|  |
| --- |
| Hogeschool Utrecht |
| Graphics System documentation |
| Architecture Graphics Service |

|  |
| --- |
| **Team Architecture Graphics 2012 (version 1.0)**  Tom de Bruijn, Johan van der Slikke, Guido van Tricht, Patrick van der Willik  **Team Architecture Graphics 2013 (version 2.0)**  Stan Verhoeckx, Dennis Bullee, Tim Bazuin, Ruwan Kellaert, Mark Wallenburg  **Team Architecture Graphics & SAR 2016 (version 5.0 -5.1)**  Lars Celie, Gert Kommer, Rory van Ligten  Leo Pruijt (since version 2.1)  November 2016 |

**Contents**

[1 Introduction 3](#_Toc466470563)

[2 Functionality 3](#_Toc466470564)

[2.1 Draw Architecture at Root Level 3](#_Toc466470565)

[2.2 Zoom In, Zoom Out 3](#_Toc466470566)

[2.3 Edit Properties and Refresh 3](#_Toc466470567)

[2.4 Export Diagram 3](#_Toc466470568)

[3 Decisions and justification 4](#_Toc466470569)

[4 Software Partitioning 4](#_Toc466470570)

[4.1 Context 4](#_Toc466470571)

[4.2 Layers 6](#_Toc466470572)

[5 JHotDraw drawing 7](#_Toc466470573)

[5.1 Setup JHotDraw 7](#_Toc466470574)

[5.2 Implementation in HUSACCT 8](#_Toc466470575)

[6 User interface 10](#_Toc466470576)

[7 Drawing Task 12](#_Toc466470577)

[7.1 DrawingController 12](#_Toc466470578)

[7.2 Subtype Controllers - Analysed and Defined Controllers 12](#_Toc466470579)

[7.3 Zooming 12](#_Toc466470580)

[7.4 Diagram Options 13](#_Toc466470581)

[8 Abstraction layer 14](#_Toc466470582)

[9 Testing 15](#_Toc466470583)

[9.1 Manual Tests – Black Box 15](#_Toc466470584)

[9.2 Automated Tests - White Box 15](#_Toc466470585)

[10 Module and Rule diagram 16](#_Toc466470586)

[10.1 How to Activate the Module and Rule diagram? 16](#_Toc466470587)

[10.2 User Documentation 16](#_Toc466470588)

[10.3 Points for improvement 17](#_Toc466470589)

# 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, visualize 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.

# Functionality

The functionality is described and illustrated in the User Manual.

## Draw Architecture at Root Level

When starting-up a diagram, by default a diagram is drawn of the modules and their relations at root level.

## Zoom In, Zoom Out

Several options are provided to zoom in, or zoom out:

* Zoom In (default)
* Zoom In with Context
* Multi Zoom In
* Zoom In on a path in the decomposition bar, at the bottom of the diagram editor
* Zoom Out

## Edit Properties and Refresh

Several options may be selected, whereafter the diagram is refreshed. E.g.:

* Show Dependencies
* Show Violations
* Show External Libraries
* Proportional Line Width
* Layout Strategy
* Dependency type

## Export Diagram

An image of the shown architecture can be saved at a defined location.

# Decisions and justification

|  |  |
| --- | --- |
| Decision | Justification |
| We 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. This library can be used in Windows, Linux and Mac. | NF1  NF2 |
| We use *JInternalFrames* to show our graphics view.  This decision was made by the Control team. | NF3 |
| We’ve separated presentation, task, and domain logic into separate layers to enable maintainability and portability (Swing and JHotDraw). | NF2 |
| 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 or defined architecture of an application. | NF3 |
| 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 specific communication with the analyse, define and validate services. |  |
| 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 domain 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 |
| A *UserInputListener* is implemented into the controllers of the Presentation layer to handle requests. 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 inform us of any changes in their components. Different actions are performed after an update, dependent on the triggering component. | 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 SwingWorker threads in order to keep the application responsive during CPU intensive work and to prevent synchronization problems. | NF4 / NF7 |
| We’re providing multi-input methods for commands in order to provide a user with a rich user experience | NF4 |
| 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. |  |

# Software Partitioning

## Context

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 graphics service manages the other views of the service in the service itself.

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

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

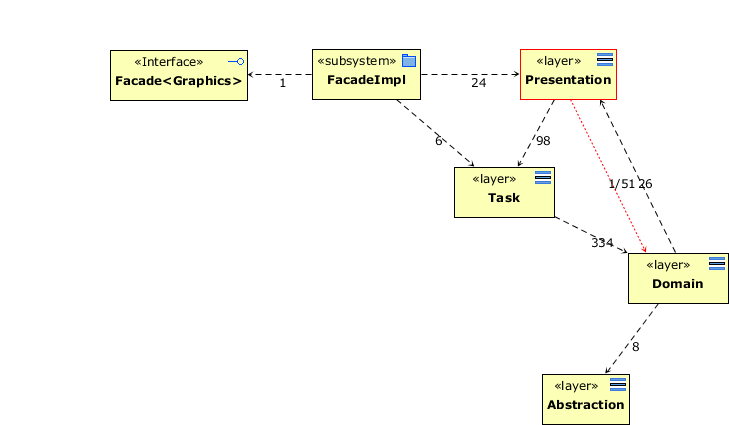
The Control Service is used for selecting the language of the application, and to tell other services when to update their view based on the changes made in the Graphics Service.



## Layers

The HUSACCT wide software architecture separates each components into three layers: presentation, task and domain. An abstraction layer is added.

* Presentation layer is Swing dependent, while Task and Domain are not.
* Task layer knows how to create a drawing.
* Domain keeps trach of the elements in the drawing and abstracts the *JHotDraw* dependent technology, as explained in the following subsection.
* Abstraction contains the “export to image” feature only, currently.



In version 4.4.1 the architecture of the graphics component has been restructured. This was necessary since exceptions were thrown quite often, due to thread synchronization problems. Furthermore, control of the presentation and of the drawing domain was mixed up to a great extent and the control classes had become too complex. Since version 4.4.1 layers Presentation, Task, and Domain are strictly separated. However, two exceptions apply, since DrawingView, included in Domain, has both domain and presentation capabilities. Consequently, DrawingView is allowed to make use of event handlers of Presentation (back calls), and Presentation is allowed to make use of DrawingView (not of Drawing, which explains the single violation).

To prevent thread synchronization problems, between threads creating the drawing and the Swing dispatch thread that handles the user interface events, the usage of DrawingView is regulated between Presentation and Task:

* A drawing task trigger, first detaches a DrawingView from the Presentation, then creates a new Drawing, and finally attaches DrawingView to Presentation again.
* SwingWorker is used to create separate threads for the drawing tasks.   
  See also: <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>
* DrawingView, including a drawing, is detached and attached to prevent thread synchronization problems.

# 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.

## 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.

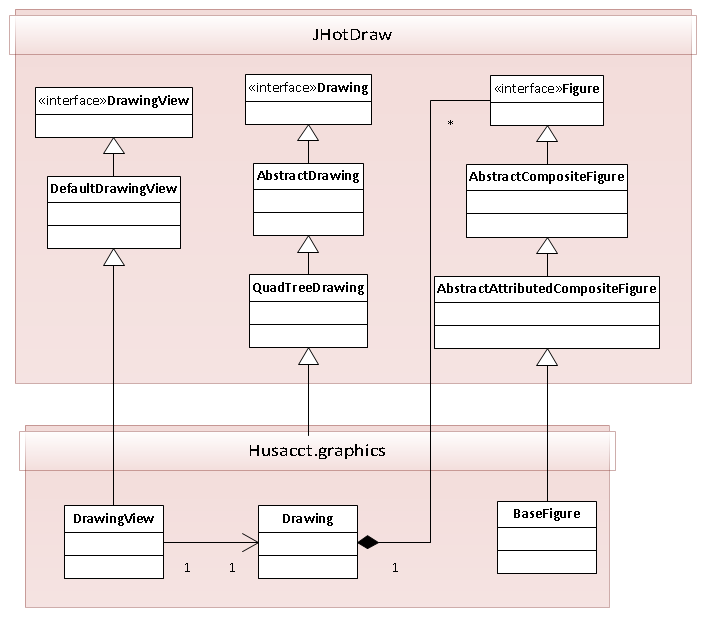


Diagram JHotDraw implementation HUSACCT

*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.

## Implementation in HUSACCT

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

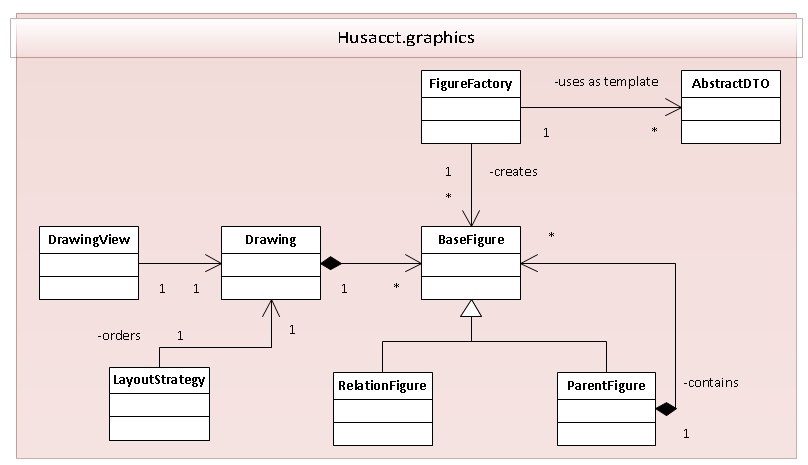


Diagram JHotDraw implementation HUSACCT

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.

### 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.

### 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.

# 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 GraphicsPresentationController for further processing. Most of the logic concerning event handling is done by the UI classes, generating more specific events to notify the GraphicsPresentationController.

The GraphicsPresentationController is therefore not concerned with handling the low-level Click events. Instead it concerns itself with high-level actions, for example zooming, or hiding figures.

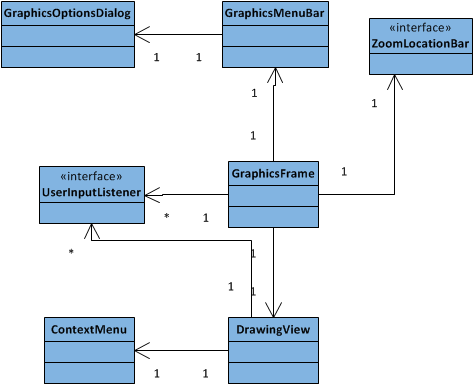


Diagram User interface widgets

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.

# Drawing Task

The Task and domain layer contain all the logic for the Graphics service main features. They are controlled from the diagram-independant main controller, *DrawingController* and diagram specific subtype controllers like AnalysedController and DefinedController.

## DrawingController

DrawingController has the responsibility to create the top-level architecture, zoom-in, zoom-out, etc. Functionality is triggered through the GraphicsPresentationController, like zoom in, zoom out, refresh, toggle dependencies and toggle violations. These and more features are described in the chapter below.

## Subtype Controllers - Analysed and Defined Controllers

Diagram specific controllers extend the abstract DrawingController. It is their responsibility to create figures (ModuleFigures and RelationFigures) specific for a type of diagram based on data received from the other HUSACCT-components. The subtype controllers have no knowledge of the actual drawing and the drawing process; they only retrieve data from the services and create diagram figures on request.

A subtype controller requests data from other services. The AnalysedController is specific for the implemented architecture diagram and it requests data from the Analyse service and the Validate service. DefinedController is responsible for the intended architecture diagram and it requests data from the Define service, the Analyse service and the Validate service.

### 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 subtype 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 create figures.  
*The Drawing (JHotDraw) and the JInternalFrame (GraphicsFrame) can be easily reused.*

## Zooming

The graphics service supports several kinds of zooming: single-zoom and multi-zoom.

The most simple version, single- zoom, is to select a physical or logical module and view the inside of it, which can consist of other physical or logical modules.

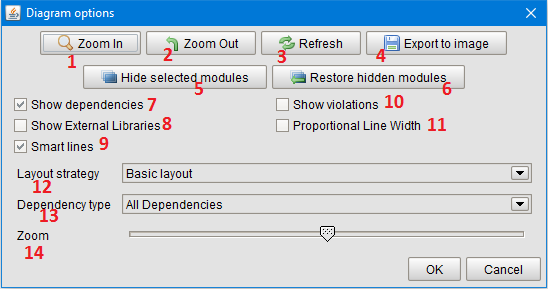
This multi-zoom is triggered when more than one module figure (lines figures are excluded) is selected upon the zoom. This option shows the insides of both levels in one diagram to supply a user with an easy overview of the software.

Both zoom options are activated by: 1) double clicking on a figure in the diagram (single zoom);   
2) selecting one figure (single-zoom) or several (multi-zoom) and pressing the zoom button in the menu bar; 3) or in the options dialogue; or 4) right-click context menu.

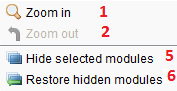
## Diagram Options

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

#### 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.

#### Show/Hide violations (10)

*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.

#### Smart lines on/off (9)

*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.

#### 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.

#### Change dependency type (13)

To switch between the different dependency types in the implemented architecture diagram the user can choose between “all dependencies”, “Access Call and references” and “Associations as in class diagrams”.

# 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*.

# Testing

## Manual Tests – Black Box

Most testing needs to be performed manually.  
Good resources for testing are: 1) The SRMA test code and workspace (including examples of all module and rule types); 2) HUSACCT code and (current) workspace.

In the table below some points of attention.

|  |  |
| --- | --- |
| **Test Action** | **Attention** |
| Draw diagram   * Implemented * Intended * Without violation * With violations | 1) Correctness of number of drawn modules and relations:   * Analysed modules versus Code * Defined modules versus defined architecture * Dependency relations versus info in Analysis Overview. * Violation relations versus info in Validate View.   2) Correctness of the layout  3) Correctness of adjusting/redrawing |
| Zoom-in, Zoom-out  Multiple Zoom  Zoom with context | 1) Correctness as above  2) Parent figures drawn  3) Names of (parent) figures  4) Path at bottom of editor |
| Select option   * Via tool bar * Via options dialogue |  |
| Select relation   * Dependency * Violation |  |
| Export to diagram |  |

## Automated Tests - White Box

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

# Module and Rule diagram

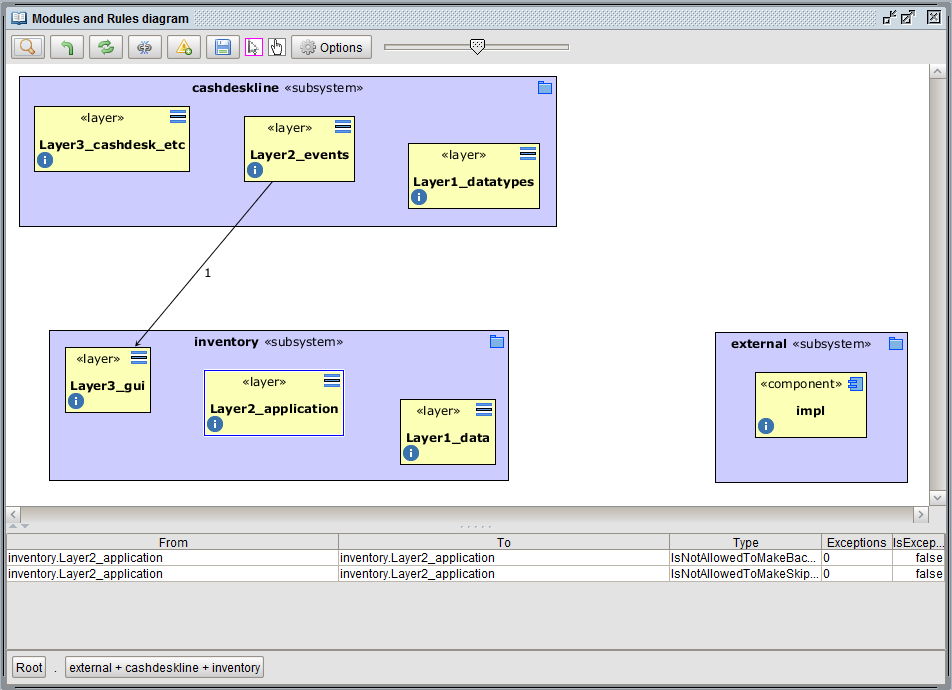
In HUSACCT version 5.0 an initial version of a module and rule diagram was incorporated, which was further improved in version 5.1. However, the added value of this type of diagram is very limited. Since it is complicating the menu structure and likely to confuse (student) users of the tool, the diagram type is not activated within version 5.1.

## How to Activate the Module and Rule diagram?

1. Include a diagram menu with two diagram items in the Defined menu.
   1. Edit class husacct.control.presentation.menubar. DefineMenu.java
   2. Enable JMenu architectureDiagram and JMenuItem moduleAndRuleDiagramItem and their usage within the class.
   3. Compare with previous version(s). See the commit of gertk at 2016-04-20.
2. Include a diagram menu with two diagram items in the Toolbar menu
   1. Edit class husacct.control.presentation.toolbar.ToolBar
   2. Compare with previous version(s).
3. Initialize the Modules and Rules diagram and activate it on user request.
   1. Edit class husacct.control.presentation.viewcontrol.ViewController
   2. Compare with previous version(s). Edit at four locations in the code.
4. Edit the text of the definedArchitectureDiagramContainer (within class ViewController) to “ModuleAndDependenciesDiagram”.

## User Documentation

The user documentation below (version 06-2016) is included in case the diagram type is activated again.

The modules and rules diagram shows the modules defined in the intended architecture and the rules assigned to these modules. The rules will be shown in two different ways.

A rule is shown as an arrow between two modules in the diagram if these modules are incorporated in the rule (as to-module and from-module). The number attached to the arrow represents the number of rules that constrain usage from one module to another.

The other option to indicate a rule is an icon on a module; used for the types of rules that are assigned to one module only such as rules of type facade convention, naming convention or visibility convention.

When a module is selected (in this case Inventory.Layer2\_application is selected), the assigned rules of the module are shown with their details.

### To be edited in user manual

Paragraph numbers are from version 5.1.

1.1.2 4b Changed title to modules and dependencies diagram

1.1.2 4c Added modules and rules diagram

3.3 Changed the image because of new diagram title

3.3 Changed the chapter structure from one big chapter to support the added renamed and added diagrams. The new subchapters are: 3.3.1 modules and dependencies diagram, 3.3.2 modules and rules diagram, 3.3.3 diagram functionality

4.5.1 Added chapter about the different types of dependencies and how they are shown when UML link is selected.

4.5.3 Changed image and

4.5.3 Added dependency type explanation

## Points for improvement

The diagram should be developed further to really add value.

1. Logical layout of the diagram  
   E.g. by menas of a layered layout.
2. Indicating all relation rules in the diagram
   1. E.g. also for Façade convention, Back call ban and Skip call ban
   2. Maybe only to be shown for a selected module
3. Be able to switch between views
   1. Current view: all usages are allowed, except when a rule forbids usage.
   2. ~UML Component diagram: no usage is allowed, except when indicated by a dependency relation.