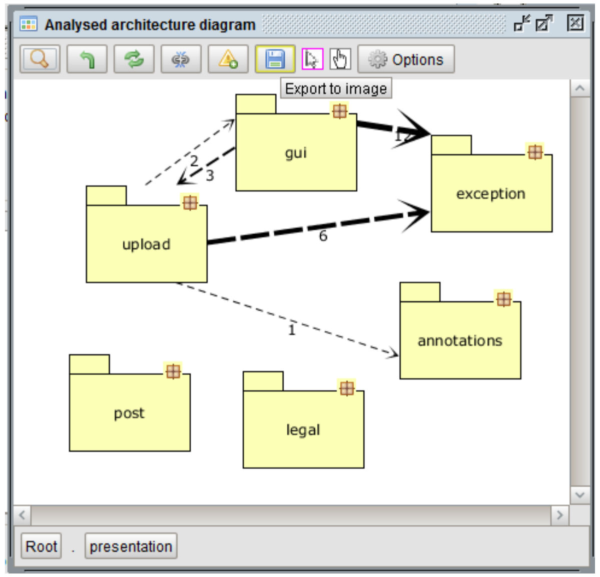
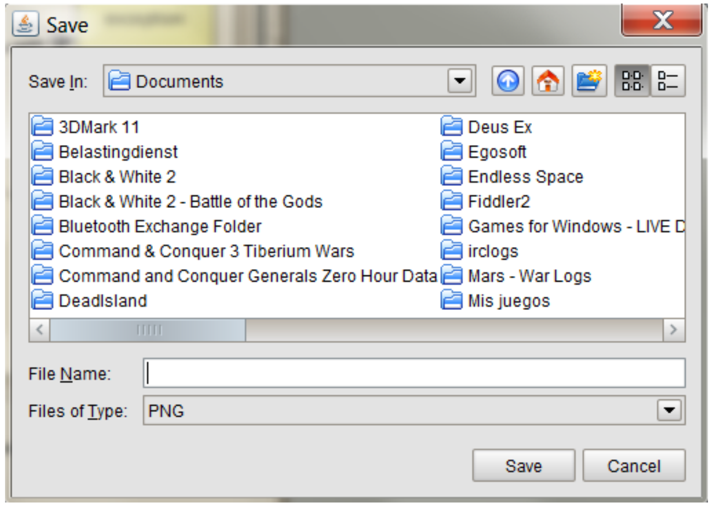


Figure7 Action 1


Save

Save In: Documents

3DMark 11	Deus Ex
Belastingdienst	Egosoft
Black & White 2	Endless Space
Black & White 2 - Battle of the Gods	Fiddler2
Bluetooth Exchange Folder	Games for Windows - LIVE D
Command & Conquer 3 Tiberium Wars	irclogs
Command and Conquer Generals Zero Hour Data	Mars - War Logs
DeadIsland	Mis juegos

File Name:

Files of Type: PNG

Save Cancel

Figure8 Action 2

Expected Result	Realized Result	Satisfying?	Solution
The image is saved in the defined folder with the defined file name.			

4

5 White Box Tests

In the HUSACCT project itself several JUnit tests are included. Much of the Graphics Service cannot be tested however, as it is all part of the JHotDraw library which can only be tested manually.

5.1 Improvements

5.1.1 Presentation

5.1.1.1 *ParentFigure*

The *ParentFigure* object is a container that can container other Figures and automatically sorts them. Because of the way *JHotDraw* is implemented adding Figures to just this container doesn't work / makes it impossible to select the child figures. To circumvent this, the figures contained within this parent container are added to both the drawing and the *ParentFigure*. This should be corrected: Adding to the *ParentFigure* will automatically properly render the child figures.

5.1.1.2 *Drawing contains user interface interaction*

The *Drawing* class contains code to save the contents of the drawing to disk. This is user interface code does not belong in the Drawing. This code should ideally be moved to one of the Controller classes during the next re-factor iteration.

5.1.1.3 *Decorators*

The HUSACCT currently uses decorators to apply specific styles to the rendered figures. However, due to constant refactoring the decorators are in a semi-used state. A decision has to be made to either fully remove the decorators and come up with a more stable solution or the decorator pattern should be properly re-implemented.

5.1.2 Task

5.1.2.1 *Wrong dependency count, define diagram*

The amount of dependencies in the logical diagram is not accurate to the real scenario. It can be easily compare with the physical diagram. The dependency count is based on the mapping logic of the define service. They return physical paths that do not reflect the mapping. Instead they return the mapped physical module and all of its children. This structure was not agreed upon and is the cause of the wrong dependency count. This problem was found when they changed the structure of the data very late in the project.

5.1.2.2 *Validate checkConformance call*

Checking the need to call the *checkConformance* on the validate service before showing violations. We hope, but have not been able to test this, that the team responsible for this service has internally checked if they have validated yet.

5.1.2.3 *Level-by-level requests*

No longer using a level-by-level request system on other services. This causes major performance issues for other services. This occurs when the graphics service needs to request data from a large amount of combinations of DTOs.

5.1.2.4 *Zooming*
The single and multi zoom could be combined in one combined logical method. There was a split during development that wasn't refactored as it would take too much time during the project.

Better figure context support. It's too unstable now, because it had to be implemented at the last moment. It will remember *context figures* for the multi-zoom, but a refresh or a zoom in even deeper will clear this context.

When classes/interfaces are selected for multi zoom from within a multi zoom they are considered context and their parent package is not shown a level deeper. These classes should be shown in a parent container.

When multi zoom is applied it might be beneficial to always show modules in their parent and even those in their parent figures.

5.1.2.5 *Violation line color*
To understand which color to use for a violation line the system has to go through every violation DTO and see which has the highest severity. From there it remembers to color to use for that violation line. If there was some order in those DTOs it would not be necessary to check all the DTOs. Logic about which severity (int) is the highest should also not be present in the graphics service. Instead it should somehow be set in the validate service and communicated back to graphics.

5.1.2.6 *Threading*
Support for multi threads to draw simultaneously without conflicting with each other. Another option is an alternative catch system that will only draw from the latest created thread stopping running ones.

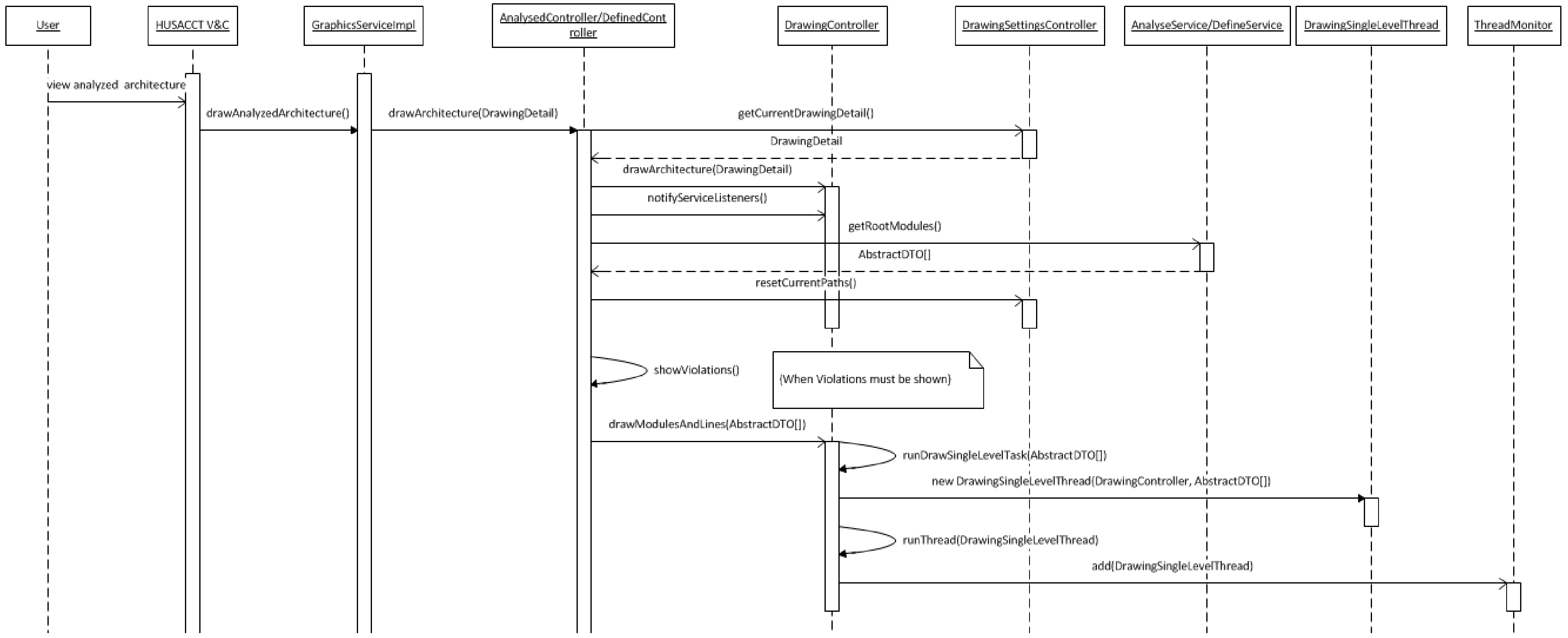
5.1.2.7 *Multizoom*
The current multizoom functionality has been organically developed during the 2012 phase of HUSACCT from nothing into a full feature. The first draft of the zoom system only allowed zooming in on a single figure. This proved to be of limited use, so another solution was implemented.

Multizoom (and by extension, context zoom) allows the user to zoom in on multiple objects, even if those object are non-zoomable. If they are not zoomable, they are simply carried over into the next zoom state unmodified (as with context figures in context zoom).

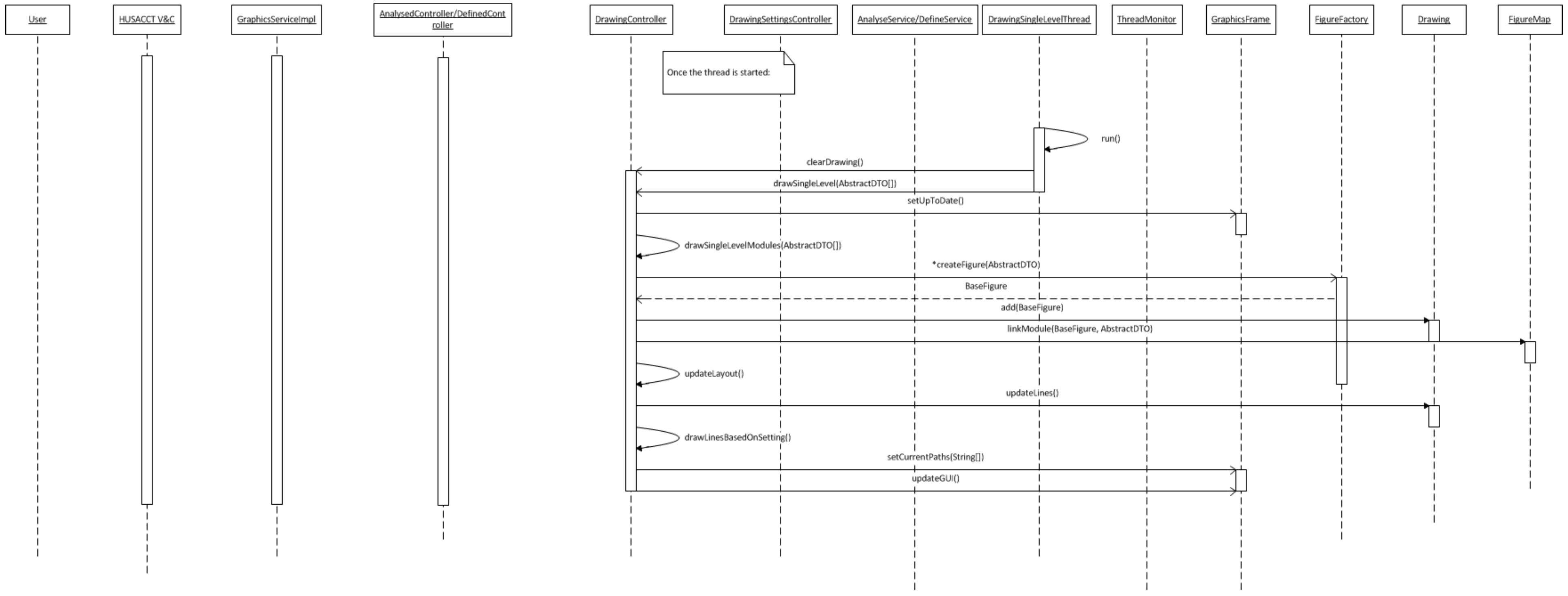
Multizoom currently achieves this goal by keeping a list of strings which contains the identifies all the figures that zooming needs to be applied to. This can be improved significantly by replacing this with a collection of DTO's or figures, for speedier access and a more transparent structure.

6 Attachment – Sequence diagrams

6.1 View Analyzed and Defined Architecture – Part 1

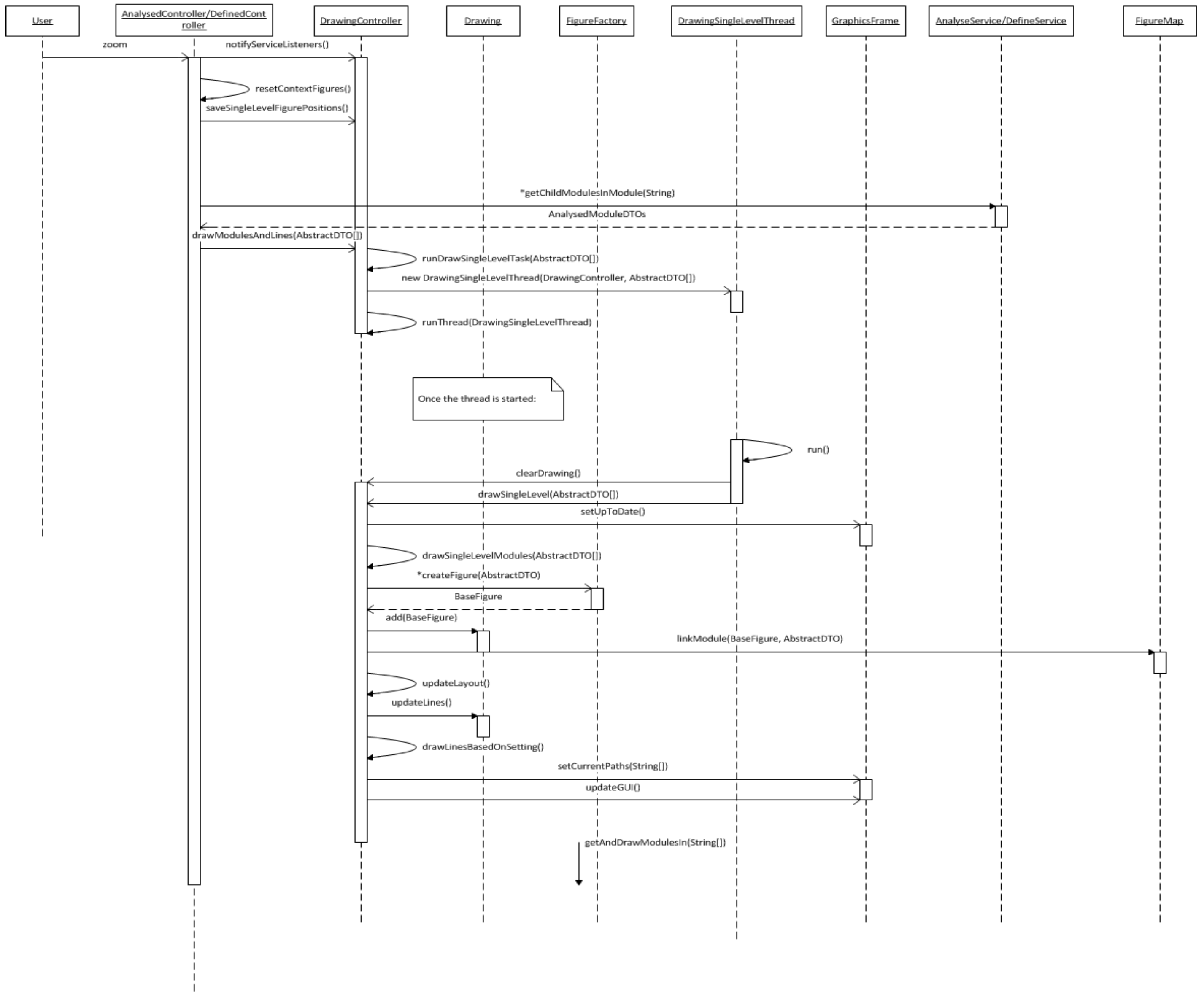

User
HUSACCT V&C
GraphicsServiceImpl
AnalysedController/DefinedController
DrawingController
DrawingSettingsController
AnalyseService/DefineService
DrawingSingleLevelThread
ThreadMonitor
view analyzed architecture
drawAnalyzedArchitecture()
drawArchitecture(DrawingDetail)
getCurrentDrawingDetail()
DrawingDetail
drawArchitecture(DrawingDetail)
notifyServiceListeners()
getRootModules()
AbstractDTO[]
resetCurrentPaths()
showViolations()
{When Violations must be shown}
drawModulesAndLines(AbstractDTO[])
runDrawSingleLevelTask(AbstractDTO[])
new DrawingSingleLevelThread(DrawingController, AbstractDTO[])
runThread(DrawingSingleLevelThread)
add(DrawingSingleLevelThread)

6.2 View Analyzed and Defined Architecture – Part 2 – Thread



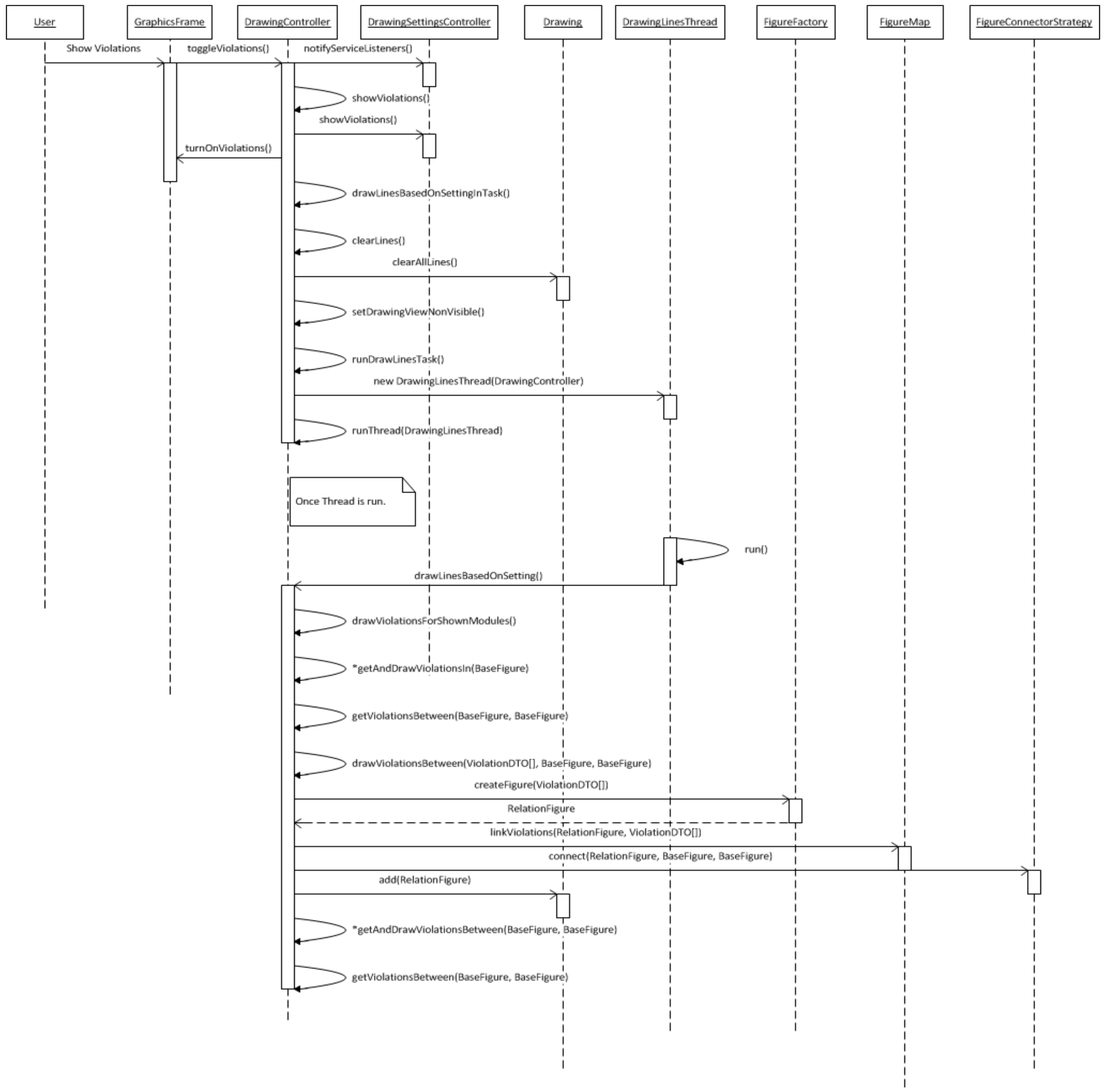
User
HUSACCT V&C
GraphicsServiceImpl
AnalysedController/DefinedController
DrawingController
DrawingSettingsController
AnalyseService/DefineService
DrawingSingleLevelThread
ThreadMonitor
GraphicsFrame
FigureFactory
Drawing
FigureMap
Once the thread is started:
run()
clearDrawing()
drawSingleLevel(AbstractDTO[])
setUpToDate()
drawSingleLevelModules(AbstractDTO[])
*createFigure(AbstractDTO)
BaseFigure
add(BaseFigure)
linkModule(BaseFigure, AbstractDTO)
updateLayout()
updateLines()
drawLinesBasedOnSetting()
setCurrentPaths(String[])
updateGUI()

6.3 Zoom on Architecture

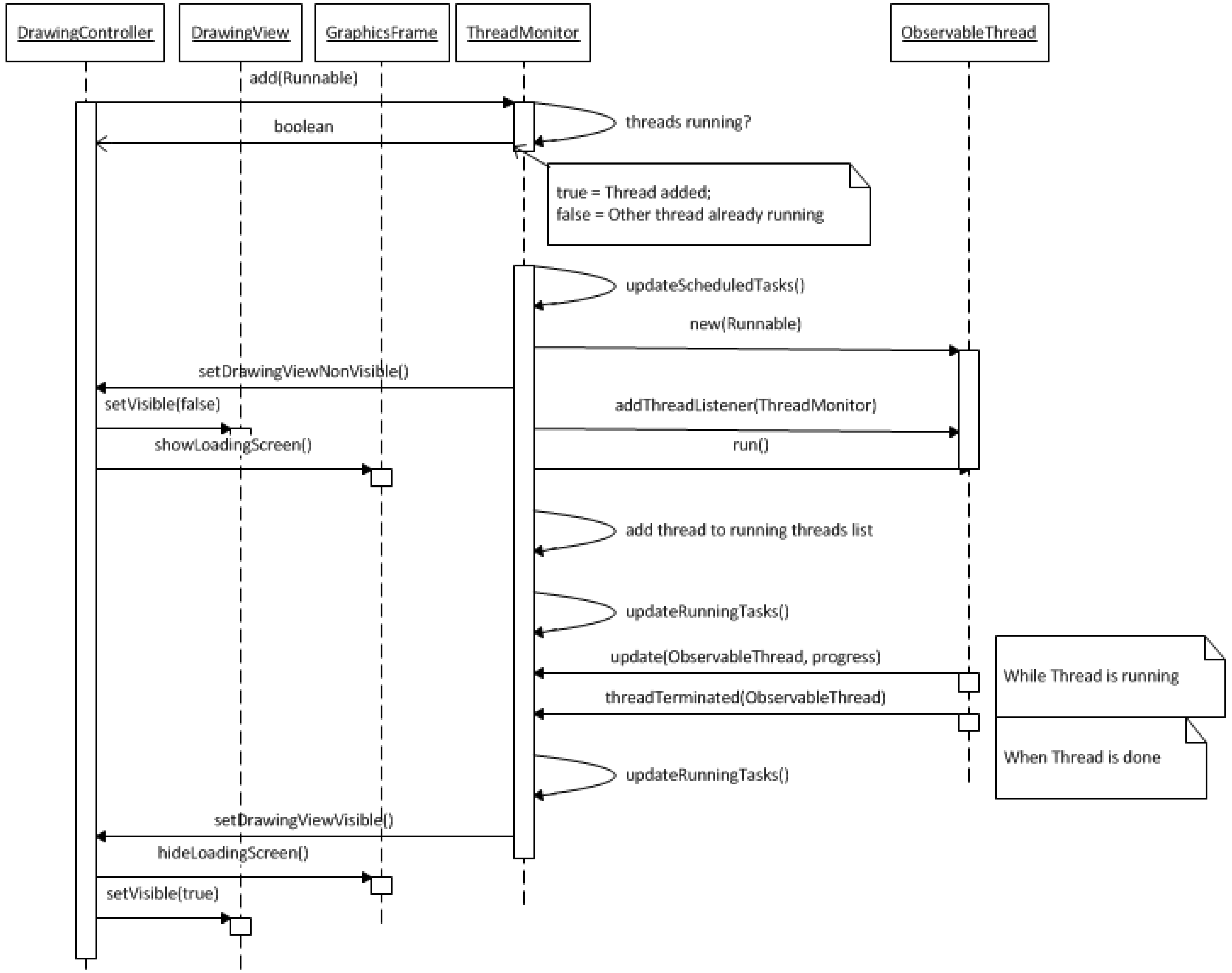


User
AnalysedController/DefinedController
DrawingController
Drawing
FigureFactory
DrawingSingleLevelThread
GraphicsFrame
AnalyseService/DefineService
FigureMap
zoom
notifyServiceListeners()
resetContextFigures()
saveSingleLevelFigurePositions()
*getChildModulesInModule(String)
AnalysedModuleDTOs
drawModulesAndLines(AbstractDTO[])
runDrawSingleLevelTask(AbstractDTO[])
new DrawingSingleLevelThread(DrawingController, AbstractDTO[])
runThread(DrawingSingleLevelThread)
Once the thread is started:
run()
clearDrawing()
drawSingleLevel(AbstractDTO[])
setUpToDate()
drawSingleLevelModules(AbstractDTO[])
*createFigure(AbstractDTO)
BaseFigure
add(BaseFigure)
linkModule(BaseFigure, AbstractDTO)
updateLayout()
updateLines()
drawLinesBasedOnSetting()
setCurrentPaths(String[])
updateGUI()
getAndDrawModulesIn(String[])

6.4 Show Violations

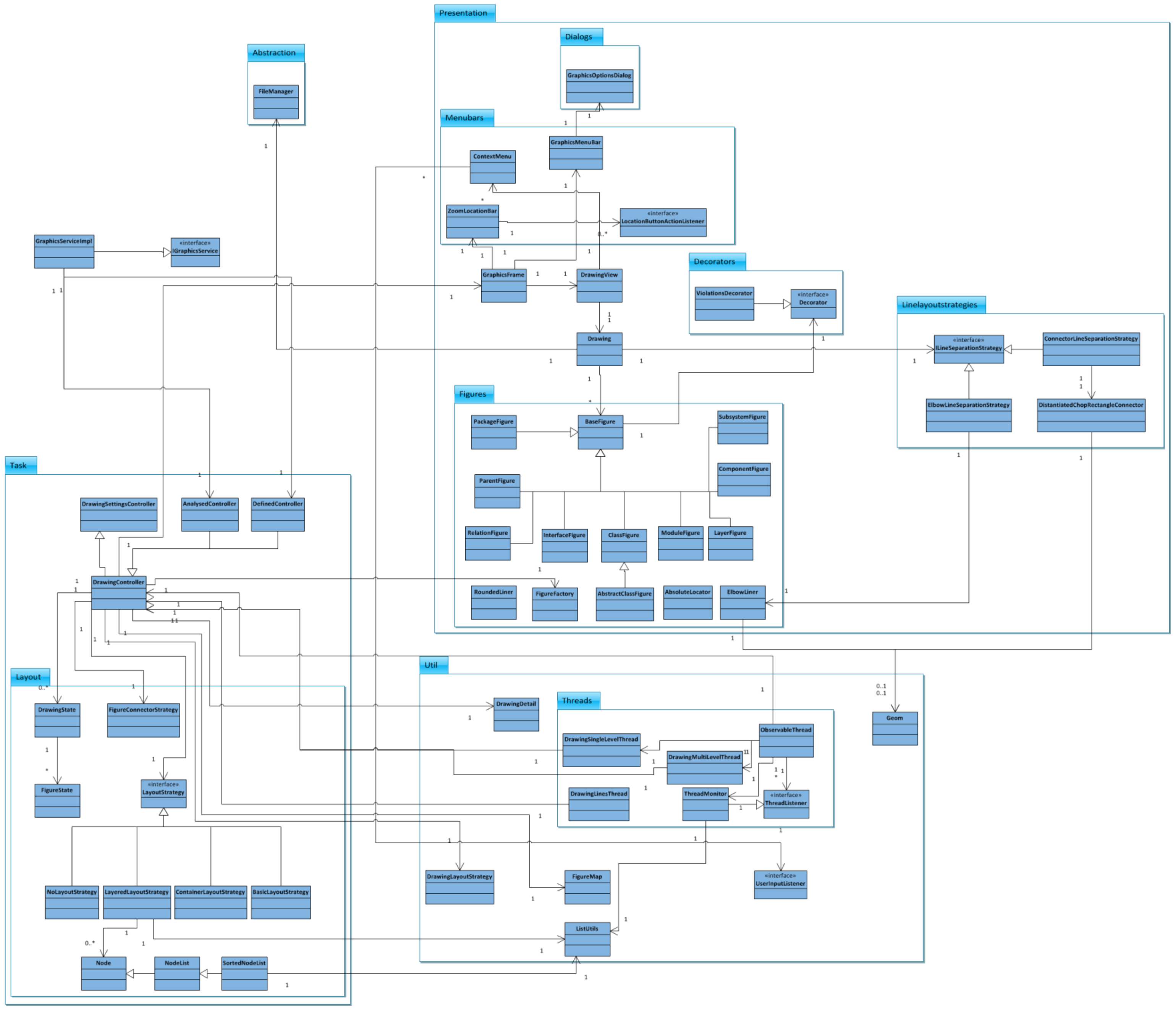

User
GraphicsFrame
DrawingController
DrawingSettingsController
Drawing
DrawingLinesThread
FigureFactory
FigureMap
FigureConnectorStrategy
Show Violations
toggleViolations()
notifyServiceListeners()
showViolations()
showViolations()
turnOnViolations()
drawLinesBasedOnSettingInTask()
clearLines()
clearAllLines()
setDrawingViewNonVisible()
runDrawLinesTask()
new DrawingLinesThread(DrawingController)
runThread(DrawingLinesThread)
Once Thread is run.
run()
drawLinesBasedOnSetting()
drawViolationsForShownModules()
*getAndDrawViolationsIn(BaseFigure)
getViolationsBetween(BaseFigure, BaseFigure)
drawViolationsBetween(ViolationDTO[], BaseFigure, BaseFigure)
createFigure(ViolationDTO[])
RelationFigure
linkViolations(RelationFigure, ViolationDTO[])
connect(RelationFigure, BaseFigure, BaseFigure)
add(RelationFigure)
*getAndDrawViolationsBetween(BaseFigure, BaseFigure)
getViolationsBetween(BaseFigure, BaseFigure)

6.5 Threading



7 Attachment – Class diagram

7.1 Class diagram


Abstraction
FileManager
Presentation
Dialogs
GraphicsOptionsDialog
Menubars
ContextMenu
GraphicsMenuBar
ZoomLocationBar
«interface» LocationButtonActionListener
GraphicsFrame
DrawingView
Decorators
ViolationsDecorator
«interface» Decorator
Drawing
Linelayoutstrategies
«interface» ILineSeparationStrategy
ConnectorLineSeparationStrategy
ElbowLineSeparationStrategy
DistantiatedChopRectangleConnector
Figures
PackageFigure
BaseFigure
SubsystemFigure
ComponentFigure
ParentFigure
RelationFigure
InterfaceFigure
ClassFigure
ModuleFigure
LayerFigure
RoundedLiner
FigureFactory
AbstractClassFigure
AbsoluteLocator
ElbowLiner
GraphicsServiceImpl
«interface» IGraphicsService
Task
DrawingSettingsController
AnalysedController
DefinedController
DrawingController
Layout
DrawingState
FigureConnectorStrategy
FigureState
«interface» LayoutStrategy
NoLayoutStrategy
LayeredLayoutStrategy
ContainerLayoutStrategy
BasicLayoutStrategy
Node
NodeList
SortedNodeList
Util
DrawingDetail
Threads
DrawingSingleLevelThread
DrawingMultiLevelThread
ObservableThread
DrawingLinesThread
ThreadMonitor
«interface» ThreadListener
Geom
DrawingLayoutStrategy
FigureMap
«interface» UserInputListener
ListUtils