

HUSACCT

Software Architecture
Reconstruction and
Compliance Checking Tool

USER Manual

November, 2016

INTRODUCTION

HUSACCT provides support for software architecture reconstruction (SAR) and software architecture compliance checking (SACC) with respect to the modular architecture of Java and C# systems. SACC functionality monitors the conformance of the implemented architecture (in the source code) to the intended software architecture (in the design). Basic SAR functionality helps to get an understanding of the implemented architecture, e.g. by browsing or visualizing the software units (packages, namespaces, classes) with their dependencies. More advanced SAR functionality aids the reconstruction of the system's intended architecture in terms of logical modules such as layers, components and subsystems.

The prominent feature of HUSACCT is the support of semantically rich modular architectures (SRMAs)[1], which are composed of modules of different types (like subsystems, layers, components and external systems) and rules of different types. To perform an SACC, first an intended software architecture has to be defined which describes the modules and their types, the rules (constraints), and the assignment of implemented software units to the modules. Next, HUSACCT checks the compliance to the defined rules, based on static analysis of the source code, and it reports violations to the rules.

To download the build or watch an introduction video, visit: <http://husacct.github.io/HUSACCT/>

License and Warranty

HUSACCT is free-to-use open-source software. You can make use of HUSACCT free of charge under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. You can redistribute the software and/or modify it for your own use, but you are not allowed to include the software, parts of the software or documentation, in other products (for commercial or non-commercial use).

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU Affero General Public License for more details: <http://www.gnu.org/licenses/>.

HUSACCT Development

HUSACCT means: HU Software Architecture Compliance Checking Tool, where HU stands for "HU University of Applied Sciences Utrecht", which is located in Utrecht, the Netherlands. HUSACCT has been developed at the Institute for ICT, and has started as part of the PhD research of Leo Pruijt [2], the architect and lead developer of HUSACCT. Students of the specialization "Advanced Software Engineering" have participated actively in the development. During the Spring semester of 2011, four teams of students have developed the first prototypes. During the spring semesters of 2012 and 2013, six teams worked concurrently on an integrated version of the tool. In December 2012 version 1.0 was released and in September 2013 version 2.0. Since then, the development of HUSACCT has been going on. The functionality is extended, and the accuracy, usability and performance have been improved in version 3 and 4. HUSACCT version 5 added advanced SAR functionality, with thanks to two student teams that participated in the development in spring 2016.

TABLE OF CONTENTS

Table of Contents	2
1 Getting Started.....	4
1.1 Download and run HUSACCT	4
1.2 Overview of HUSACCT's functionality.....	4
1.2.1 Overview of the Menu Options.....	5
1.2.2 Tour: Overview of the SACC work Process.....	6
2 Menu: File	11
2.1 New Workspace	11
2.2 Open Workspace.....	11
2.3 Save Workspace	12
3 MENU: Define intended architecture	13
3.1 Module Types and Rule Types	13
3.1.1 Common Module Types.....	14
3.1.2 Common Rule Types	15
3.2 Define Intended Architecture	16
3.2.1 Overview	16
3.2.2 Add Modules	17
3.2.3 Assign Software units.....	18
3.2.4 Add Rules	18
3.2.5 Add Exceptions to a Rule	19
3.2.6 Set Expression and/or Configuration Filter To a Rule	19
3.2.7 Move Layers	20
3.2.8 Conflicting Rules.....	21
3.2.9 View Intended Architecture in Browser.....	22
3.3 Intended Architecture Diagram	23
3.4 Import and Export Architecture.....	24
3.5 Report Architecture	24
4 Menu: Analyse Implemented Architecture.....	25
4.0 Dependency Types.....	25
4.1 Application Properties	27

4.2	Analyse Application.....	28
4.2.1	Accuracy of Code Analysis.....	28
4.2.2	Limitations.....	28
4.3	Analysed Application Overview	29
4.3.1	Decomposition View	29
4.3.2	Usage View.....	30
4.3.3	Code Viewer.....	31
4.4	Implemented Architecture Diagram	32
4.4.1	Menu Bar.....	32
4.4.2	Options Dialog.....	34
4.4.3	Zoom Options.....	36
4.4.4	Browse Dependencies & View Code	38
4.5	Reconstruct Architecture	39
4.5.1	Concept, Workflow and Approaches	39
4.5.2	SAR Example.....	43
4.5.3	MoJo.....	46
4.6	Analysis History	48
4.7	Report Dependencies (Dependency Report)	48
4.8	Export/Import Analysed Model	49
5	Menu: Validate Conformance.....	50
5.1	Validate Now.....	50
5.1.1	Violations per Rule	50
5.1.2	All Violations	51
5.1.3	Violations in Diagrams	53
5.2	Violations Export and Report	54
6	MENU: Tools	55
6.1	Options.....	55
6.1.1	General.....	55
6.1.2	Validate - Configuration	56
7	Literature	60

1 GETTING STARTED

1.1 DOWNLOAD AND RUN HUSACCT

- 1) Visit: <http://husacct.github.io/HUSACCT/>

At this site you can watch an introduction video, access the documentation and download the latest release of HUSACCT. Select “Download HUSACCT_x.x JAR File” and save the jar in a directory.

- 2) The tool starts when you open or double click the jar-file (if your local Java-settings are ok).

Note: HUSACCT requires Java 1.8.

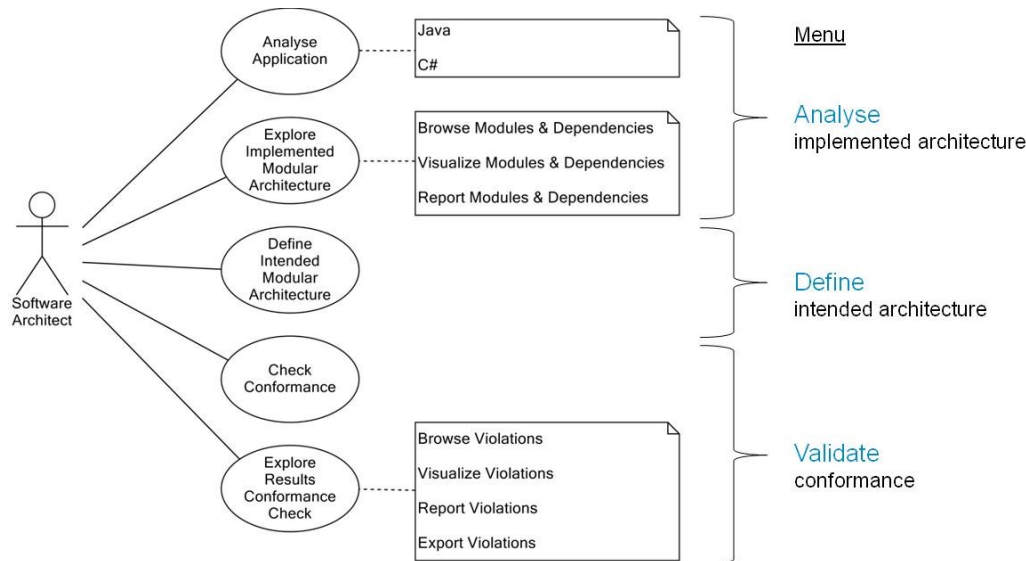
- a. If not (and you are running Windows), create a shortcut and edit Target to:
`java -jar <pathToHUSACCT>HUSACCT_x.x.jar`
 For example: `java -jar C:\Tools\HUSACCT\HUSACCT_5.1.jar`,
 or: `"C:\Program Files\Java\jre1.8.0_73\bin\java.exe" -jar C:\Tools\HUSACCT\HUSACCT_5.1.jar`
- b. Alternatively, start a Command prompt, cd to the directory with the HUSACCT jar and start the tool from the command line, for example: `C:\Tools>java -jar HUSACCT_5.1.jar`.
 If needed, check the Java version with command: `java -version`.

Note: In case of large systems, you can get an `OutOfMemoryError` with the message “Java heap space”. With the `-Xmx` JVM argument, you can set the heap size. For instance, for systems over 1.000.000 lines of code, allow the JVM to use 4 GB of memory. E.g. with the following command in a (Windows) shortcut or bat file: `java -jar -Xmx4g HUSACCT_5.1.jar`.

1.2 OVERVIEW OF HUSACCT'S FUNCTIONALITY

HUSACCT (HU Software Architecture Compliance Checking Tool) is a tool that provides support to analyze implemented architectures, define intended architectures, and execute conformance checks. Browsers, diagrams and reports are available to study the decomposition style, uses style, generalization style and layered style [3] of intended architectures and implemented architectures[4].

The diagram below shows an overview of the provided functionality, accessible via the menu options.



1.2.1 OVERVIEW OF THE MENU OPTIONS

The remainder of this user manual describes and illustrates the functionality per menu option. The table below provides an overview of the options per menu.

Menu	Option
File	New Workspace
	Open Workspace
	Save Workspace
	Close Workspace
	Exit
Define intended architecture	Define intended architecture
	Intended architecture diagram
	Export architecture
	Import architecture
	Report architecture
Analyse implemented architecture	Application properties
	Analyse application
	Analysed application overview
	Implemented architecture diagram
	Analysis history
	Reconstruct architecture
	Export analysis model
	Import analysis model
	Report dependencies
Validate conformance	Validate now
	Export violations
	Report violations
Tools	Options
Help	About HUSACCT
	Documentation

1.2.2 TOUR: OVERVIEW OF THE SACC WORK PROCESS

Follow the steps below to introduce yourself to the main SACC functionality. The following chapters contain detailed instructions per menu option.

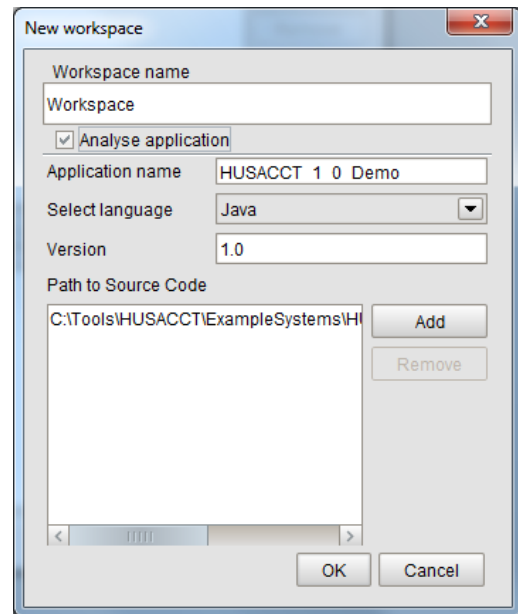
- 1) Select a software system to work on, acquire the source code, and (if available) the intended modular architecture and related architectural rules and guidelines.

In this case the code and architecture of HUSACCT version 1 are used to illustrate the steps.

- 2) Create a Workspace

Menu: File => New workspace.

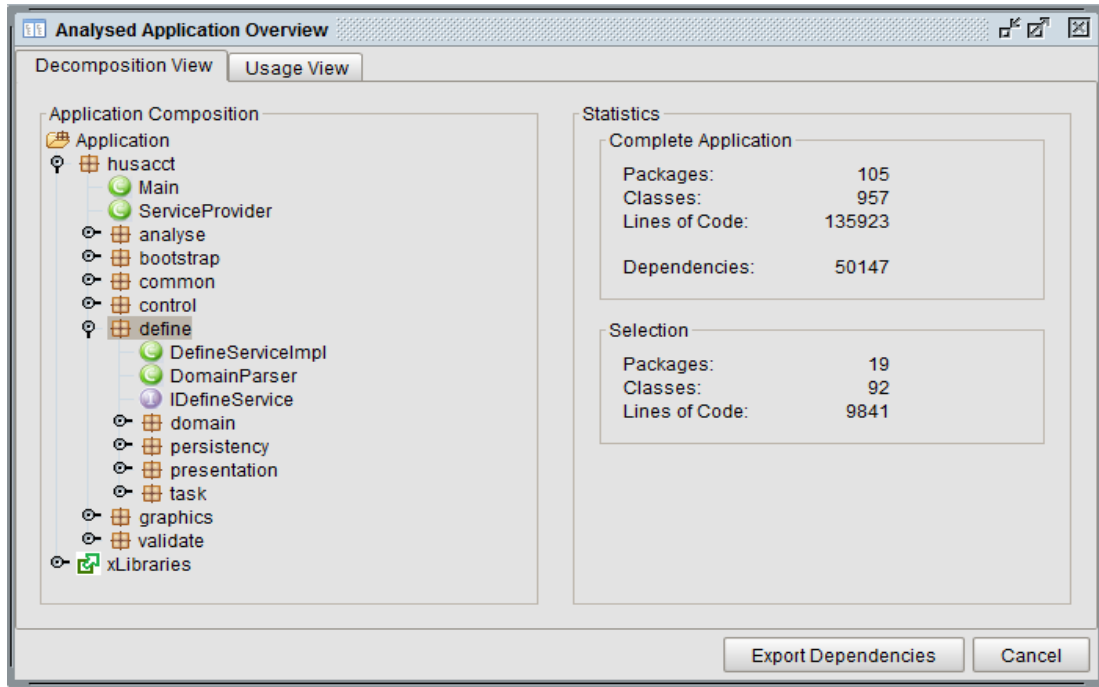
Mark "Analyse Application", enter the required data, and click on the OK-button.



- 3) Study the implemented architecture
 - a. Analysed application overview

Menu: Analyse implemented architecture => Analysed application overview

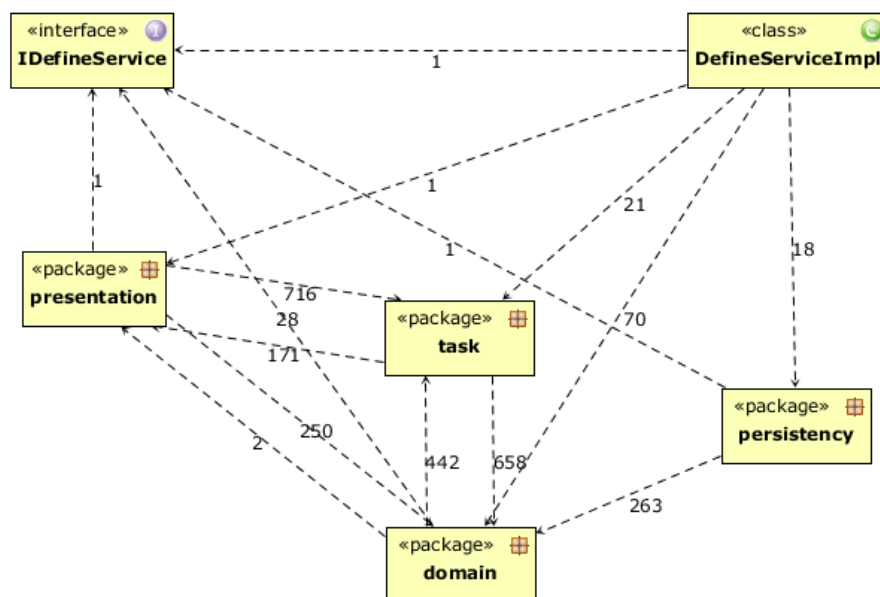
Study the Decomposition view, the related statistics, and the Usage view.



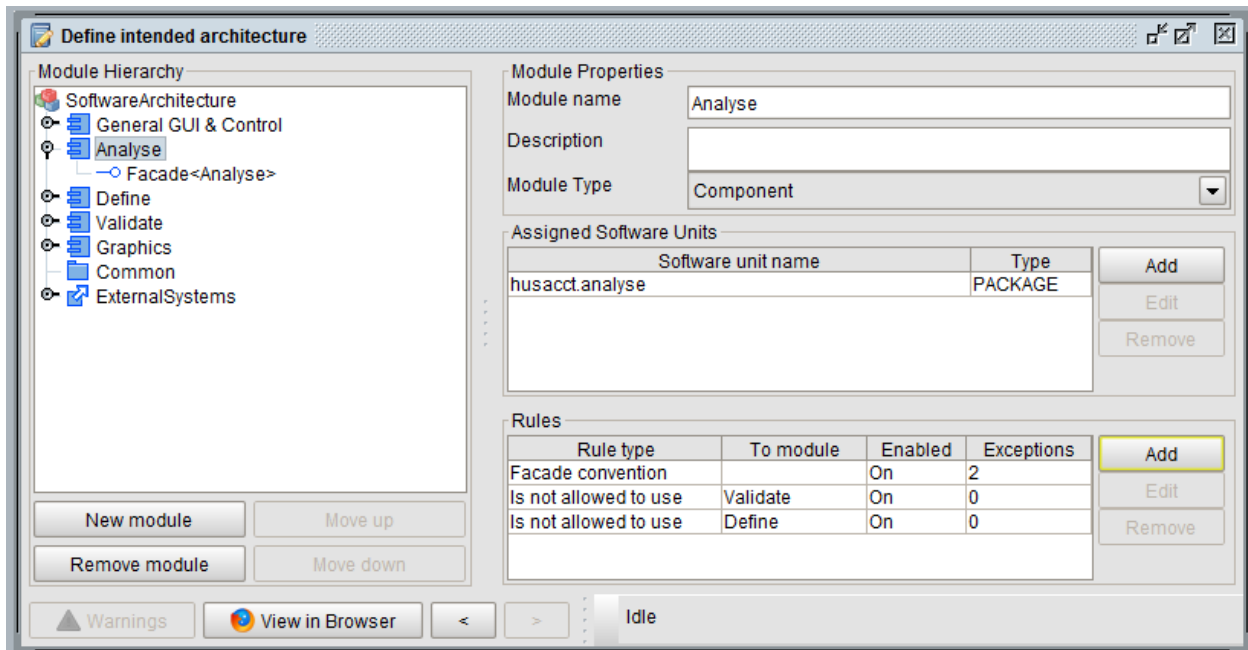
- b. Implemented architecture diagram

Menu: Analyse implemented architecture => Implemented architecture diagram.

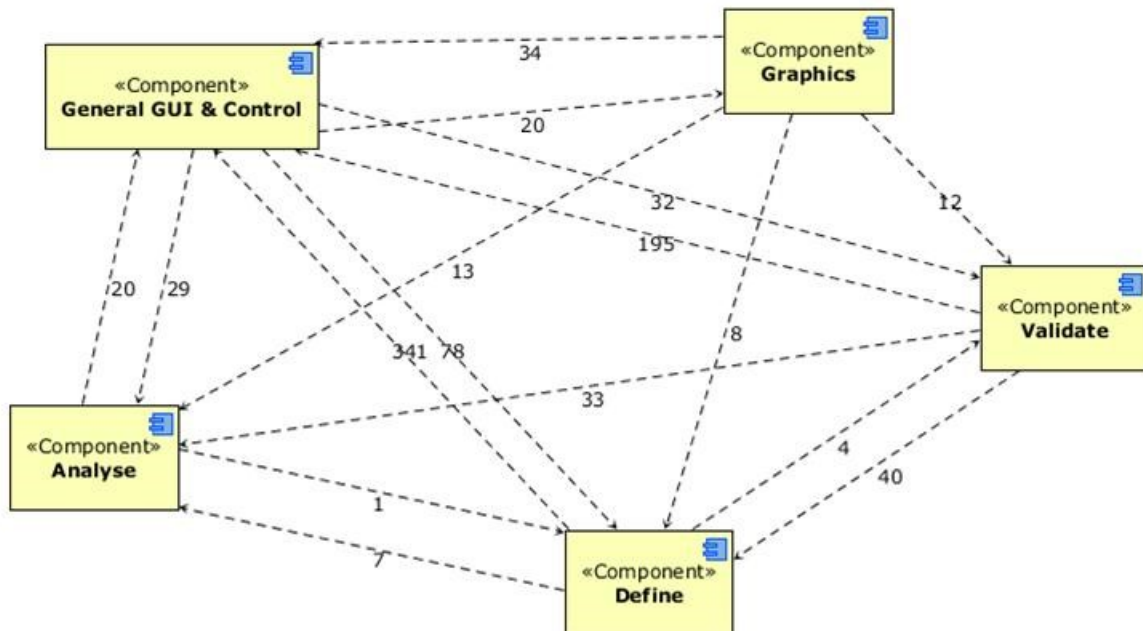
Make use of the zoom options.



- 4) Define the intended architecture
 - a. Create modules, add rules, and assign implemented software units to the modules.
Menu: Define intended architecture => Define intended architecture



- b. Intended architecture diagram
Menu: Define intended architecture => Intended architecture diagram.
Make use of the options to zoom in, hide modules, or save the diagram as a picture.



- 5) Check the conformance of the Implemented architecture tot the Intended architecture
 - a. Check the conformance

Menu: Validate conformance => Validate now.

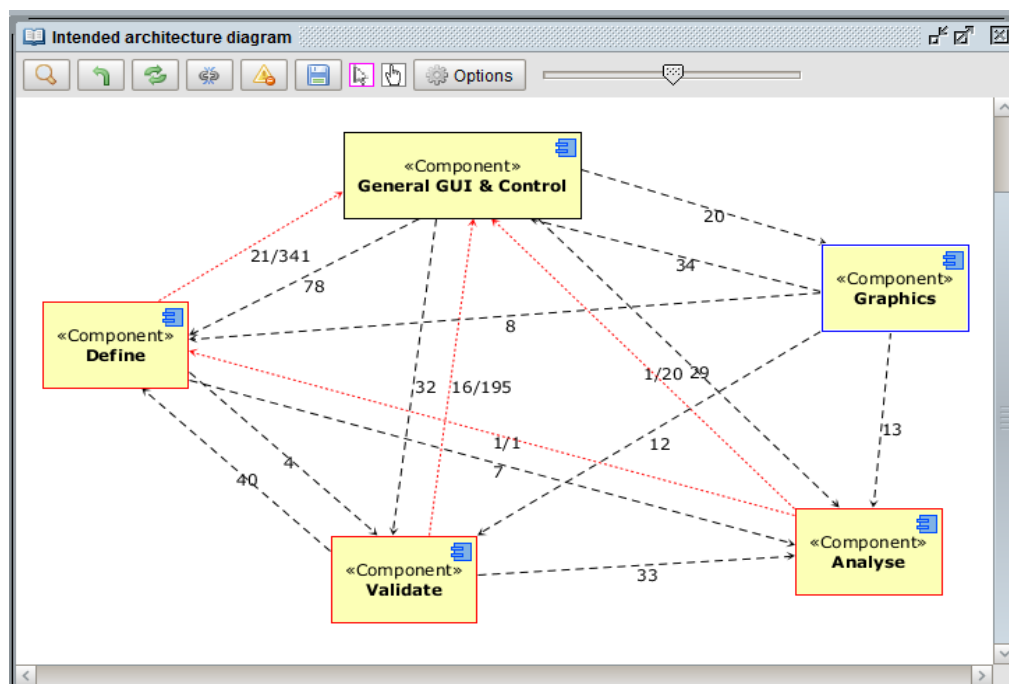
Validate conformance					
Violations Per Rule All Violations					
Rules with Number of Violations					
Id	Logical module from	Rule type	Logical module to	Violations	
1	Analyse	Is not allowed to use	Define	1	
2	General GUI & Control	Facade convention		38	

Violations						
From	To	Rule type	Dep.type	Direct	Line	
husacct.define.presentation.jdialog.AppliedRuleJDialog	husacct.control.ControlServiceImpl	Facade convention	Import	Direct	5	^
husacct.define.presentation.jdialog.AppliedRuleJDialog	husacct.control.ControlServiceImpl	Facade convention	Declaration	Direct	59	
husacct.define.presentation.jdialog.ViolationTypesJDialog	husacct.control.ControlServiceImpl	Facade convention	Import	Direct	6	
husacct.define.presentation.jdialog.ViolationTypesJDialog	husacct.control.ControlServiceImpl	Facade convention	Declaration	Direct	32	
husacct.define.presentation.jdialog.AddModuleValuesJDi...	husacct.control.ControlServiceImpl	Facade convention	Declaration	Direct	44	
husacct.define.presentation.jdialog.AddModuleValuesJDi...	husacct.control.ControlServiceImpl	Facade convention	Import	Direct	5	
husacct.define.presentation.jdialog.SoftwareUnitJDialog	husacct.control.ControlServiceImpl	Facade convention	Import	Direct	5	
husacct.define.presentation.jdialog.SoftwareUnitJDialog	husacct.control.ControlServiceImpl	Facade convention	Declaration	Direct	42	v

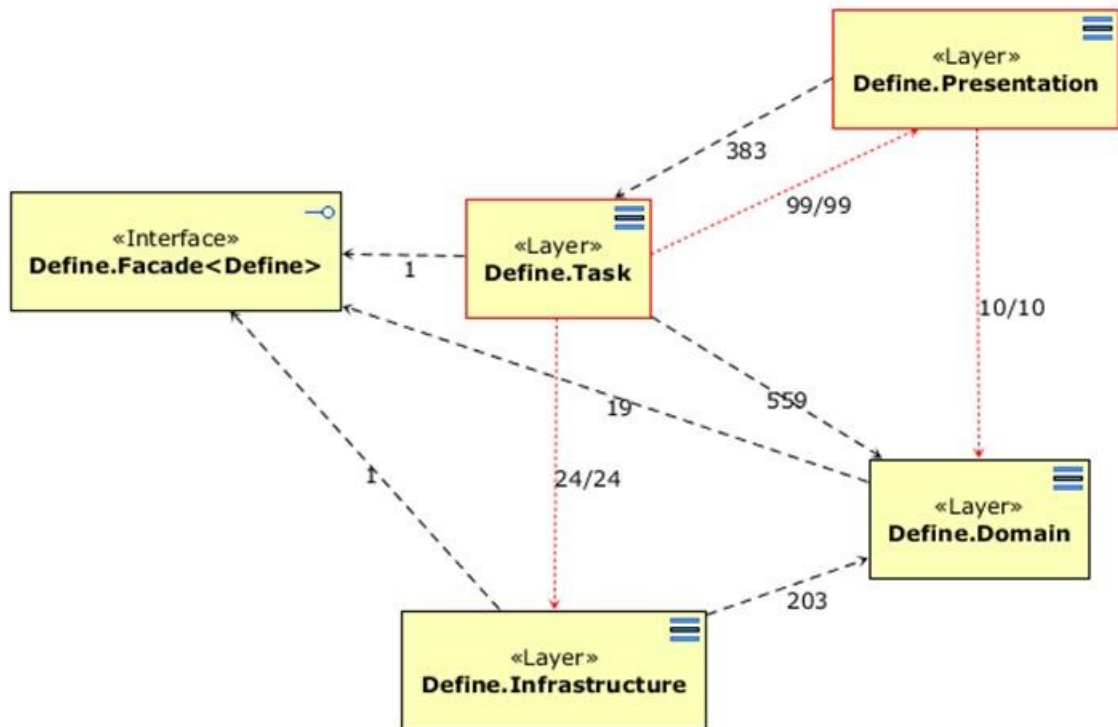
- b. Intended architecture diagram with violations

Menu: Define intended architecture => Intended architecture diagram.

Activate the options 'Show violations'.



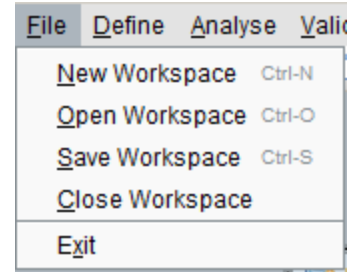
- a. Implemented architecture diagram with violations
Menu: Analyse implemented architecture => Implemented architecture diagram.
Activate the options 'Show violations'.



2 MENU: FILE

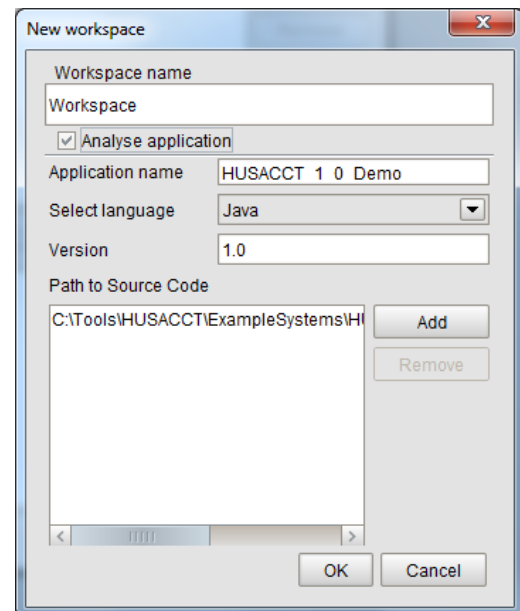
This menu allows you to manage your current workspace.

A workspace within HUSACCT contains all the information needed to analyse a target software system, study its implemented architecture, define its intended architecture and perform a compliance check. The workspace data may be stored in a file, which allows you to continue later on. Without a workspace, you cannot start working.



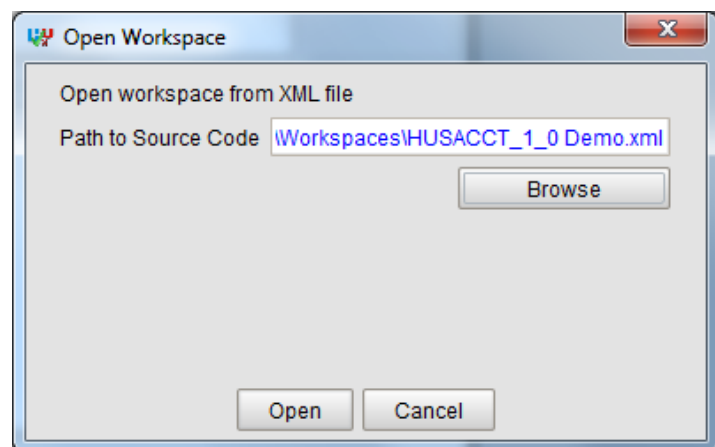
2.1 NEW WORKSPACE

- 6) To create a workspace, select File => New workspace.
- 7) Enter a name.
- 8) Select OK, if you want to start defining a new intended architecture.
- 9) Select "Analyse Application", if you first want to analyse the source code. If so, continue with the following steps.
- 10) Select the programming language.
- 11) Enter the version number of your application (not the Java version number).
- 12) Click on "Add" and select the directory where the *source code* is located. If needed, you can add several paths, or remove (old) paths.
- 13) Click "OK" and HUSACCT will start analysing the implemented application.
Thereafter, the implemented architecture may be studied (menu 'Analyse implemented architecture').
Furthermore



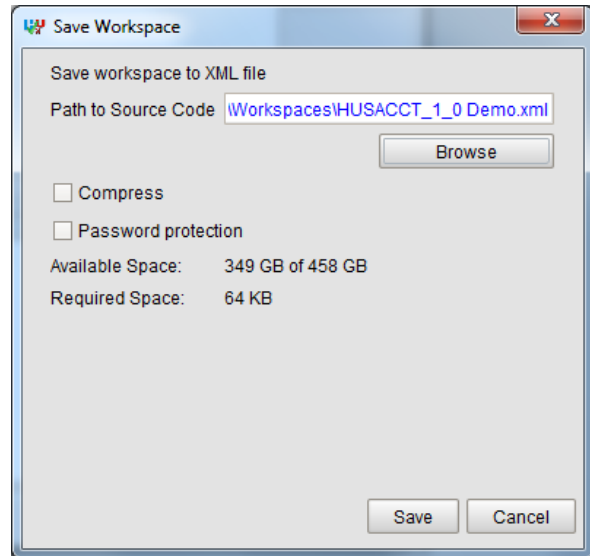
2.2 OPEN WORKSPACE

- 14) To open a workspace, select File => Open workspace
- 15) Select the file you want to open, using the "browse" function
- 16) Click open
- 17) When the file is password protected, you will be prompted for a password



2.3 SAVE WORKSPACE

- 18) To save a workspace, select File => Save workspace.
- 19) Select a directory, by using the "Browse" function, where the workspace file needs to be stored.
- 20) A couple of options is available:
 - Compress: Compresses the file to lower the required disk space
 - Password protection: Protect the file with a password, this makes the file unreadable when opening with a text editor.
- 21) Click "Save" and your workspace will be saved



3 MENU: DEFINE INTENDED ARCHITECTURE

3.1 MODULE TYPES AND RULE TYPES

HUSACCT stands out in its support of semantically rich modular architectures, which are very common in practice. A semantically rich modular architecture (SRMA) includes modules of semantically different types, while a variety of types of rules may constrain the modules [5]. As an example of an SRMA, Figure 1 shows a small part of an architecture model of one of the systems at an airport. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, et cetera. Examples in the rest of this document refer to elements in Figure 1.

Figure 1 shows UML icons for three semantically different types of modules: packages, components and interfaces. Layers are the fourth module type in the model (indicated by lines, since layers are not supported by UML). Finally, Spring and Hibernate represent the fifth type of module in the model: external system.

UML dependency relations in this example indicate is-only-allowed-to use rules; for instance, module HiWebApp is only allowed to use the modules HiForms and HimInterface, no others. Some other rules are not visible in the diagram. For example, rules related to the layered style, like “Technology Layer is not allowed to use Interaction Layer. Other examples of not visible rules are naming rules and rules inherent to components with interfaces.

Common Module and Rule Types

To enable compliance checks of SRMAs, rich sets of module and rule types should be supported. In a previous study [1], we presented a classification of common module types and common rule types. In this study, we use these common types as functional requirements to SRMA support. The next subsections describe these common module and rule types concisely to enhance practical understanding before the metamodel is presented. For a more in-depth discussion of the common module and rule types, we refer to our previous study.

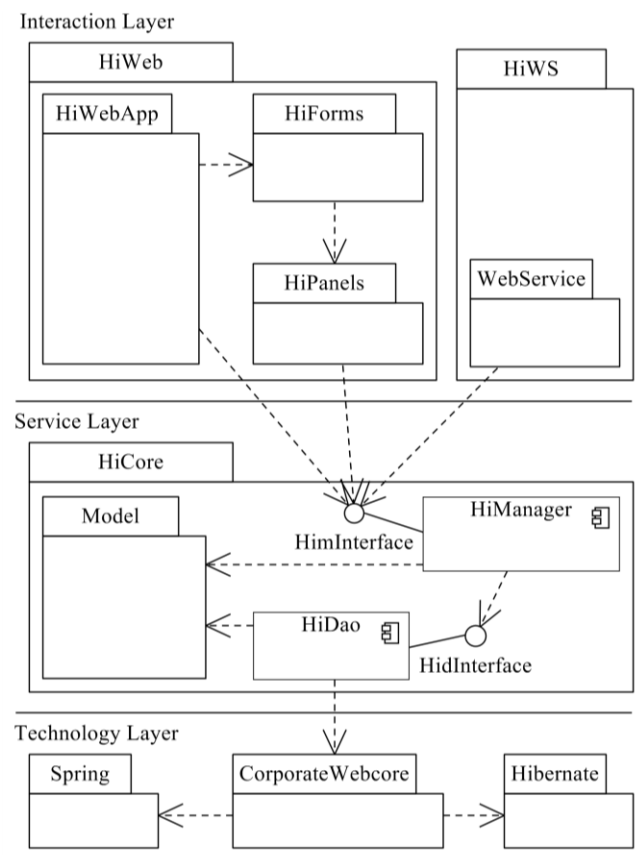


Figure 1. Example of an SRMA model

3.1.1 COMMON MODULE TYPES

SRMAs may contain modules of different types, with very different semantics. HUSACCT provides support for the following common types (of modules relevant for static SACC):

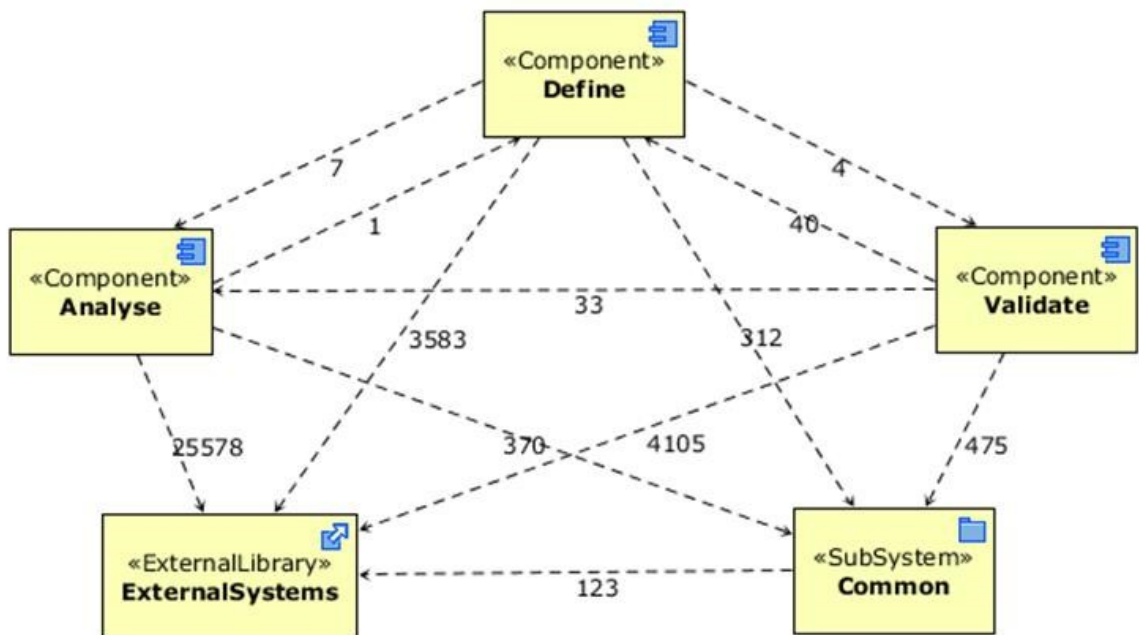
Subsystems represent units in the system design with clearly assigned responsibilities, but with no additional semantics. Comparable terms are logical cluster, or packages.

Layers represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. We cite Larman [6], who summarizes the essence of a layered design as “the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers.”

Components within software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas [7]). Consequently, a component differs from a logical cluster in the fact that it has a Interface sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior as in the “component and connector view” of architecture [3].

Interfaces are related to a component and act as facades, as described by the facade pattern [8]. An Interface at design level differs from the Java interface. An Interface may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.

External Library represents platform and infrastructural libraries or components used by the target system. HUSACCT’s SACC support includes the identification of usage of external systems and checks on constraints regarding their usage.



3.1.2 COMMON RULE TYPES

Modular architectures may contain rules of different types, where each rule type characterizes another kind of constraint on a module. These constraints are categorized in literature [3], [9] as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static SACC, resulted in two categories related to properties and relationships: Property rule types and Relation rule types. The rule types supported by HUSACCT are described and exemplified in Table 1.

Property rule types constrain a certain characteristic of the elements included in the module and their sub modules. Clements et al. [3] distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Facade convention, Inheritance convention), which represent the property Implementation information.

Relation rule types specify whether a module A is allowed to use a module B. The basic types of rules are “is allowed to use” and “is not allowed to use”. However, we encountered useful specializations of both basic types, which we included in the classification. Table 1 shows the two included specializations of “Is not allowed to use” (both specific for layers), and the three specializations of “is allowed to use”.

Table 1. Common rule types

Category\Type of Rule	Description (D), Example (E)	Ref ¹
Property rule types		
Naming convention	D: The names of the elements of the module must adhere to the specified standard. E: HiDao elements must have suffix DAO in their name.	[16]
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.	[3]
Facade convention	D: No incoming usage of the module is allowed, except via the facade. E: HiManager may be accessed only via HimInterface.	[8]
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.	[16]
Relation rule types		
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HiPanels is not allowed to use HiWS.	[17]
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.	[18]
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a lower layer that is more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.	[18]
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms.	[3]
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-module(s). E: HiForms is only allowed to use HiPanels.	[16]
Is the only module allowed to use	D: No elements, outside the selected module(s) are allowed to use the specified to-module. E: HiDao is the only module allowed to use CorporateWebcore.	[16]
Must use	D: At least one element of the module must use the specified to-module. E: HiDao must use CorporateWebcore.	[17]

¹ Ref= primary literature reference

3.2 DEFINE INTENDED ARCHITECTURE

This section will elaborate on the common tasks that you will need to perform to define an intended architecture. Keep in mind that most of these functionalities can also be accessed by using the right-click mouse button.

New module
Remove module
Move up
Move down

3.2.1 OVERVIEW

The define view is split into 5 areas, see the figure below. Each area is described below.

#1 Module Hierarchy

This area provides a decomposition view on the intended architecture. The hierarchy of modules is shown, while modules may be added, selected, and edited. You can select a module by clicking on it.

#2 Module Properties

This area shows the properties of the selected module. You can change the module's name, description, and type.

#3 Assigned Software Units

This area shows the software units in the implemented architecture, which are assigned to the selected module. The buttons are only enabled if a module is selected.

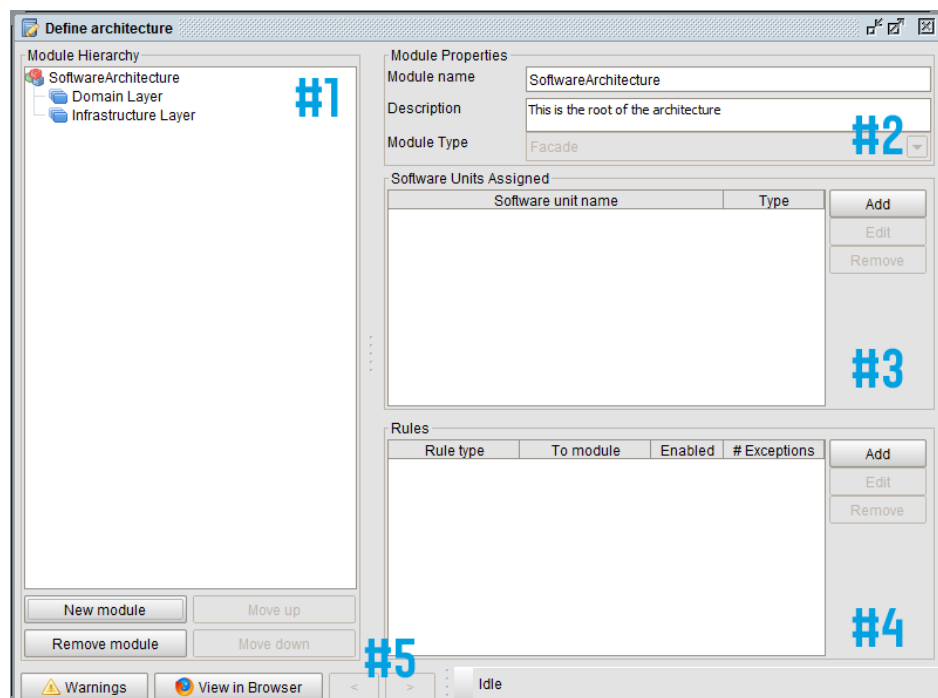
#4 Rules

This area shows the rules, which are defined for the selected module. The buttons are only enabled if a module is selected.

#5 Toolbar

The toolbar has some features that are not strictly necessary to define the architecture, but which provide additional support. (Currently, most of the functionality below is disabled.) The first button is the warnings button. When activated, a dialog appears with warnings about inconsistencies in your

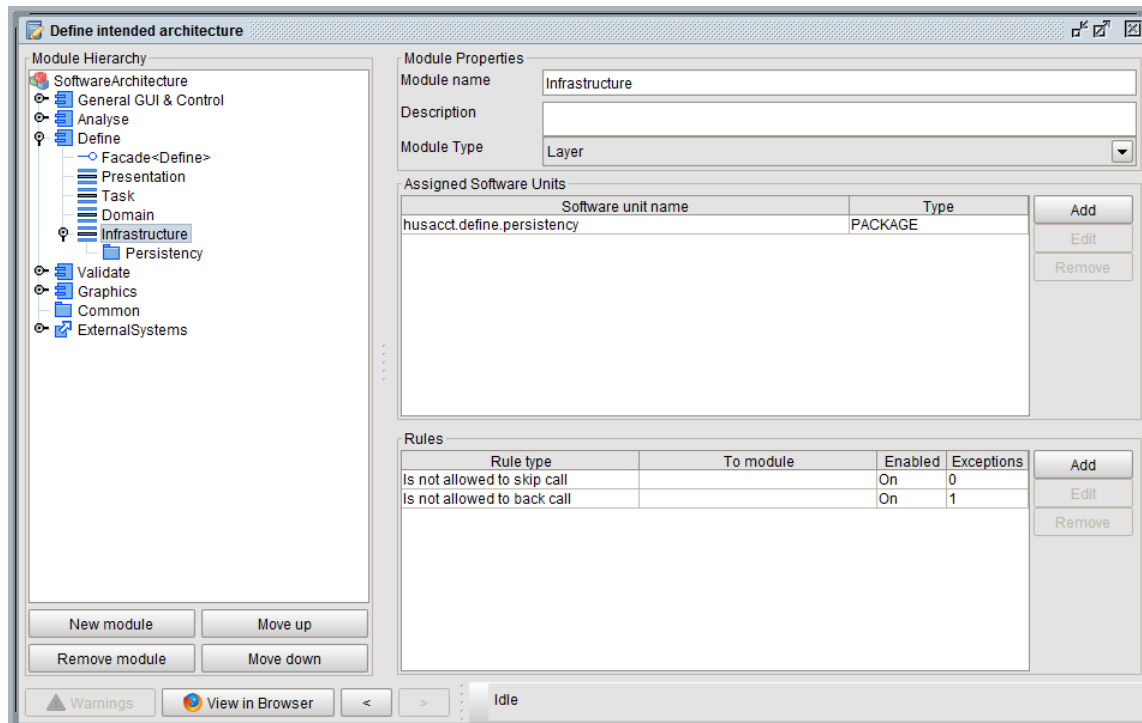
intended architecture. The second button, 'View in Browser', will activate a HTML-report/overview with all modules, applied rules and assigned software units. Next, there are undo and redo buttons. Made a mistake? Undo it!



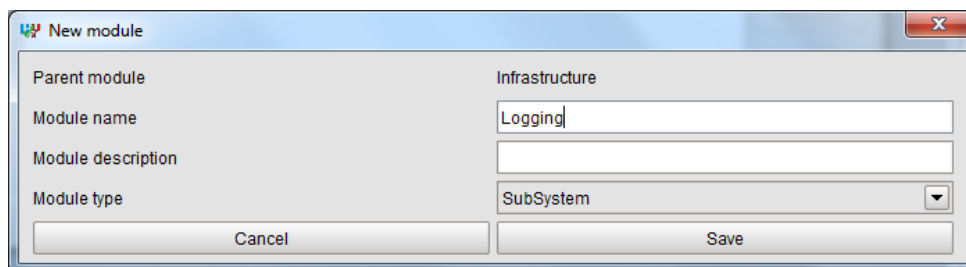
3.2.2 ADD MODULES

Accomplish the following steps to add a module to the logical architecture:

1. Select nothing or select the root node if you want to create a module to the root of the architecture. Otherwise select the module you wish to create a child module for.



2. Click on the “New Module” button.



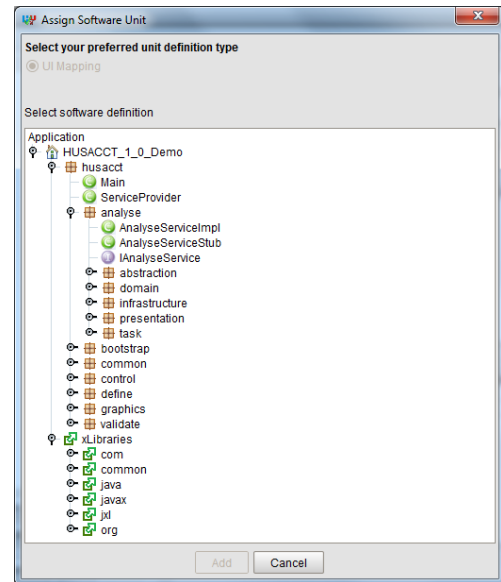
3. Enter a module name. Optional: Enter a description.
4. Select a module type.
5. Click on the “Save” button and the new module will be added to the module hierarchy.

3.2.3 ASSIGN SOFTWARE UNITS

Note: To be able to assign software units to a module, the application needs to be analysed in advance.

Accomplish the following steps to assign one or several software units to a module:

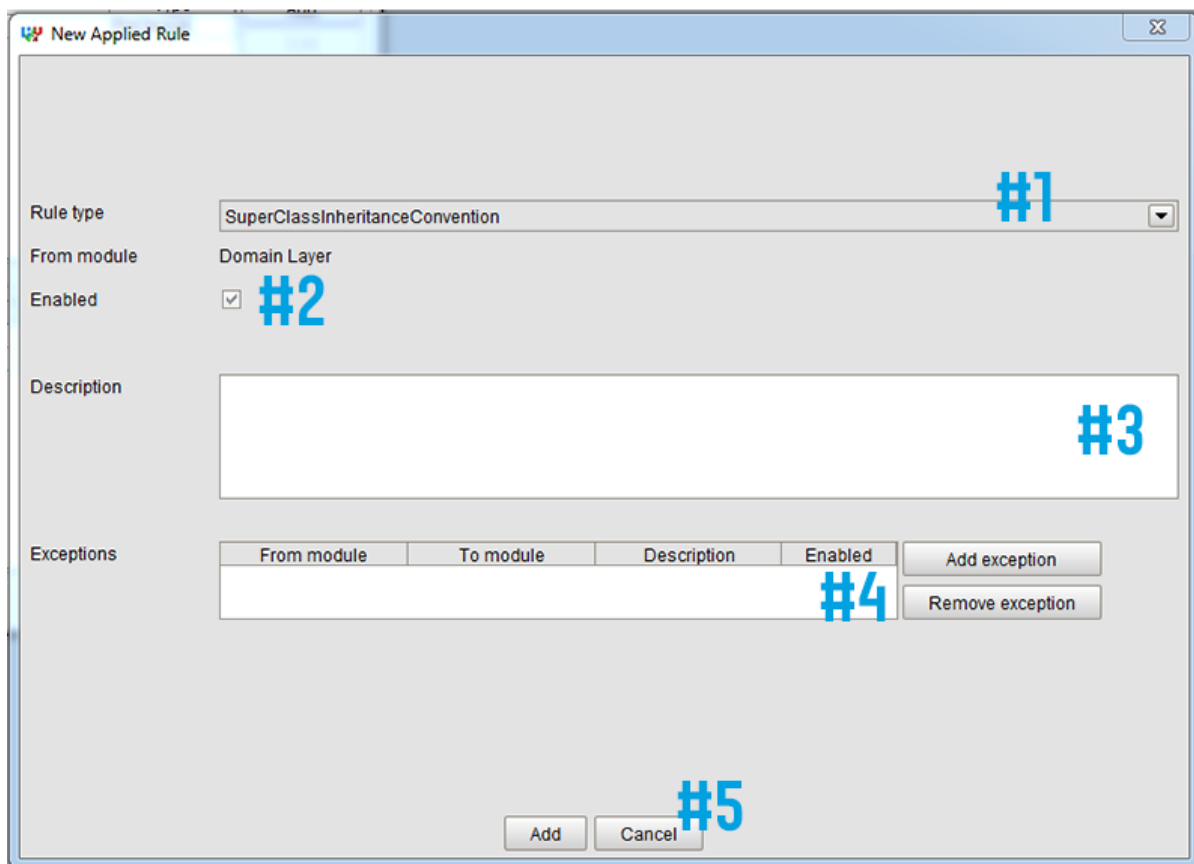
1. Select the module you wish to map.
2. Click on the “Add” button in the *Software Units Assigned* section.
3. Select the software unit you wish to map to the selected module. You can use the “Shift” key or “Ctrl” key to select multiple software units.
4. Click on the “Add” button to assign the selected software units to the selected module.



3.2.4 ADD RULES

Accomplish the following steps to add a rule to a module:

1. Select the module you wish to create a rule for.
2. Click on the “Add” button in the *Rules* section and the New Applied Rule form appears.



3. Select the rule type you wish to create (*#1 in the figure*).
4. In case of relation rules: Select the To-module of which the usage is constrained.
5. By default the rule is Enabled, but the rule may be disabled (*#2 in the figure*); temporarily, or continuously, e.g. in case of generated default rules.
6. Optional: Enter a description for this rule (*#3 in the figure*).
7. Optional: Add exception rules (*#4 in the figure*). Read subsection “Add Exceptions to a Rule”.
8. Enter an Expression, if required. Read subsection “Set Filter and/or Expression to a Rule”.
Rules of the following rule types require an Expression: Naming convention, Visibility convention.
9. Click on the “Add” button to add this rule with all its exception rules (*#5 in the figure*).

3.2.5 ADD EXCEPTIONS TO A RULE

Accomplish the following steps to add an exception to a rule:

1. Open the rule details panel. You can do this by adding a new rule, or by selecting an existing rule and pressing the “Edit” button in the *Rules* panel.
2. Press the “Add exception” button.
3. Select the rule type of the exception rule you wish to create. However, most rules have only one exception rule type available: is allowed to use.
4. Select the From-module and/or To-module.
5. Enable or disable this exception rule if needed.
6. Optional: Enter a description for this rule.
7. Optional: Configure the Violation Types.
8. Fill in any details required by the rule type. For example, the naming convention rule requires you to enter a regex.
9. Click on the “Add” button to add this exception rule to the main rules.

3.2.6 SET EXPRESSION AND/OR CONFIGURATION FILTER TO A RULE

3.2.6.1 *Naming convention*

In case of a rule of this type, all the class names and/or package names within the selected module should meet a specified Expression. Expressions have to be combinations of: 1) a text (case-sensitive) that has to be part of the name; and 2) *, which represent unconstrained text.

Examples are provided in the table below.

Use ‘Configure Filter’ to specify whether the rule should be applied to packages and/or to classes. By default, both are selected.

EXPRESSION	VALIDATES	RESULT
Friends	domain.locationbased.foursquare.History	False
	domain.locationbased.latitude.Friends	True
	infrastructure.socialmedia.locationbased.foursquare.FriendsDAO	True
	infrastructure.socialmedia.locationbased.foursquare.MyFriendsDAO	True
*Account	domain.locationbased.foursquare.MyAccount	True
	domain.locationbased.latitude.Map	False

	infrastructure.socialmedia.locationbased.foursquare.AccountDAO	False
DAO *	infrastructure.socialmedia.locationbased.foursquare.DAOFourSquare	True
	infrastructure.socialmedia.locationbased.foursquare.IMap	False
	domain.locationbased.foursquare.History	False

The first Expression in the table enforces that a class/packages must contain the word 'Friends'.

The second Expression enforces that a class/package must end with the word 'Account'.

The third Expression enforces that all the classes and/or packages must start with the word 'DAO'.

3.2.6.2 Visibility convention

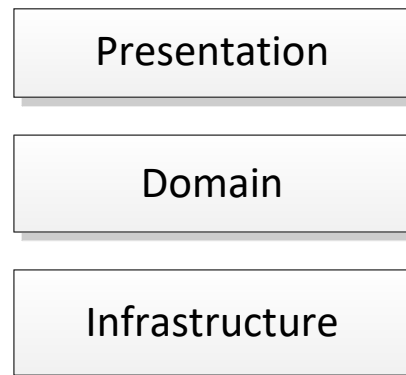
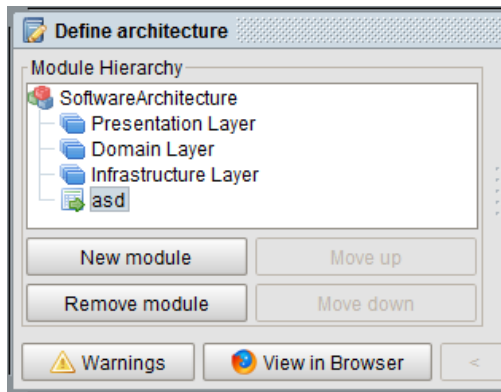
In case of a rule of this type, the visibility of all the classes and packages in the specified module must have the specified level of visibility, or lower. Use 'Configure Filter' to specify the level of visibility allowed as maximum. By default, all levels of visibility are selected. So, adding a visibility convention rule without setting these filter options may result in false negatives (non-reported violations).

Note: Setting a rule of this type requires code analysis first, since visibility settings are language dependent. When the code is not analysed previously, the Configure Filter option will be disabled.

3.2.7 MOVE LAYERS

For modules of type "Layer" the hierarchical position is an important attribute. To other types of modules, the hierarchical position is of no importance.

The module hierarchy in the figures below is equivalent.

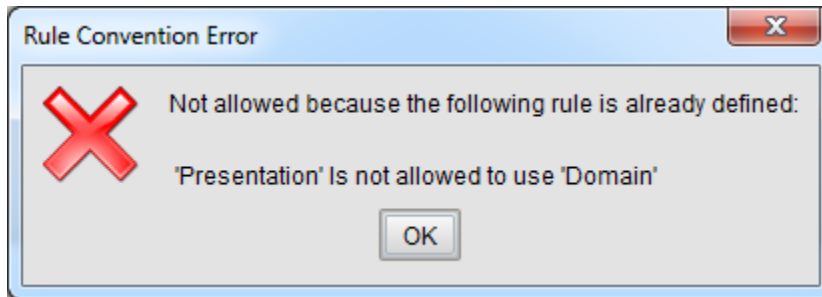


To move a layer up or down you will need to:

1. Select a layer.
2. Click on the "Move up" or "Move down" button in the *Module Hierarchy* section.

3.2.8 CONFLICTING RULES

It is not possible to define conflicting rules. For instance, if you were to create a rule that would state that the “Presentation” module is not allowed to use the “Domain” module. It would be impossible to define a rule that would state that the “Presentation” module must use the “Domain” module. If you would try to, this will result in the following error message.



Here is a list of all rules that cannot be defined if a rule of a certain type is already in place.

USE CASE	FORBIDDEN WHEN FOLLOWING RULE IS DEFINED
Naming convention	<ul style="list-style-type: none"> • “Naming convention” rule in the same module
Visibility convention	<ul style="list-style-type: none"> • “Visibility convention” rule in the same module
Subclass convention	<ul style="list-style-type: none"> • “Subclass convention”: rule in the same module • Same checks as a “must use” rule
Interface convention	<ul style="list-style-type: none"> • “Interface convention” rule in the same module • Same checks as a “must use” rule
Is not allowed to use	<ul style="list-style-type: none"> • “Is only allowed to use”, “is only module allowed to use”, “Is allowed to use” or “must use” rule from the selected module to the selected “module to”
Is only allowed to use	<ul style="list-style-type: none"> • “Is not allowed to use” rule from this module to the selected “module to” • “Is only allowed to use”, “is only module allowed to use”, “is allowed to use” or “must use” rule from this module to other then the selected “module to” • “Is only module allowed to use” rule from other then the selected module to the selected “module to”
Is only module allowed to use	<ul style="list-style-type: none"> • “Is not allowed to use” rule from this module to the selected “module to” • “Is only module allowed to use”, “is only module allowed to use”, “is allowed to use” or “must use” rule from other then the selected module to the selected “module to”
Is allowed to use	<ul style="list-style-type: none"> • “Is not allowed to use” rule from this module to the selected “module to” • “Is only allowed to use” rule from this module to other then the selected “module to” • “Is only module allowed to use” rule from other then the selected module to the selected “module to”
Must use	<ul style="list-style-type: none"> • “Is not allowed to use” rule from this module to the selected “module to” • “Is only allowed to use” rule from this module to other then the selected “module to”

	<ul style="list-style-type: none"> • “Is only module allowed to use” rule from other then the selected module to the selected “module to”
Skip call	<ul style="list-style-type: none"> • Same checks as a “is not allowed to use” rule for the 2nd layer below the selected layer, and each layer below this 2nd layer. You can see this layer as the selected “module to” layer.
Back call	<ul style="list-style-type: none"> • Same checks as a “is not allowed to use” rule for each layer above the selected layer. You can see this layer as the selected “module to” layer.

3.2.9 VIEW INTENDED ARCHITECTURE IN BROWSER

When you click on the ‘View in Browser’-button on the main screen of the define component, a report will be generated. This report consists out of an overview of modules, applied rules and software units and of a table with all the modules with their software units and applied rules. For example, see the figures below.

An overview of your architecture

There is a total of:

- 6 Modules;
- 8 Applied Rules (8 active);
- 0 Software Units.

[+] Expand All

Module	Software Unit	Applied Rule
▶ Presentation Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall
Domain Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall
▶ Infrastructure Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall

An overview of your architecture

There is a total of:

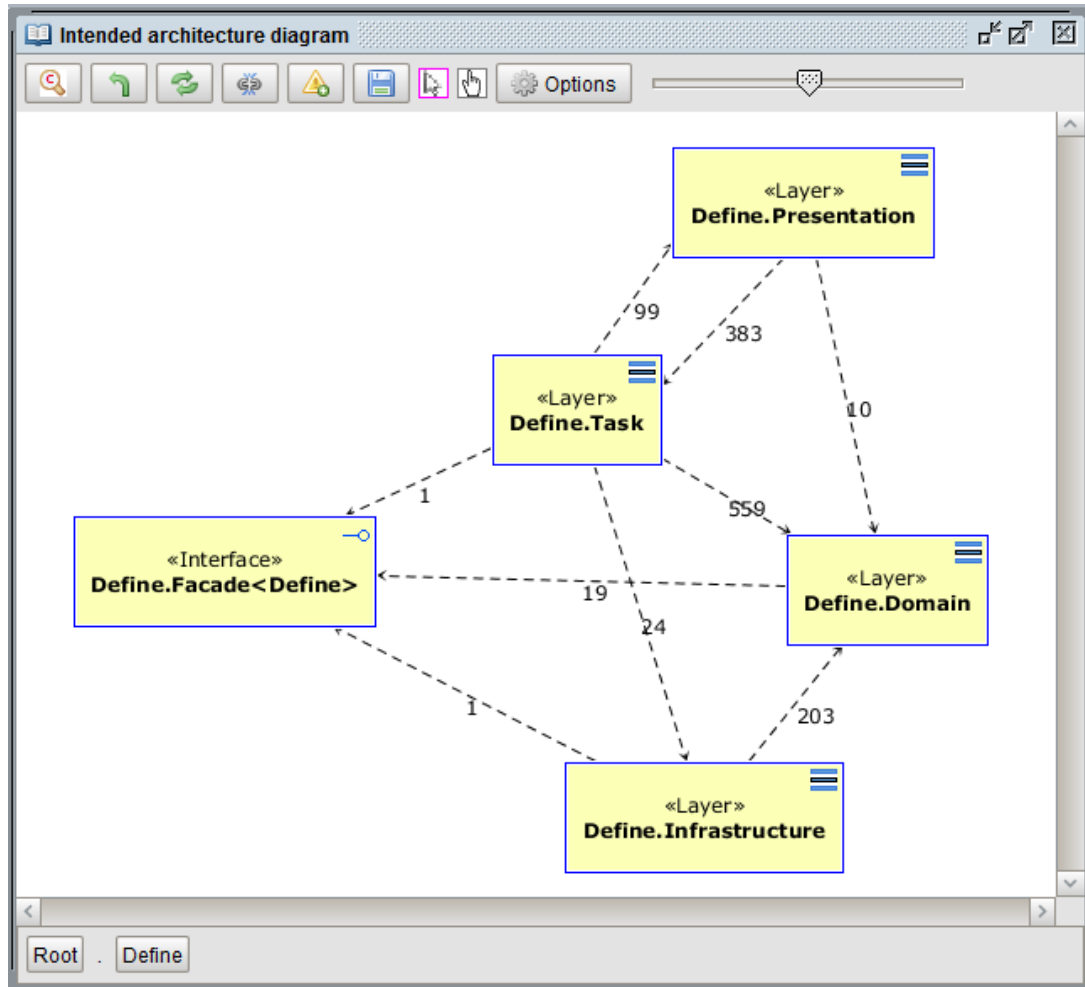
- 6 Modules;
- 8 Applied Rules (6 active);
- 0 Software Units.

[-] Collapse All

Module	Software Unit	Applied Rule
▼ Presentation Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall
qwe		VisibilityConvention
Domain Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall
▼ Infrastructure Layer		IsNotAllowedToMakeSkipCall IsNotAllowedToMakeBackCall
▼ asdas		FacadeConvention
Facade		

3.3 INTENDED ARCHITECTURE DIAGRAM

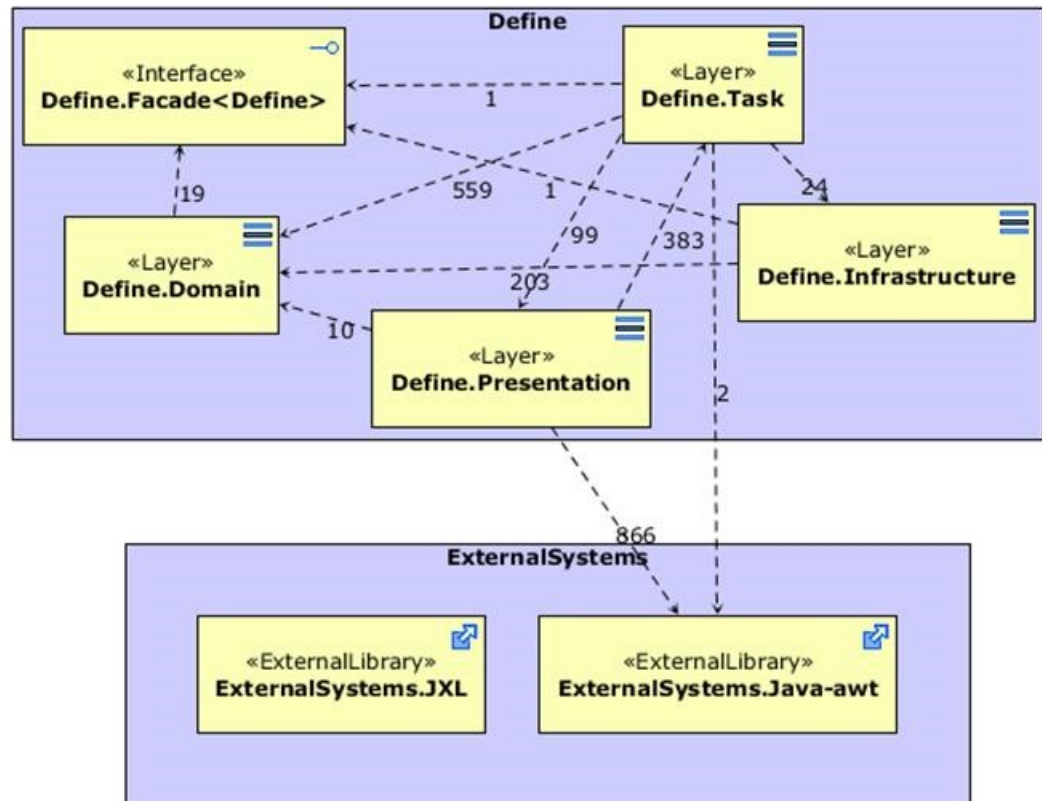
An intended architecture diagram shows the modules of the intended architecture at a certain hierarchical level. For each module the name and module type are shown. Furthermore, dependency arrows are shown between the modules, where the number indicates the number of detected dependencies in the source code of the software units assigned to the modules.



The following functionality may be used to adjust and export diagrams (for more instruction and details, visit the section on Implemented architecture diagram):

- Move modules within the diagram.
- Hide specific modules, and restore these modules, if needed.
- Zoom in on a selected module (default zoom, or zoom with context), or on several selected modules (multi zoom, see the example below). Furthermore: zoom out, refresh.
- Export a diagram to an image file (png file).
- Optional: show dependencies, show violations, show thickness of dependency arrow relative to the number of dependencies.
- Pan function.

Example of a multi zoom diagram:



3.4 IMPORT AND EXPORT ARCHITECTURE

The definition of the intended architecture may be exported as an xml file. It may be reused by an import of the file in different workspaces. The modules and rules are reusable, but the assignment of implemented software units to modules must likely be changed.

3.5 REPORT ARCHITECTURE

The definition of the intended architecture may be reported as a spreadsheet file. Specify the name and directory where the file will be stored.

The report will describe:

- The modules of the intended architecture and their types;
- The assignment of software units to each module;
- The rules, with their exceptions, constraining the modules.

4 MENU: ANALYSE IMPLEMENTED ARCHITECTURE

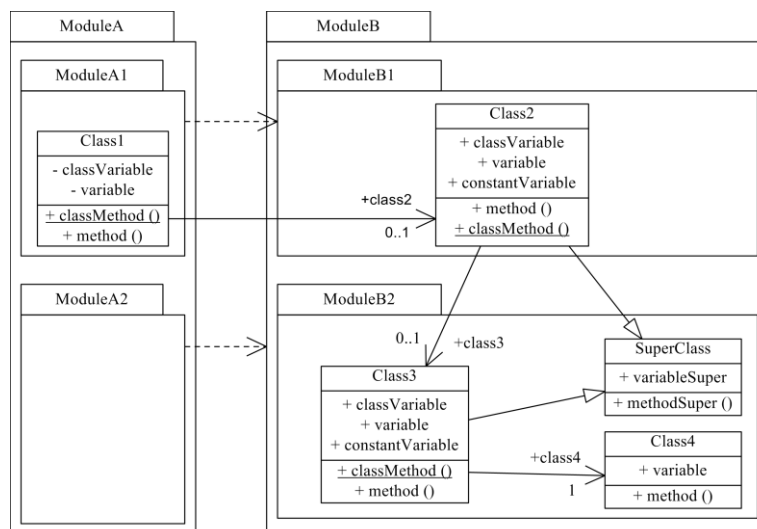
4.0 DEPENDENCY TYPES

Dependency analysis is “the process of determining a program’s dependences”[11]. Various types of dependencies are distinguished in literature. Callo Arias et al. [12] consider that all types fit into three main categories: structural dependencies, behavioral dependencies, and traceability dependencies. With HUSACCT, we focus on the category of structural dependencies, dependencies among parts of a system, is of interest here, since static analysis tools focus on dependencies that can be found by inspecting the source code.

Example of a Modular Implemented Architecture

The different types of dependency reported by HUSACCT are specified in the next subsections. These dependency types are illustrated on the basis of a modular architecture in UML notation, shown in the figure below. In this diagram, two modules, ModuleA and ModuleB, are shown, each with two submodules. The classes in the submodules are related via associations, showing for instance that an instance of Class1 may know several instances of Class 2. The dependency arrows show that ModuleA is allowed to use ModuleB1 and that Module A2 is allowed to use ModuleB. However, not all rules are visible. The following list shows the full set of relationship rules:

- ModuleA1 is allowed to use ModuleB1;
- ModuleA2 is allowed to use ModuleB, so also both sub modules, ModuleB1 and ModuleB2;
- ModuleA1 is not allowed to use ModuleB2;
- The submodules of ModuleA are allowed to use each other. The same type of rule applies to ModuleB.



Example of a modular architecture in UML notation.

Direct Structural Dependency Types

A dependency between two modules is *direct*, if there is an explicit reference to the to-class in the from-class (or in a super class of the from-class). For example, ModuleA in the figure depends on ModuleB, because a class in ModuleA1 uses a class in ModuleB1 with an explicit reference to that class. In Java, a preceding specification of an import command is required.

An overview of the direct structural dependency types distinguished by HUSACCT is shown in the first table below, together with several examples.

Indirect Structural Dependency

A dependency relation is indirect, when a code constructs in the from class results obviously in a dependency, but when the type of depended-upon class cannot be resolved without analyzing the code of another class (the to-class, or a superclass of the to-class, or a superclass of the from-class). For example, ModuleA1 in the figure depends on ModuleB2 via a class in ModuleB1. An overview of indirect structural dependency types distinguished by HUSACCT is shown in the second table below, together with several examples.

The dependency types Annotation, Declaration, Import, and Inheritance are in concept and in practice quite simple to differentiate. These four types have in common that they represent preparing activities, which do not take care of transformations.

The other types (Access, Reference, and Call) have in common that they represent executing activities, which take care of the transformations.

A dependency of type **Access** represents the actual usage (e.g., read or write) of a variable of another class (the server-class). In case of such an action, an Access dependency should link only to the server class that contains the variable, and does not consider the type of the accessed variable. Access of a variable of the client-class by the client-class itself is not interesting in case of architectural dependency analysis and is not reported as an Access dependency. However, if a class accesses a variable, there is a dependency to the type of that variable as well. This

Direct structural dependency types

Dependency Type	Example Code (from Class1 in the fig.)
Access Instance, Class variable; Object reference.	variable = class3.variable; variable = Class3.classVariable;
Annotation Class annotation	@Class3
Call Instance, Class method; Constructor.	variable = class3.method(); new Class3();
Declaration Class, Instance, Local variable; Parameter; Return type.	private Class3 class3; public void method(Class3 class3) {}
Import Class import	import ModuleB.ModuleB2.Class3;
Inheritance Extends class; Implements interface	public class Class1 extends Class3 { } public class Class1 implements Interface1 { }
Reference Type; Type cast; Type of used variable	method(class3); Object o = (Class3) new Object();

Indirect structural dependency types

Dep. Type	Example Code (from Class1 in the fig.)
Access Instance, Class variable; Object reference.	variable = class2.class3.variable; variable = class2.variableSuper;
Call Instance method; Class method.	variable = class2.class3.method(); variable = class2.methodSuper();
Inheritance Extends – extends; Access inherited variable; Call inherited method.	public class Class1 extends Class2 { } public class Class2 extends SuperClass { }

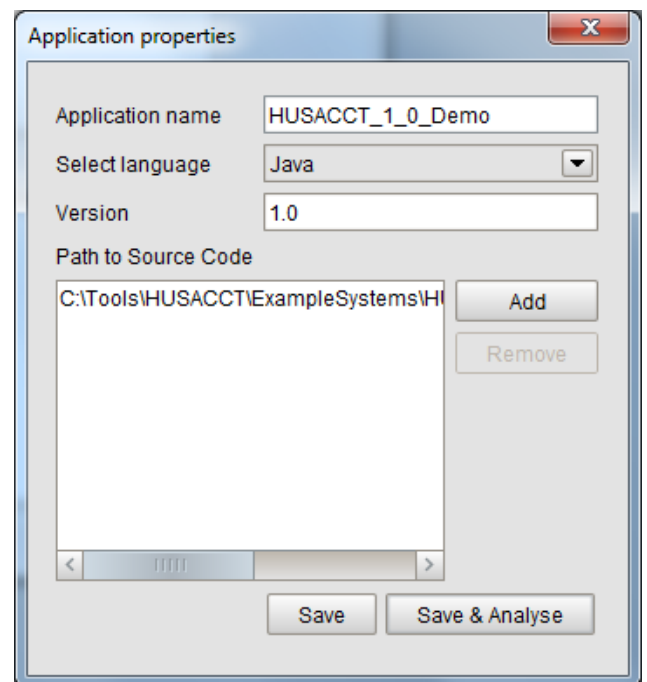
dependency on the variable type is reported, but as type Reference, not as type Access.

Dependency type **Call** represents the invocation of a method of the server-class. In case of such an action, a Call dependency should link only to the server class that contains the method, not to the return type of the invoked method. Calling a method of the client-class by the client-class itself, is not interesting in case of architectural dependency analysis and is not reported as a dependency. Again, the dependency on the return type of the method is useful and is reported as type Reference, not as type Call. In case of chained call and/or access statements, only a Reference dependency on the return type or type of the last element in the chain is useful, since dependencies on the preceding used types are reported as Call or Access dependencies.

A dependency of type **Reference** represents a link to the server-class or an object of type of the server-class in the context of an operational activity. At code level, references are often included in access statements or call statements, where they precede the actual variable or method to appoint the used class or object. In these situations, Reference dependencies are not useful to report, since they coincide with the Access and Call dependencies. Consequently, many tools do not report these dependencies (HUSACCT also does not). However, if a reference is not followed by an access or call statement, it is useful to be reported. For example, in case an object is passed as an argument.

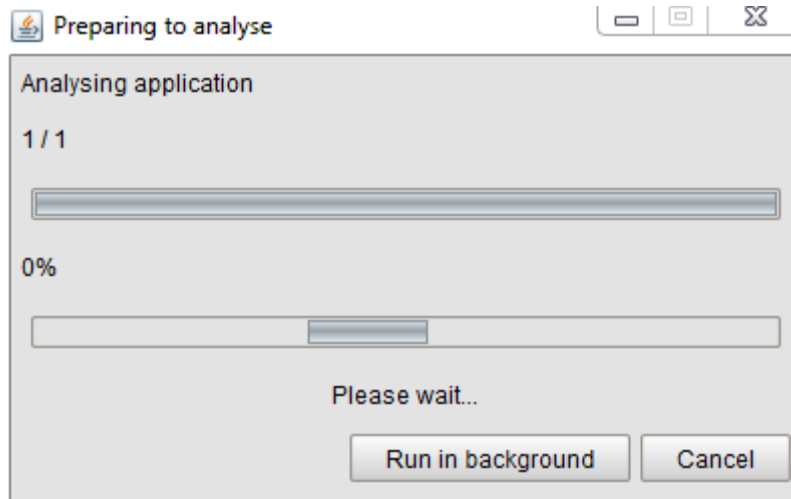
4.1 APPLICATION PROPERTIES

This option may be used to edit the application properties, like name, version, and path.



4.2 ANALYSE APPLICATION

When you run HUSACCT and create a new workspace you have the option to directly analyse a project. There is no difference between direct analysing and analysing on a later moment, when analysing is started the previous analysed information is cleared and the code will be completely reanalysed. Analysing a project can take up some time. Obviously, when the project contains many files it might take tens of seconds, while a small project is analysed within a second.



In the screenshot above you notice two loading bars which display the progress of the analysing. The top bar is implemented for future extensibility for analysing multiple projects. The lower bar is for the progress of the analysis of the current project. This bar starts running after the initial analysis process has finished and the repository is filled with raw data. Thereafter, hierarchical structures, external libraries and dependencies are derived from the raw data, and a dependency cache is build up.

4.2.1 ACCURACY OF CODE ANALYSIS

Since version 4.0, the accuracy of dependency detection and HUSACCT Code analysis is improved up to the level that all dependencies in the Java-based SACC Accuracy Test [1] are detected. Furthermore, dependency types and subtypes are reported correctly for these tests.

[1] Pruijt, L., Köppe, C., and Brinkkemper, S. (2013).

On the Accuracy of Architecture Compliance Checking: Accuracy of Dependency Analysis and Violation Reporting. 21st International Conference on Program Comprehension (pp. 172–181). San Francisco, CA, USA. IEEE Computer Society Press.

4.2.2 LIMITATIONS

- Two or more dependencies on the same type at the same line (or, in case of long expressions, several lines) are reported only once if the following attributes also have the same value: dependency type, subtype, isIndirect.

- The line number of a dependency may not be accurate in case of long expressions, which overlap several lines. In these cases, the first statement within the expression will be reported at the same line as the last statement in the expression. However, dependencies caused by arguments will be reported with their correct line number.
- C# dependency detection is less accurate than Java dependency detection.

4.3 ANALYSED APPLICATION OVERVIEW

The “Analysed Application Overview” helps you to study the decomposition of the software units of the implemented application, and the dependencies between these units. Two views are provided, which provide insight into the implemented architecture; useful for architecture reconstruction work.

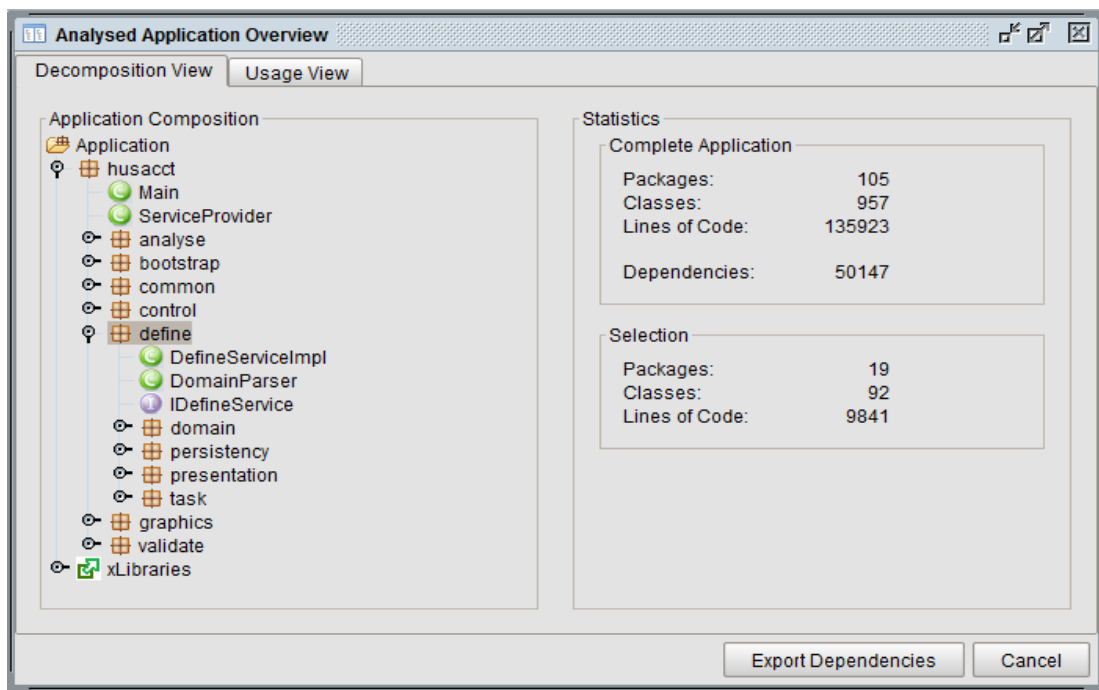
4.3.1 DECOMPOSITION VIEW

The “Decomposition View” is designed to let you study the hierarchical structure of the application and the used external systems. Software units may be selected and opened. Statistical information is shown for the application as a whole. Furthermore, when a unit is selected, statistical information about this unit is shown.

Packages: Number of packages, e.g. within a selected software unit or one of its child units.

Classes: Number of classes, e.g. within a selected software unit or one of its child units.

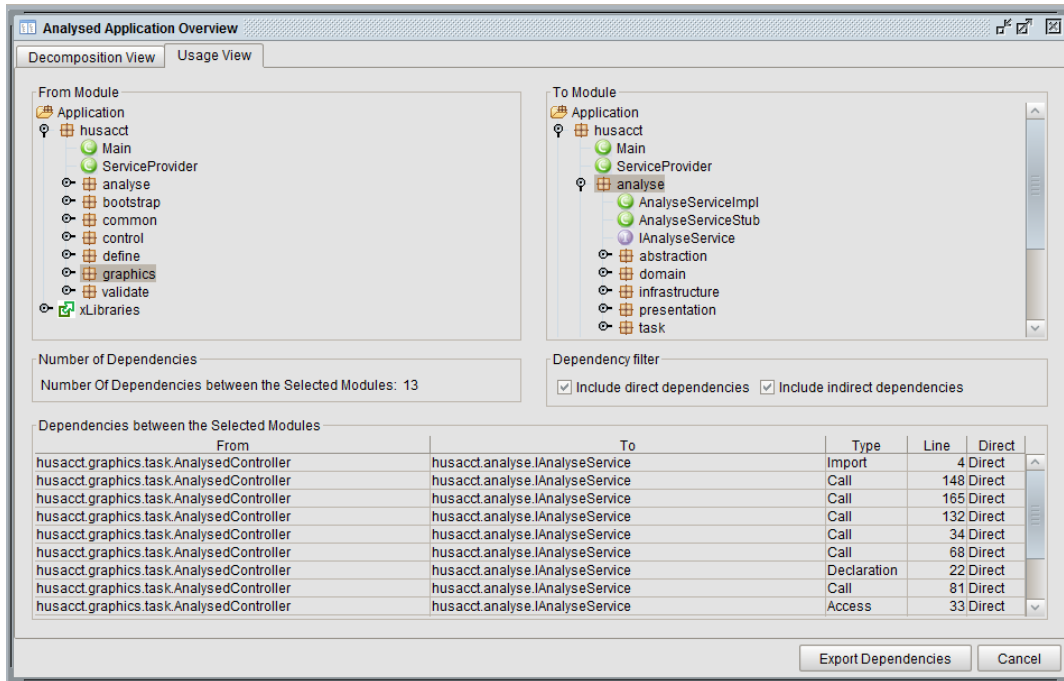
Lines of codes: Total number of lines of code (including comment and blank lines) within the classes, e.g. within a selected software unit or one of its child units.



4.3.2 USAGE VIEW

The “Usage View” is designed to study the dependencies between the software units. Select modules on the left (the From Module) and right (the To Module). If there are dependencies between these modules, the total number of dependencies is shown, while the bottom part of the form will show with detailed information about each dependency.

The Dependency Filter may be used to filter on direct or indirect dependencies.

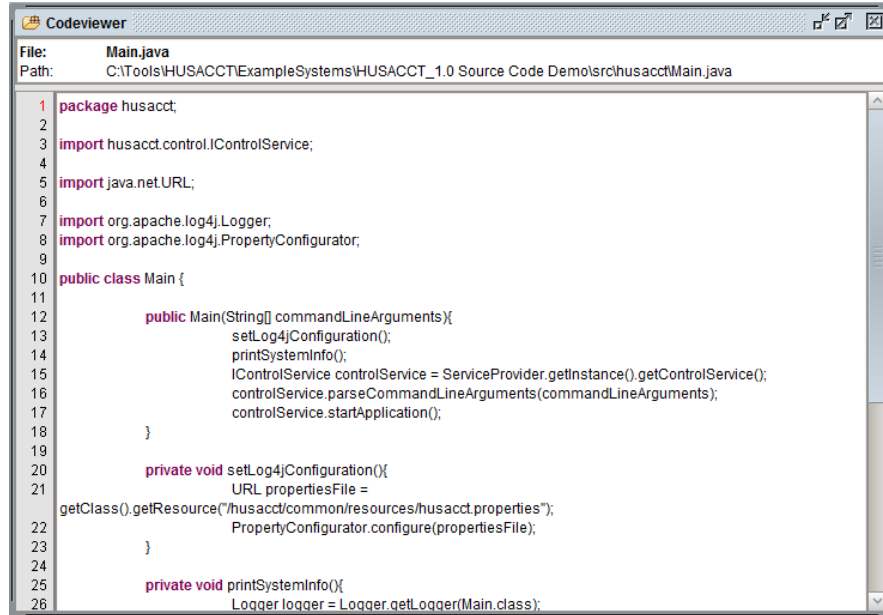


4.3.3 CODE VIEWER

The code viewer allows direct inspection of the source code.

The code viewer can be activated in both views of the Analysed Application Overview.

Decomposition View: Select a class or interface in the tree → right mouse click → Show source code.

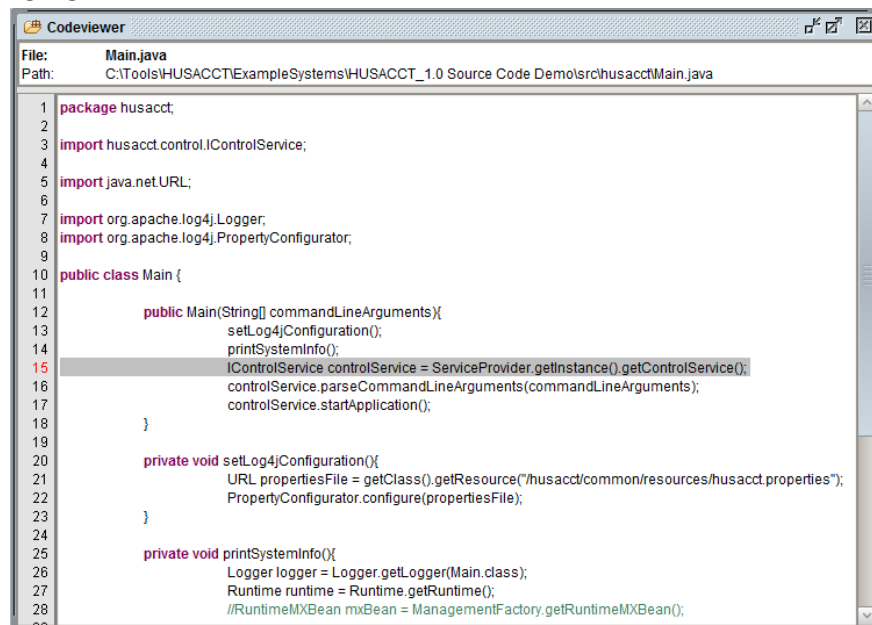


```

1 package husacct;
2
3 import husacct.control.IControlService;
4
5 import java.net.URL;
6
7 import org.apache.log4j.Logger;
8 import org.apache.log4j.PropertyConfigurator;
9
10 public class Main {
11
12     public Main(String[] commandLineArguments){
13         setLog4jConfiguration();
14         printSystemInfo();
15         IControlService controlService = ServiceProvider.getInstance().getControlService();
16         controlService.parseCommandLineArguments(commandLineArguments);
17         controlService.startApplication();
18     }
19
20     private void setLog4jConfiguration(){
21         URL propertiesFile =
22         getClass().getResource("/husacct/common/resources/husacct.properties");
23         PropertyConfigurator.configure(propertiesFile);
24     }
25
26     private void printSystemInfo(){
27         Logger logger = Logger.getLogger(Main.class);
28     }
29 }

```

Usage View: Double click on a dependency in the dependency table. The line which includes the dependency is highlighted.



```

1 package husacct;
2
3 import husacct.control.IControlService;
4
5 import java.net.URL;
6
7 import org.apache.log4j.Logger;
8 import org.apache.log4j.PropertyConfigurator;
9
10 public class Main {
11
12     public Main(String[] commandLineArguments){
13         setLog4jConfiguration();
14         printSystemInfo();
15         IControlService controlService = ServiceProvider.getInstance().getControlService();
16         controlService.parseCommandLineArguments(commandLineArguments);
17         controlService.startApplication();
18     }
19
20     private void setLog4jConfiguration(){
21         URL propertiesFile = getClass().getResource("/husacct/common/resources/husacct.properties");
22         PropertyConfigurator.configure(propertiesFile);
23     }
24
25     private void printSystemInfo(){
26         Logger logger = Logger.getLogger(Main.class);
27         Runtime runtime = Runtime.getRuntime();
28         //RuntimeMxBean mxBean = ManagementFactory.getRuntimeMxBean();
29     }
30 }

```

Furthermore, the code viewer may be activated in the same way as in the Usage View from dependency overviews in the diagram tools and dependency overviews in the Validate Conformance views.

4.4 IMPLEMENTED ARCHITECTURE DIAGRAM

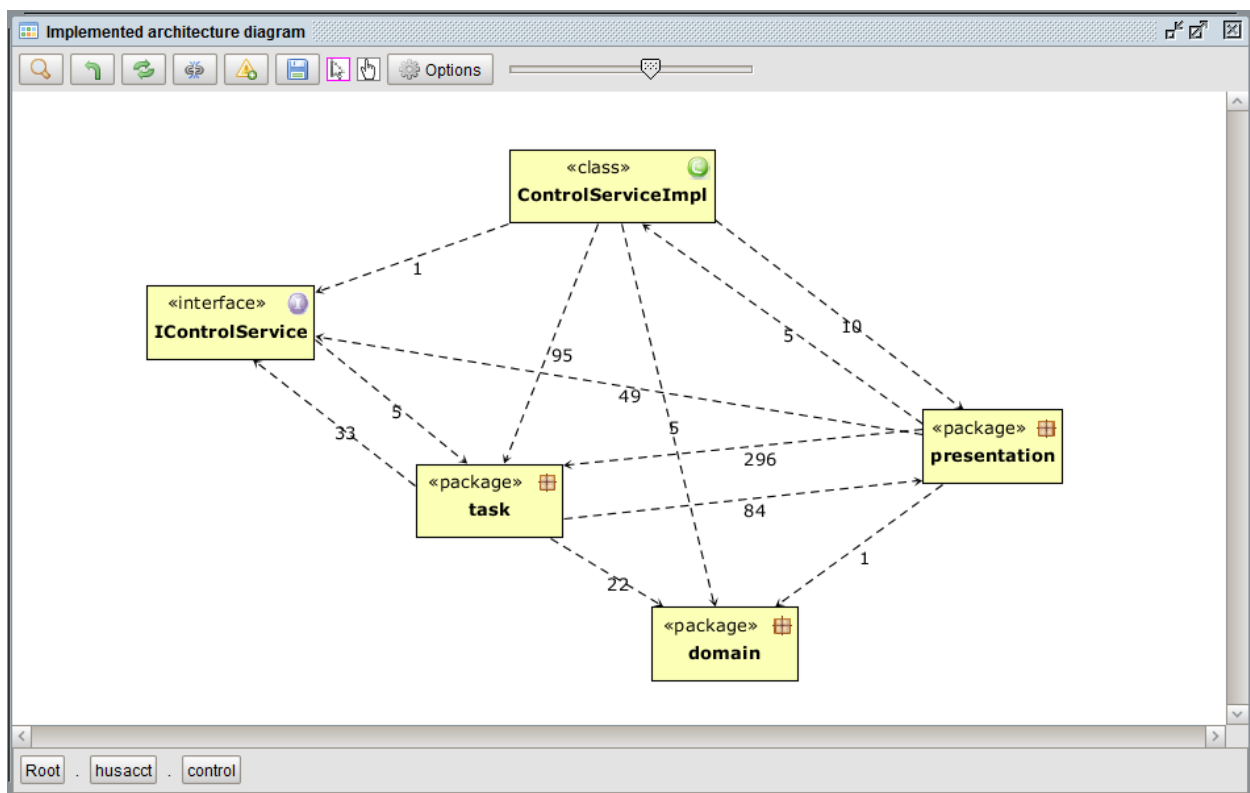
An implemented architecture diagram shows the implemented software units (packages, classes, interfaces) with their dependencies at a certain hierarchical level.

As visible in the picture below, of each software unit the name and type (in text or as icon) are shown, while dependency arrows with the number of detected dependencies are drawn from the unit to the units where it depends upon.

Modules may be selected, one or several concurrently, and moved to another position within the drawing canvas. That way, a comprehensible diagram may be created.

Note: The layout of the diagram is not stored and will be lost after zoom in, zoom out, refresh, or after a status change caused by actions under other menu options. So, if you want to save the layout, export the diagram as an image.

At the bottom of the editor, the path within the decomposition hierarchy is shown. As indicated, the diagram represents the contents of package husacct.control.



4.4.1 MENU BAR

At the top of the diagram editor, icons are shown which represent functionality to adjust and export a diagram. This functionality is explained below.



#2 Zoom Options

Left click to zoom, right click to select zoom options.

- Zoom In
- Zoom In with Context

#3 Return

Return to the previous level of abstraction in the decomposition hierarchy.

#4 Refresh

Refresh the diagram. Changes to the layout will get lost.

#5 Dependencies

Click to toggle between including and not including dependency arrows.

#6 Violations

Click to toggle between including and not including violating dependency arrows.

#7 Export diagram to file

Export the diagram to an image file.

#8 Mouse tools

Click the leftmost option for the select tool, the rightmost option for the pan tool.

The item bordered in cyan is the currently selected tool.

The pan option is useful if the diagram is larger than the shown part in the editor. In that case, select the pan option, click on the canvas, hold the left mouse button and move the mouse. As a result, the whole diagram will move with your mouse movement.

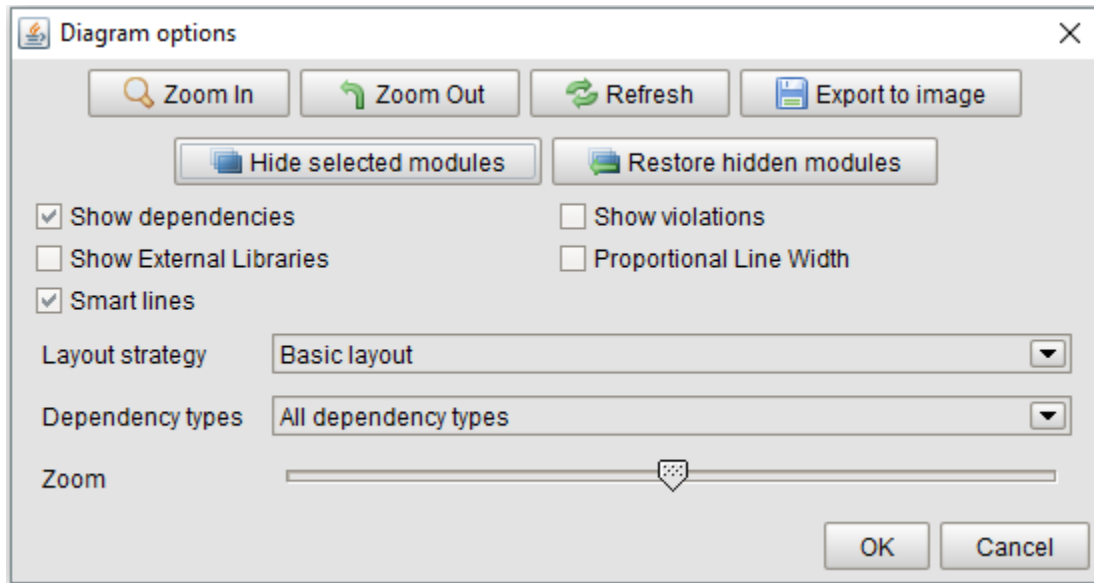
#9 Options

Click to open the options menu.

#10 Zoom slider

Slide to zoom in or zoom out.

4.4.2 OPTIONS DIALOG



The options dialog contains several functionalities also available in the menu bar.
The additional options are described below.

Hide selected modules

Removes the module or software unit temporarily from the diagram. It is not removed in the repository.

Restore hidden modules

The modules or software units, which were previously hidden, will be included again in the diagram.

Show External Libraries

This checkbox reflects whether or not external libraries will be displayed.
Select this option at root level.

Proportional Line Width

When selected, the width of the dependency arrows will be shown relative to the number of dependencies represented by the arrow.

Smart lines

This option takes care of a smart distribution of the lines over the diagram.

Layout strategy

Select a suitable layout strategy. Currently only a few strategies are provided.

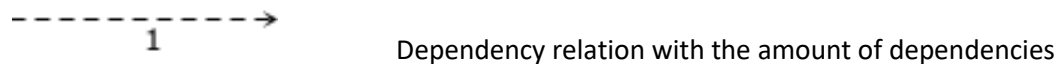
Dependency types (only available in Implemented architecture diagram)

Select the types of dependencies to be included in the diagram. Three options are available: All dependency types (default option); Only dependency types Access, Call, and Reference; Only dependency types shown in a class diagram. The latter option transforms the dependency diagram in a class diagram with unidirectional associations (with multiplicity), inheritance and implements relations between the classes in the diagram. In case of packages, the links between the classes in the packages are totaled and this number is shown in the middle of a directed line. Read the next paragraph for additional information.

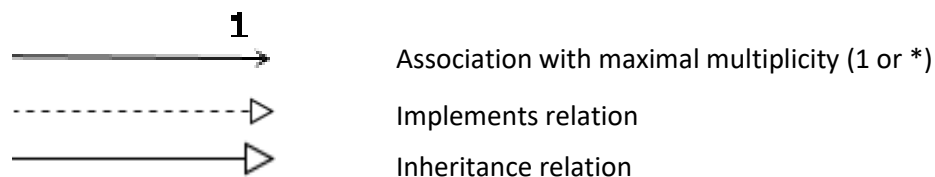
4.4.2.1 Different types of dependencies

In an implemented architecture diagram, several options are available with respect to the dependencies between the software units. These options can be selected in the options dialog, as described in the previous paragraph.

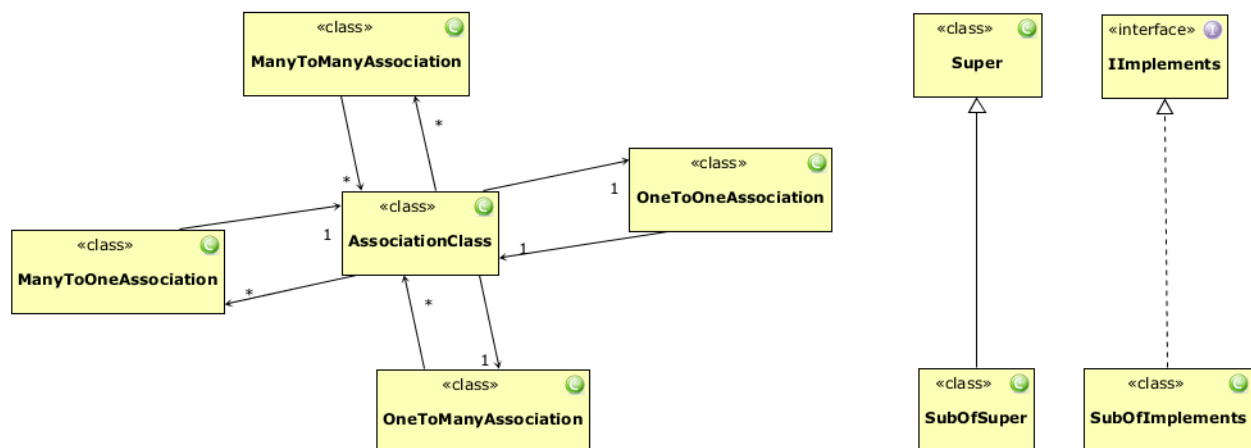
The default option will show all dependencies of all dependency types as shown in the image below. The second option shows the dependencies in the same way, but filters out dependencies not of the types Access, Call, and Reference. These three types represent executing activities, which take care of the transformations (in contrast to preparing activities, e.g. dependencies of types Import or Declaration).



The third option is to include only dependencies represented by special lines in UML class diagrams: associations (attribute as association), inheritance (generalization), and implements relations. These types of dependencies are represented by the following arrows:



An example of a class diagram that includes these dependency types is shown below.



4.4.3 ZOOM OPTIONS

Several options to zoom in are available. These options are described below.

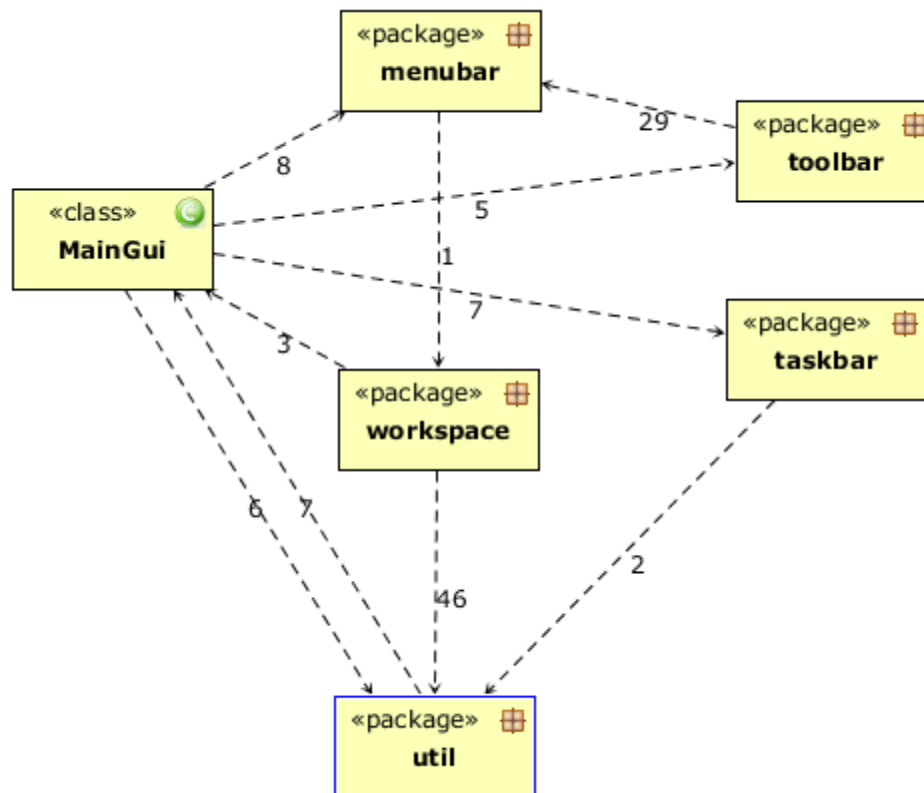
4.4.3.1 Default Zoom In

Default zoom in will show the contents of the selected module only in the decomposition diagram.

Procedure:

- 1) Check the zoom option icon in the menu bar. It should display a normal magnifying glass. If not, edit the zoom option setting with a right mouse click to 'Zoom In'.
- 2) Select a module in the diagram.
- 3) Zoom in: right mouse click => Zoom In, or click on menu bar icon, or click on icon in diagram options dialog.

Example of a default zoom in on husacct.control.presentation

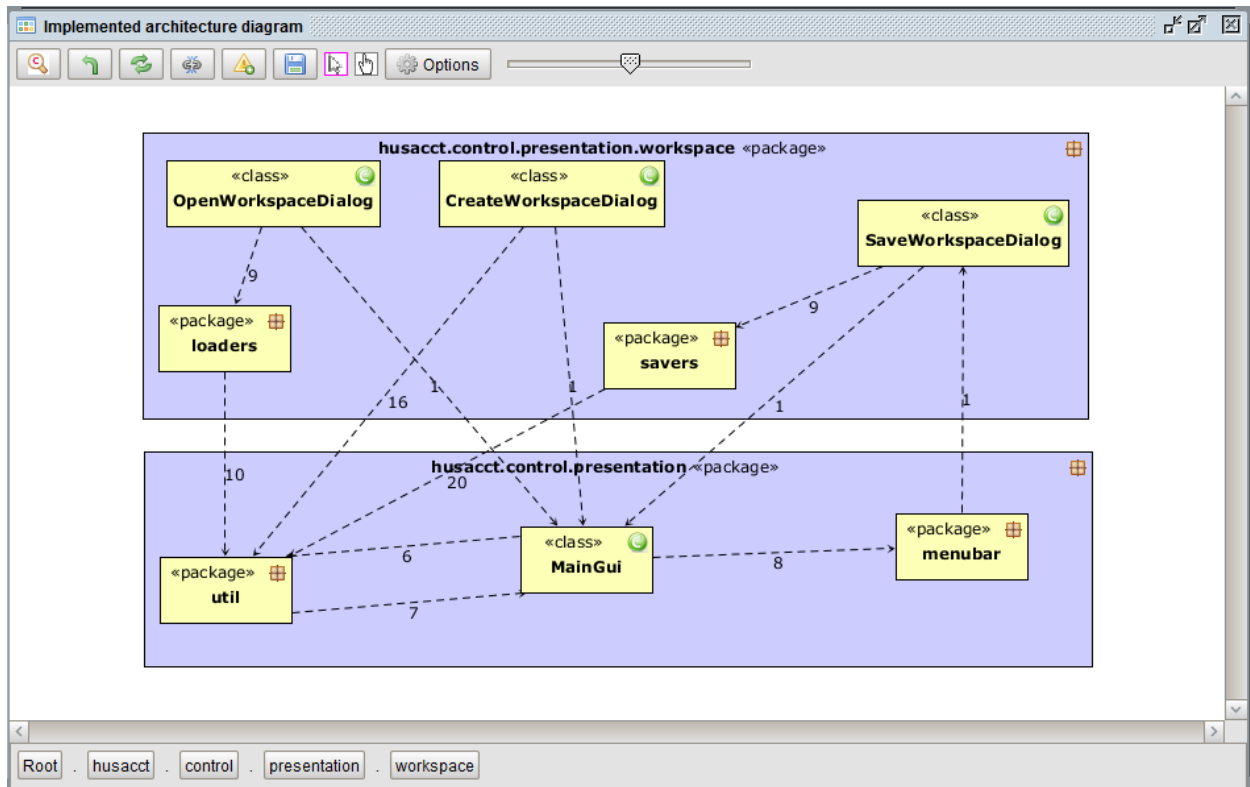


4.4.3.2 Zoom In with Context

Zoom In with Context will show the contents of the selected module in a parent figure, and it will show its context from the previous diagram: the modules related to the selected modules. The other modules from the previous diagram are not shown.

The procedure is the same as with default zoom, with as difference that in step the Zoom In with Context option is selected.

The example below is the result of selecting package `husacct.control.presentation.workspace` in the diagram above and activation of Zoom In with Context.



Procedure:

- 1) Check the zoom option icon in the menu bar. It should display a magnifying glass with a 'c' in the center. If not, edit the zoom option setting with a right mouse click to 'Zoom in with context'.
- 2) Select a module in the diagram.
- 3) Zoom in: right mouse click => Zoom In, or click on menu bar icon, or click on icon in diagram options dialog.

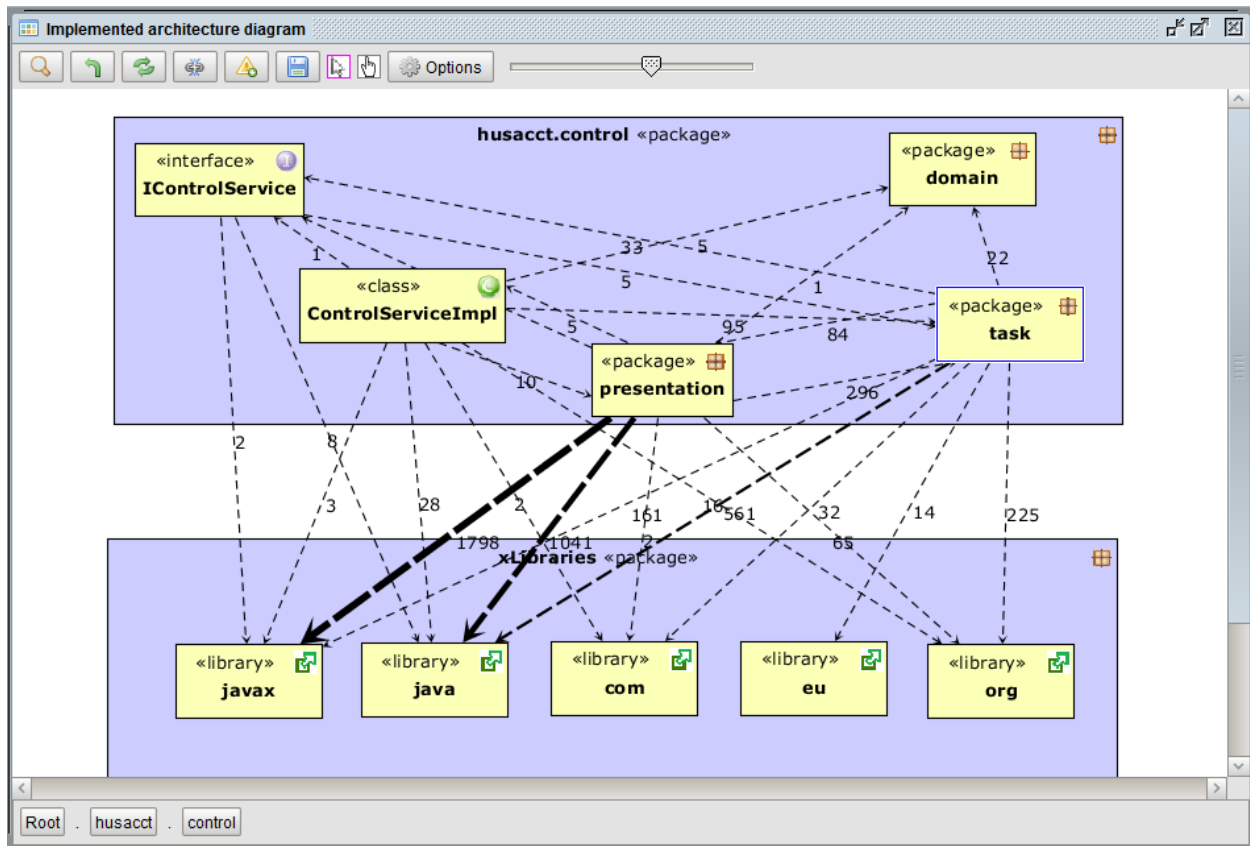
4.4.3.3 Multi zoom in

Multi zoom in will show the contents of the selected modules in the decomposition diagram and the dependencies between the submodules of the selected modules.

The procedure is the same as with default zoom, with as difference that in step 2 two or three modules or software units may be selected (hold the shift key to select the second or third module).

Example of a multi zoom in on husacct.control, while xLibraries was selected as well.

Furthermore, the option 'Proportional Line Width' is selected, and several software units are hidden.



4.4.4 BROWSE DEPENDENCIES & VIEW CODE

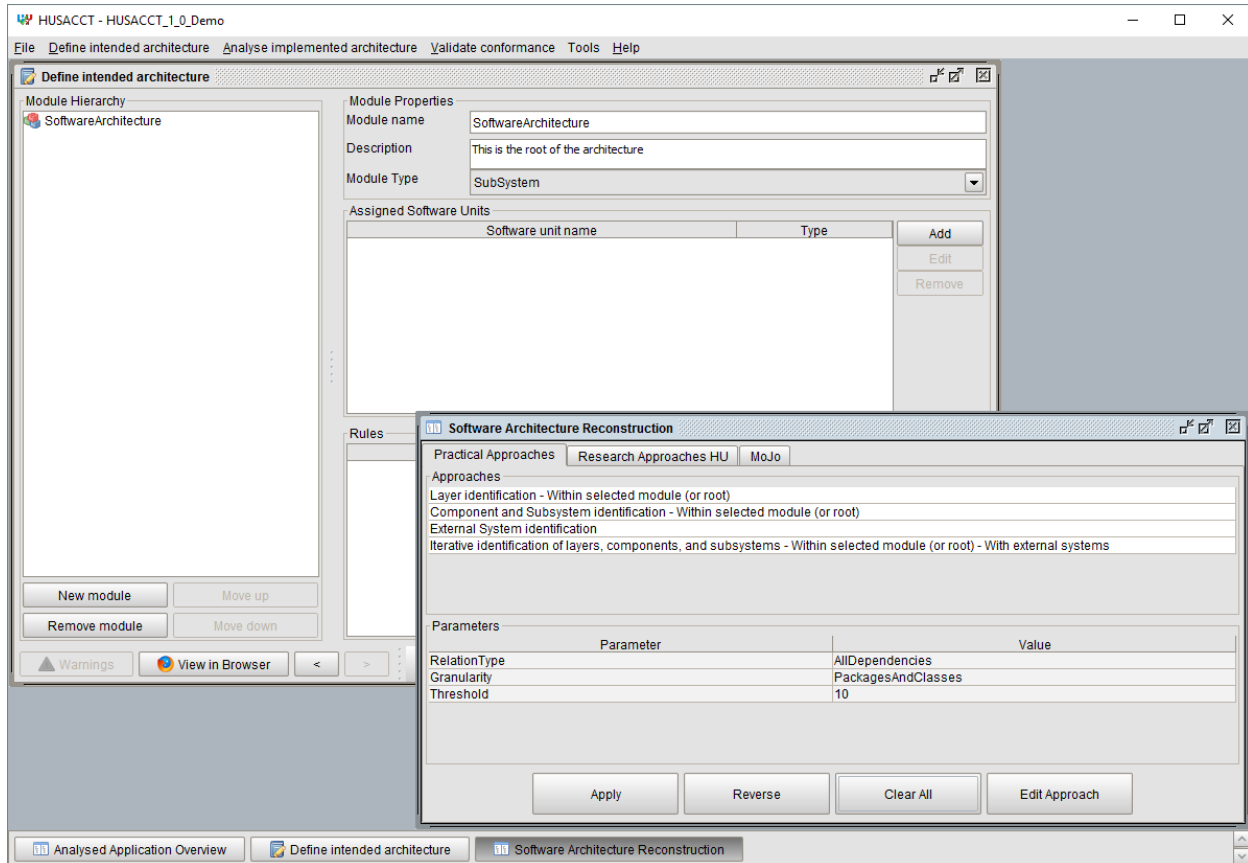
Selecting a dependency arrow will activate a dependency (or violation) table, which shows the dependencies or violations represented by the arrow.

A double click on a dependency will activate the code viewer. Within the code, the (violating) line is highlighted.

4.5 RECONSTRUCT ARCHITECTURE

To enable the reconstruct architecture functionality, a workspace has to be created and code has to be analyzed. Note that the intended architecture will be adjusted by this functionality, so start with a new workspace or save an existing workspace with another name.

When the menu item is enabled, the Software Architecture Reconstruction (SAR) view starts up, containing the Define frame and the SAR frame, as shown below.



4.5.1 CONCEPT, WORKFLOW AND APPROACHES

Software architecture reconstruction (SAR) techniques may be used to understand and maintain software systems, especially in these cases where architectural documentation is outdated or missing. Basic architecture reconstruction support is provided under other menu options of the Analyse menu. The packages and classes of a software system, and their interdependencies, can be browsed and visualized in diagrams. Make use of this functionality to check and steer the advanced SAR functionality. The *Reconstruct architecture* menu option provides advanced software architecture reconstruction functionality that can be used to identify potential logical modular structures, like layers and components, based on information in the source code. Different approaches (algorithms) can be selected and executed and the identified modules are added to the intended software architecture, shown in the Define frame. Each new module in the intended architecture is of a logical type. Software

units that implement the unit are assigned to the new module. Finally, for some module types, rules are generated automatically that constrain this type of module. For example, a rule of type back call ban is generated for a module of type *Layer*.

Software architecture reconstruction is not easy in many cases. Consequently, completely automated SAR may provide poor results. Human interpretation and intervention is often needed. The SAR GUI and the provided approaches enable human intervention. Support is provided to follow a step-by-step approach, where each step focuses on a certain position in the code tree. The position is defined by selection of a module in the intended architecture. The assigned software units of the selected module form the starting point for the selected approach, or in case there is one assigned software unit only, the children of this unit. If no module is selected, the approach takes the first set of software units in the root of the source code as starting point.

The software units and their dependencies at this starting point, are input to the selected approach's algorithm. After selection of *Apply* the algorithm is executed and the results can be studied in the Define frame. Next, the outcome can be:

- 1) Accepted completely
Continue by selection of a created module and an approach and apply the approach to reconstruct the child modules of the selected module.
- 2) Manually altered
Rename (one of) the new modules, change the type of the module, or edit the assignment of software units.
Continue thereafter as described above.
- 3) Reversed
Select the *Reverse* option and the newly created (child) modules and rules *of the last apply-action only* will be deleted.
Continue by selecting and applying another approach. Or create modules yourself.
- 4) Completely cleared
Select the *Clear All* option and all the modules and rules will be removed from the intended architecture.

Practical Approaches

The *Practical Approaches* are intended for practical use and are described below. The *Research Approaches HU* are of interest to researchers and developers, and these may still be in development.

Practical Approach	Description
Layer identification	The algorithm identifies potential layers and gives each layer a number and a name. Number identifies the bottom layer. The name is taken over from the assigned software unit. If more than one unit is assigned, the name of the largest unit is selected, while the other units are represented by _etc. A more in depth specification of the layering algorithm is described in [13].
Component and Subsystem identification	First, the algorithm determines for each of the assigned software units of the selected module (or code root if no module is selected) tree) if it represents a potential component. If one or more software

	units within a package implement an interface as intended by the facade pattern, then a component and an interface is created for the package as a whole, if all internal units are directly or indirectly accessed via the interface. If case only a subset is accessed directly or indirectly via the interface, than a component is created for this subset only, while the other units are assigned to a subsystem. Next, for packages not identified as component, a module of type <i>Subsystem</i> is created and the package is assigned to it. Several more detailed conditions and exceptions do apply.
External System identification	The algorithm creates modules of type External Library to represent external classes and packages used by the system.
Combined iterative approach	This approach combines the other approaches. At each position, it first tries to identify potential layers, then components and finally subsystems. In addition, external systems are identified.

Edit Approach

An approach may be parametrized, e.g. as in case of the Layer identification approach. In such a case the parameter settings may be edited. Select an approach and click on the *Edit Approach* button and the following dialogue pops up.

The implication of the different Parameters with their values is explained in the table below.

Parameter	Implication
Threshold	In case of layers the threshold value (in the range from 0-100) is used as BackCallThreshold: the percentage of back calls allowed between two software units in different layers. If the percentage of back calls in the code is higher than the threshold, the two software units are assigned to the same layer. The percentage of back calls from software unit su2 to software unit su1 is calculated as the number of dependencies n2 from su2 to su1, divided by the number of dependencies n1 from su1 to su2, multiplied by 100 (on condition that n2 is equal or smaller than n1, otherwise it is no back call from su2 to su1). Please note that the numbers of dependencies are influenced by the parameter TypesOfDependencies.

Relation type	<p>This parameter regulates the composition of the set of included dependencies in terms of allowed dependency types.</p> <p>We have distinguished three parameter settings:</p> <p>All types</p> <p>With this parameter setting, all seven types of dependency in object-oriented code, as distinguished by HUSACCT, are included: Access, Annotation, Call, Declaration, Import, Inheritance, and Reference [14].</p> <p>Only types Access, Call, and Reference</p> <p>With this parameter setting, only dependencies of the types Access, Reference, and Call are included. These three types have in common that they represent <i>executing</i> activities [14], which take care of the transformations (in contrast to dependencies that represent <i>preparing</i> activities) .</p> <p>Only types in Class Diagrams (UML Links)</p> <p>With this parameter setting, only dependencies are included that are depicted with special lines in UML Class diagrams, namely attribute-as-association, generalization, and interface implementation lines. In terms of our classification [14], dependencies of the following types are included: a) Declarations with subtype Instance Variable, and b) Inheritance, with all Extends and Implements variant subtypes.</p>
Granularity	<p>The granularity of the software units that will be used in an approach to identify modules in the intended architecture. The following values may be selected: PackagesAndClasses, Packages, or Classes.</p> <p>For example, if option PackagesAndClasses is selected in iteration 1 of the example above, than Layer3_Main is created that contains class Main only. If option Packages is selected, than no separate layer is created for the Main class. Instead the class is assigned to the layer with the generated name analyse_etc.</p>

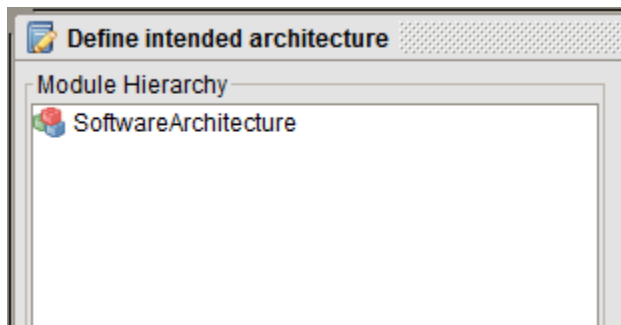
4.5.2 SAR EXAMPLE

The example below illustrates the process by means of the code of HUSACCT version 1.

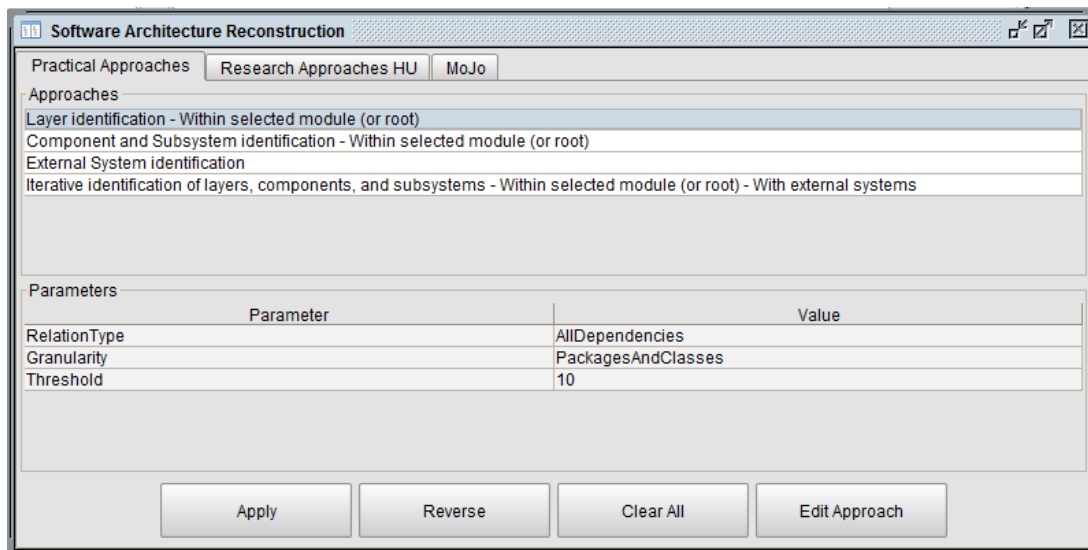
Note: The SAR approaches work quite well in this case, since HUSACCT's development is based on an intended architecture and, despite deviations, the architecture is still recognizable in the code.

Iteration1

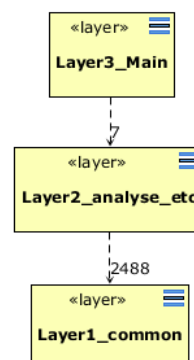
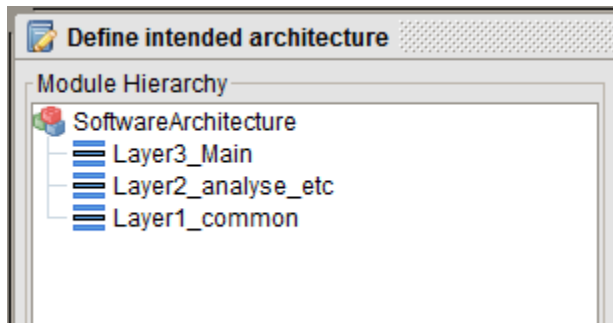
Starting point 1: Root



Selected approach 1: Layer identification

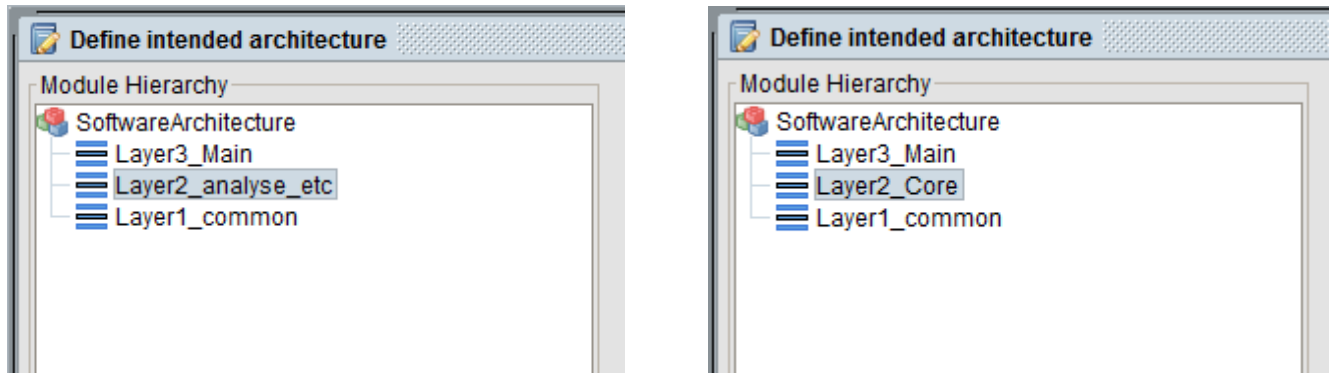


Result (after Apply) 1: Three layers



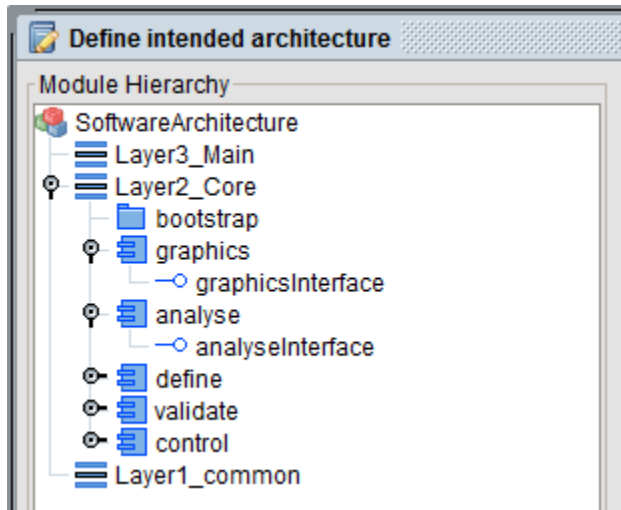
Iteration2

Starting point 2: Layer2_analyse_etc → Rename to: Core



Selected approach 2: Component and Subsystem identification

Result (after Apply) 2: Five components (each with an interface) and one subsystem



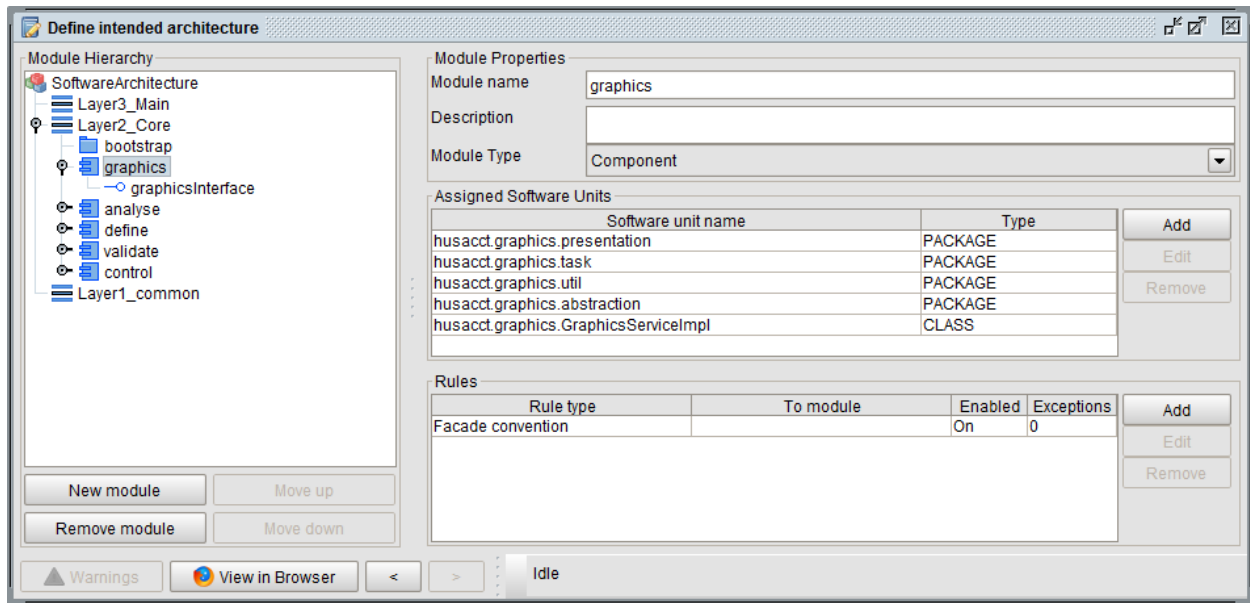
Iteration3

Starting point 3: Root

Selected approach 3: External System identification

Result (after Apply) 3: Six external systems

Intended Architecture after iteration 3

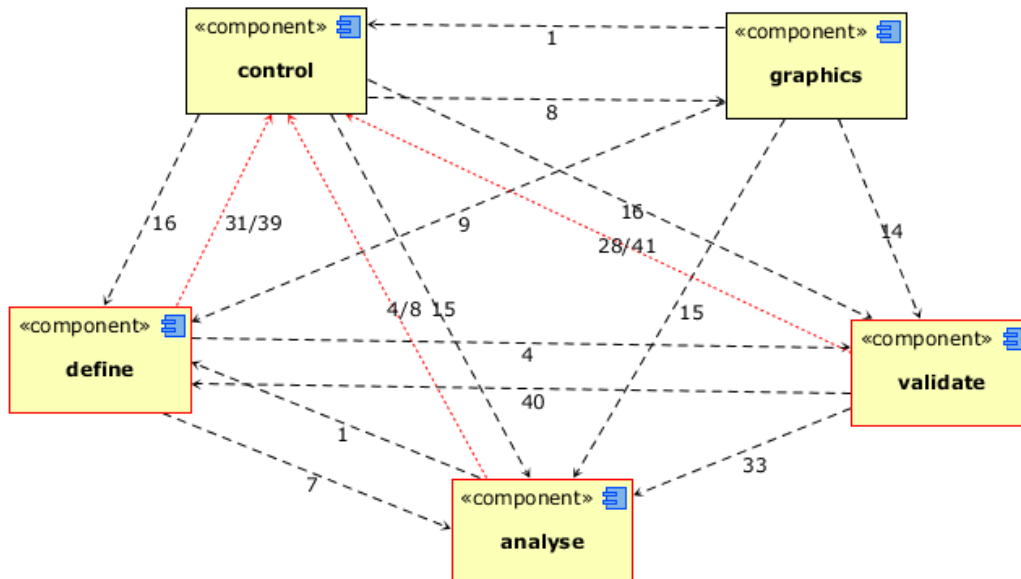


The figure above illustrates that:

- 1) software units are assigned to a module (five for component graphics);
- 2) for a component, an interface is created as well (graphicsInterface for component graphics; one class is assigned to graphicsInterface);
- 3) for a module of type *Component* a rule of type *Facade convention* is created.

Software Architecture Compliance Check

Based on the rules created for modules of type *Layer* or *Component*, a compliance check may be performed. The intended architecture diagram below shows several violations of rule *Facade convention* of component *control*.



4.5.3 MOJO

This Mojo functionality is intended for researchers to determine the effectiveness of an algorithm. The Mojo-tab makes it easy to calculate an effectiveness measure for software clustering algorithms based on Mojo distance as presented by Wen and Tzerpos in [15]. Their implementation (mojo.tar version 2.0, downloaded April 2016 from: <http://www.cs.yorku.ca/~bil/downloads/>) is used to calculate the MojoFM values displayed in the UI.

Wen and Tzerpos define the Mojo distance between two clusterings A and B of the same software system as the minimum number of Move or Join operations one needs to perform in order to transform either A to B or vice versa. The smaller the Mojo distance between an automatically created decomposition A and the “gold standard” decomposition B, the more effective the algorithm that created A.

To calculate the effectiveness of an approach conform the Mojo measure, two files are needed that describe the software clustering of a system in the format required by the MojoFM metric: 1) the “gold standard”; and 2) the clustering as a results of an algorithm. The idea behind the Mojo quality metric is that an algorithm that produces the farthest partition away from the “gold standard” results in a MojoFM value of 0%, while an algorithm that produces the “gold standard” results in a MojoFM value of 100%.

Export current intended architecture

The two files to be compared, can be created by means of the provided export functionality. First create an intended architecture; manually (e.g. the gold standard), or by execution of an algorithm. Next, export the architecture in the file format as expected by the MojoFM metric. Click on Browse in the export section. Choose a file path for the architecture to be saved. Once you’ve chosen a path you click on export to save the file in RSF format.

Software Architecture Reconstruction

Practical Approaches Research Approaches HU MoJo

Export current Intended Architecture

Export path: Browse

Export

Compare Architectures

Gold standard: Browse

To compare: Browse

Compare Architectures The calculated MojoFM value is: %

Compare architectures

Click the top Browse button after the field Gold standard in the Compare Architecture panel. Find and import the appropriate file. Next, click the bottom Browse button in the compare panel and select a file that describes the architecture that has been generated by an algorithm.

When both files are imported, click on the compare architectures button and the calculated MoJoFM value will be displayed.

Warning and HUSACCT specific extension

A characteristic of the MoJoFM calculation described by the author in the README file is the following:

“If the two decompositions do not refer to the same set of clustered objects, only the intersection of the two sets will be considered.”

Be aware of this characteristic!

To prevent that incompatible intended architectures with one overlapping cluster only (e.g. the cluster that contains xLibraries, which is generated by the algorithm for External system identification) results in a MoJoFM value of 100%, the MoJoFM code is extended in such a way that the result will be 0 % in this situation. The extension is implemented in `MoJoCalculator.mojofmValue(Vector, long, long)`.

4.6 ANALYSIS HISTORY

After analyzing any source code, HUSACCT saves information about the analysis in its App Data folder. This allows you to view the evolution of the software while your optimizing your code to get as few violations as possible. This analysis information looks like this:

Application Analysis History						
Application	Path	Date/Time	Packages	Classes	Interfaces	Dependencies
Java Benchmark	D:\Software\InstalledD...	31-01-45447 12:01:26	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	31-01-45447 11:01:12	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	31-01-45447 07:01:31	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	30-01-45447 08:01:31	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	29-06-45433 02:06:58	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	23-01-45447 09:01:01	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	23-01-45447 02:01:08	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	22-01-45447 06:01:24	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	22-01-45447 04:01:16	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	21-01-45447 03:01:31	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	17-04-45436 02:04:53	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	16-10-45392 01:10:25	13	149	30	821
Java Benchmark	D:\Software\InstalledD...	14-04-45436 11:04:18	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	14-01-45447 09:01:51	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	13-10-45392 10:10:14	13	149	30	821
Java Benchmark	D:\Software\InstalledD...	13-01-45447 05:01:36	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	13-01-45447 01:01:16	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	12-01-45447 10:01:08	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	11-01-45447 10:01:14	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	10-01-45447 04:01:44	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	09-02-45447 08:02:03	13	149	30	414
Java Benchmark	D:\Software\InstalledD...	09-02-45447 01:02:56	13	149	30	414

4.7 REPORT DEPENDENCIES (DEPENDENCY REPORT)

Select this menu option to create a spreadsheet file with all the detected dependencies within the application. Specify the location where the file will be stored, enter a file name and click on 'Report'. The dependency report contains a statistics sheet (see the figure below), which shows the numbers of dependencies per dependency type (in combination with direct/indirect) of the analysed application. Furthermore, numbers of inner class related dependencies and inheritance related dependencies are reported, as shown below. Furthermore, statistics per dependency subtype are provided.

Application: HUSACCT_1_0_Demo		Total	
Packages		104	
Classes		957	
Lines of Code		135923	
		Total	Direct Indirect
Dependencies, all		62630	58990 3640
Access		5582	5064 518
Annotation		0	0 0
Call		21075	19688 1387
Declaration		12961	12961 0
Import		3391	3391 0
Inheritance		921	815 106
Reference		18700	17071 1629
Inheritance related dependencies, all		2059	1032 1027
Inheritance relation		921	815 106
Access of inherited variable		585	97 488
Call of inherited method		364	120 244
Reference		189	0 189
Inner class related dependencies, all		13004	12998 6

On the next sheets, all dependencies are listed with their properties, as shown below.

Dependency from	Dependency to	Dependency type	Sub type	Line	Direct/Indirect	Inheritance Related	Inner Class Related
husacct.Main	husacct.ServiceProvider	Call	Class Method	15	Direct	false	false
husacct.Main	husacct.ServiceProvider	Call	Instance Method	15	Indirect	false	false
husacct.Main	husacct.control.IControlService	Declaration	Local Variable	15	Direct	false	false
husacct.Main	husacct.control.IControlService	Reference	Return Type	15	Indirect	false	false
husacct.Main	husacct.control.IControlService	Call	Interface Method	16	Direct	false	false
husacct.Main	husacct.control.IControlService	Call	Interface Method	17	Direct	false	false
husacct.Main	husacct.control.IControlService	Import		3	Direct	false	false
husacct.Main	xLibraries.java.net.URL	Declaration	Local Variable	21	Direct	false	false
husacct.Main	xLibraries.java.net.URL	Reference	Type of Variable	22	Direct	false	false
husacct.Main	xLibraries.java.net.URL	Import		5	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Call	Library Method	26	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Declaration	Local Variable	26	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Call	Library Method	30	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Call	Library Method	31	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Call	Library Method	32	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Call	Library Method	33	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.Logger	Import		7	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.PropertyConfigurator	Call	Library Method	22	Direct	false	false
husacct.Main	xLibraries.org.apache.log4j.PropertyConfigurator	Import		8	Direct	false	false
husacct.ServiceProvider	husacct.analyse.AnalyseServiceImpl	Import		3	Direct	false	false
husacct.ServiceProvider	husacct.analyse.AnalyseServiceImpl	Call	Constructor	54	Direct	false	false
husacct.ServiceProvider	husacct.analyse.IAnalyseService	Declaration	Instance Variable	26	Direct	false	false
husacct.ServiceProvider	husacct.analyse.IAnalyseService	Import		4	Direct	false	false
husacct.ServiceProvider	husacct.analyse.IAnalyseService	Reference	Type of Variable	54	Direct	false	false
husacct.ServiceProvider	husacct.analyse.IAnalyseService	Declaration	Return Type	69	Direct	false	false
husacct.ServiceProvider	husacct.analyse.IAnalyseService	Reference	Type of Variable	70	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.ILocaleService	Declaration	Instance Variable	24	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.ILocaleService	Reference	Type of Variable	53	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.ILocaleService	Import		5	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.ILocaleService	Declaration	Return Type	61	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.ILocaleService	Reference	Type of Variable	62	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.LocaleServiceImpl	Call	Constructor	53	Direct	false	false
husacct.ServiceProvider	husacct.common.locale.LocaleServiceImpl	Import		6	Direct	false	false

4.8 EXPORT/IMPORT ANALYSED MODEL

Select the Export menu option to create an xml file with all the detected packages, classes, libraries and dependencies within the application. Specify the location where the file will be stored, enter a file name and click on 'Export'.

Select the Import menu option to import in the repository all the packages, classes, libraries and dependencies within an assigned xml file. Specify the location where the file is stored and click on 'Import'. Selecting this option, after analysis of the source code, will overwrite the data already existing in the repository.

Both functions may take several minutes in case of big applications with e.g. 800K lines of code.

5 MENU: VALIDATE CONFORMANCE

5.1 VALIDATE NOW

Validate conformance checks if rules defined in the intended architecture are violated in the implemented architecture.

The following preconditions apply:

- The source code is analyzed;
- Logical modules are defined;
- The classes or packages of the source code are assigned to the defined logical modules;
- The intended architecture contains rules.

When the validation process has finished, the results are shown in the from 'Validate conformance', with two tabs: 1) Violations per Rule; and 2) All Violations.

5.1.1 VIOLATIONS PER RULE

This view provides a summary of the violations against the rules.

The upper table shows the violated rules and the number of violations per rule. When a rule in this table is selected, the violations are shown in the bottom table.

A double click on a dependency will activate the code viewer. Within the code, the violating line is highlighted in red.

Validate conformance

Violations Per Rule

All Violations

Rules with Number of Violations

Id	Logical module from	Rule type	Logical module to	Violations
1	Analyse	Is not allowed to use	Define	1
2	Define.Presentation	Is not allowed to skip call		10
3	Define.Task	Is not allowed to back call		99
4	Define.Task	Is not allowed to skip call		24
5	General GUI & Control	Facade convention		38

Violations

From	To	Rule type	Dep.type	Direct	Line
husacct.define.presentation.jdialog.AppliedRuleJD...	husacct.define.domain.SoftwareUnitDefinition	Is not allowed to skip call	Import	Direct	7
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition.Type	Is not allowed to skip call	Access	Direct	57
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition.Type	Is not allowed to skip call	Declaration	Direct	57
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition.Type	Is not allowed to skip call	Access	Direct	58
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition.Type	Is not allowed to skip call	Import	Direct	7
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition.Type	Is not allowed to skip call	Call	Direct	57
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition	Is not allowed to skip call	Import	Direct	6
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition	Is not allowed to skip call	Access	Direct	59
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition	Is not allowed to skip call	Call	Direct	58
husacct.define.presentation.moduletree.Combine...	husacct.define.domain.SoftwareUnitDefinition	Is not allowed to skip call	Declaration	Direct	58

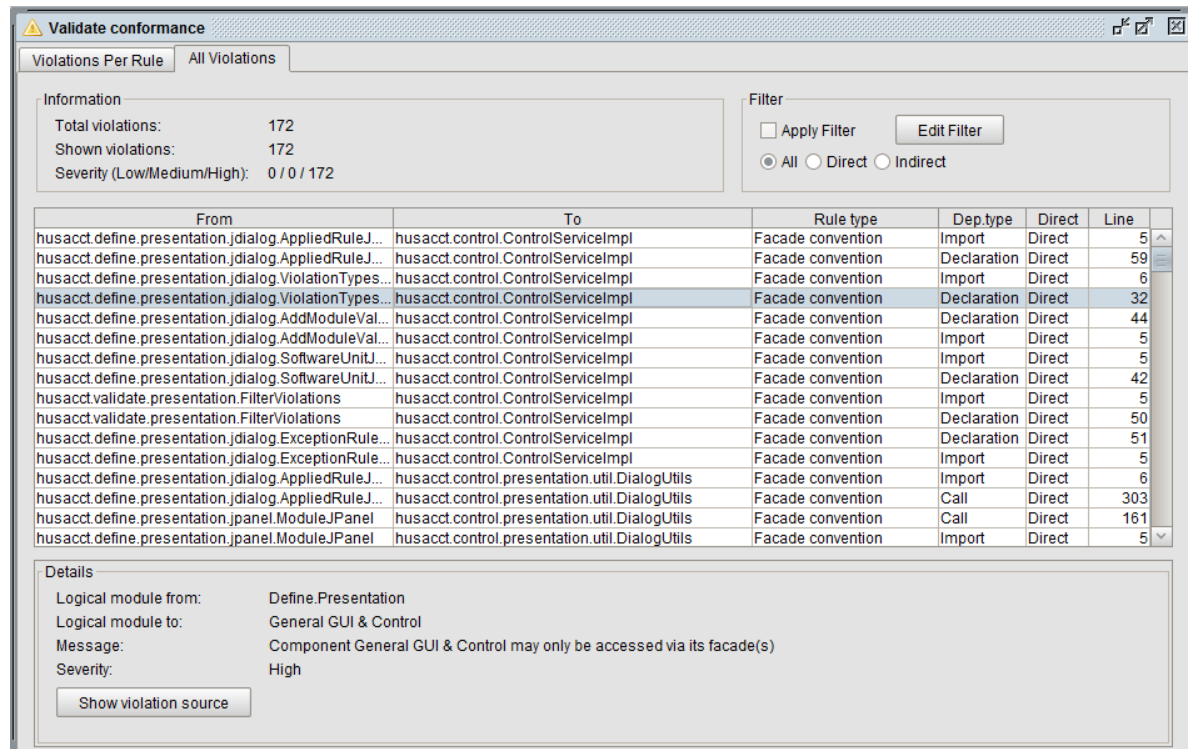
5.1.2 ALL VIOLATIONS

This view shows all violations. When a violation is selected, details are shown in the bottom panel.

A double click on a dependency, or a click on the button 'Show violation source', will activate the code viewer. Within the code, the violating line is highlighted in red.

Additional features are:

- Sorting: The violations can be sorted by clicking on the headers, e.g. 'Rule type'.
- Filtering: Explained below.



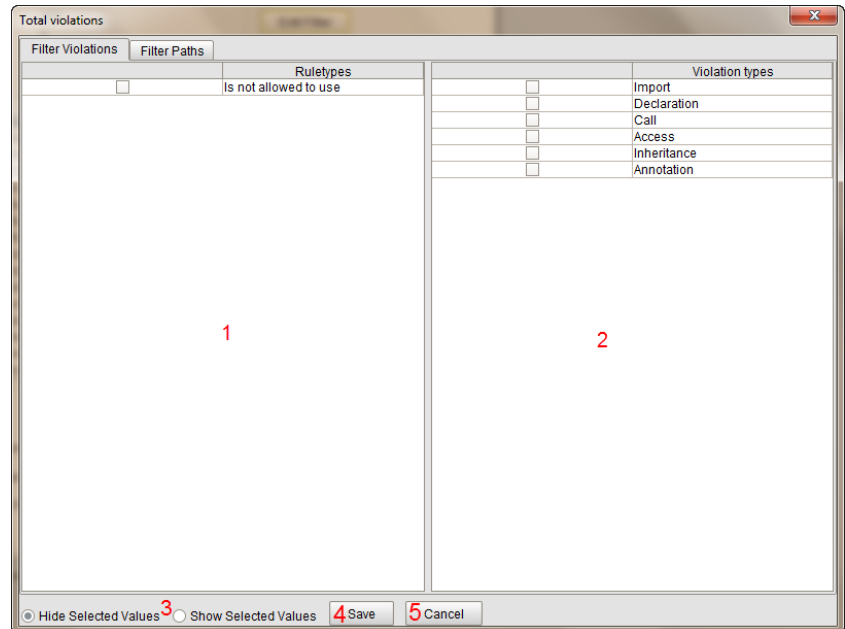
1. With the checkbox 'Apply Filter', the filter can be turned on or off.
2. When the button 'Edit Filter' is pressed the Filter dialog is opened.
3. The three radio buttons provide a filter to switch between: 1) all kinds of dependencies; 2) only direct dependencies; or 3) only indirect dependencies. This filter only works when the checkbox 'Apply Filter' is marked.

5.1.2.1 Filter dialog

This screen provides an easy to use way to filter the violations. During filtering can be chosen to show or hide the selected values. There are two tabs in this screen. In the first tab only the rule types and violation types are shown. In the second tab there is a possibility to filter on the source path of the violations.

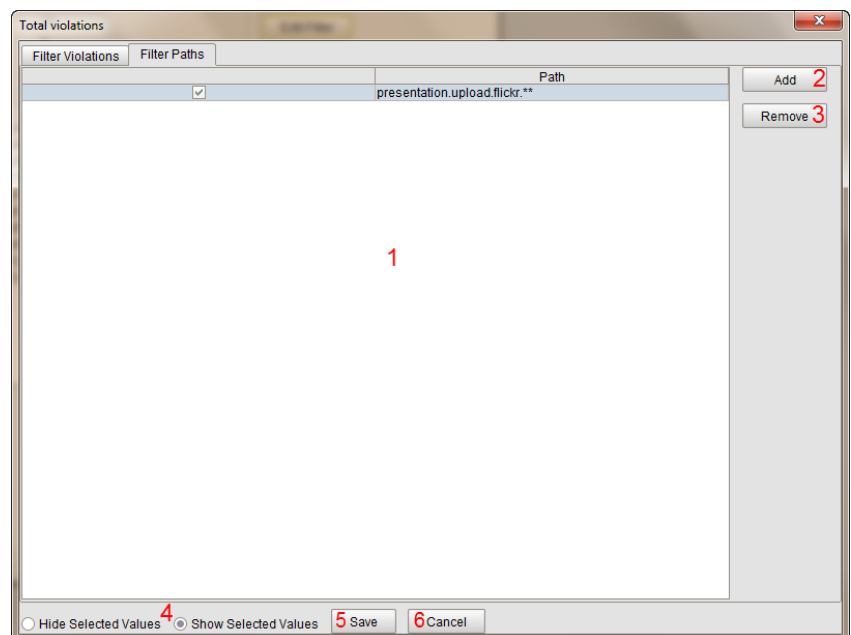
FILTER DIALOG TAB FOR FILTERING RULE- AND/OR VIOLATION TYPES

1. In this table the rule types that will be filtered can be selected.
2. In this table the violation types that will be filtered can be selected.
3. The option to choose if the filtered values must be shown or must be hidden.
4. When this button is pressed, the violations will be filtered.
5. When this button is pressed, the screen will be closed.



FILTER DIALOG TAB FOR FILTERING CLASSPATHS

1. This table shows all the paths are added to filter. The physical paths can be filtered with a regex; the possibilities will be explained in table 2.
2. When this button is pressed, the system will add an empty field to area 1.
3. When this button is pressed, the system will remove the selected row from area 1.
4. The option to choose if the filtered values must be shown or must be hidden.
5. When this button is pressed, the violations will be filtered.
6. When this button is pressed, the screen will be closed.

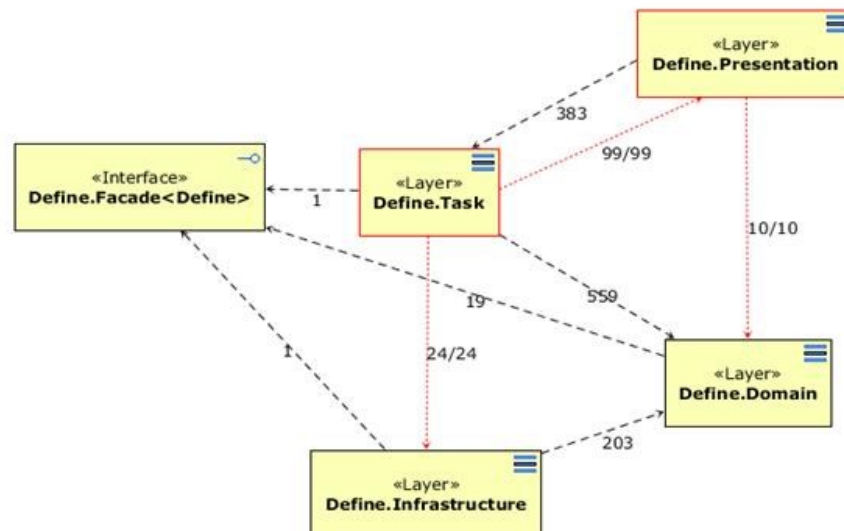


OVERVIEW OF POSSIBLE REGEXES FOR FILTERING

Your Input Example	Path Input Example	Output Example
java.St*	java.String java.StringBuffer java.String.Fake com.class.String	java.String java.StringBuffer
java.St**	java.String java.StringBuffer java.String.Fake com.class.String	java.String java.StringBuffer java.String.Fake
*.String	java.String java.StringBuffer java.String.Fake com.class.String	java.String
**.*String	java.String java.StringBuffer java.String.Fake com.class.String	java.String com.class.String
Stri	java.String java.StringBuffer java.String.Fake com.class.String	java.String java.StringBuffer java.String.Fake com.class.String

5.1.3 VIOLATIONS IN DIAGRAMS

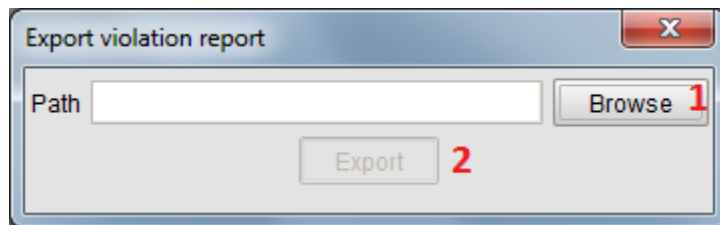
Violations may be shown in both types of diagrams, intended and implemented architecture diagrams. When the option 'Show violations' is selected, violations are made visible in the diagrams. Red dependency arrows and red modules mark violations against the rules. A red dependency arrow will show two numbers, the first represents the number of violating dependencies, while the second represents the total number of dependencies.



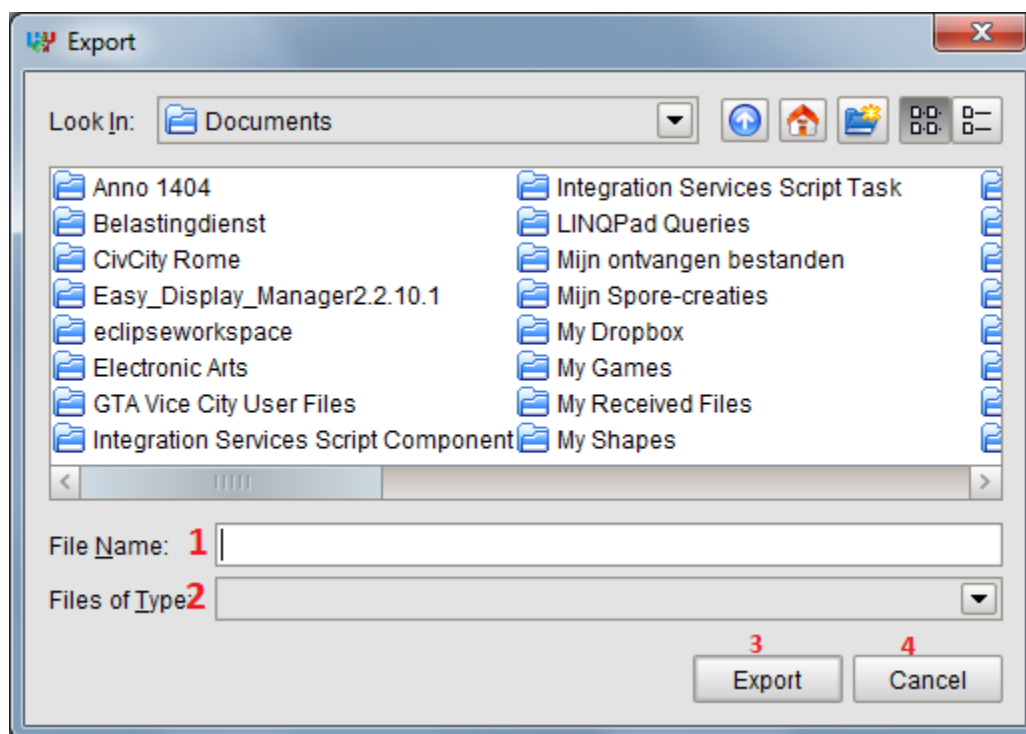
5.2 VIOLATIONS EXPORT AND REPORT

The violations can be exported in XML format and reported as the following file types: xls, html, and pdf. The xls option will generate a spreadsheet with the number of violations per rule, a sheet with all rules, and a sheet with statistics on the frequency of dependency types and subtypes over all violations. The html and pdf options will generate a report with an overview of all violations. The html report is sortable on different characteristics of the violations.

PROCEDURE



1. When this button is pressed, it opens an export dialog. (see figure 5)
2. When this button is pressed, the report will be exported.



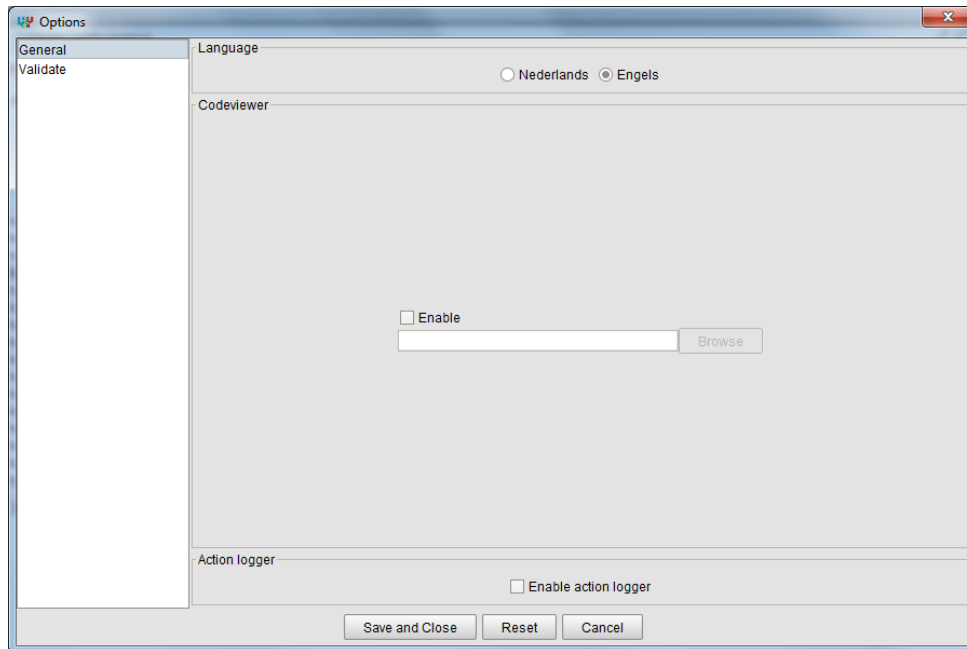
1. Select the file name and directory where the file must be saved.
2. Select the type of the file.
3. When this button is pressed, the path will be set in figure 4.
4. When this button is pressed, the screen will be closed.

6 MENU: TOOLS

6.1 OPTIONS

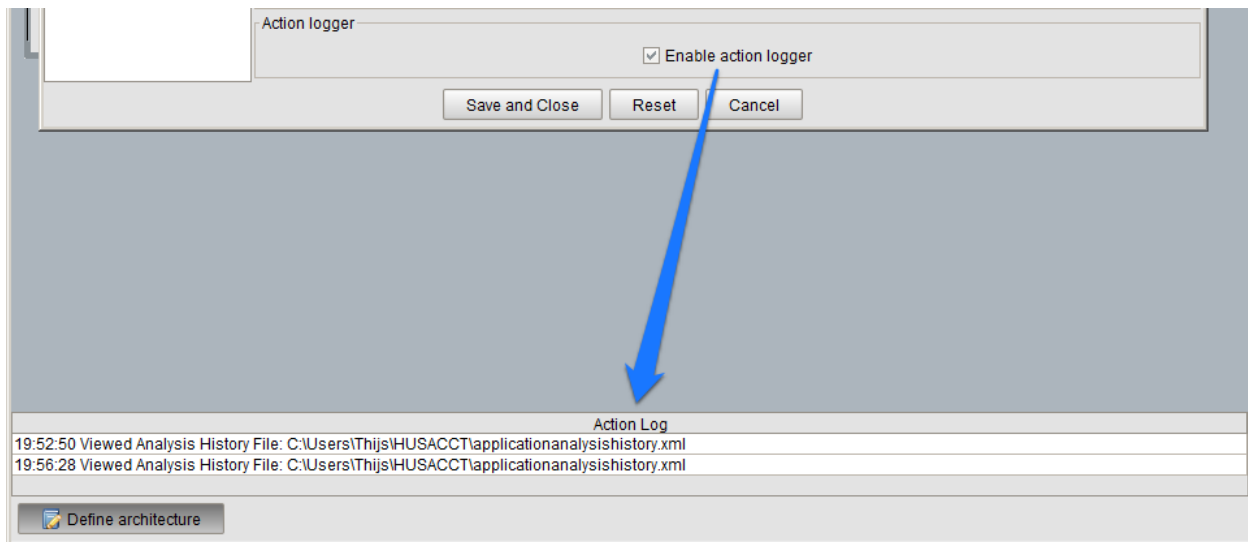
6.1.1 GENERAL

The general options allow you to: 1) select a language for the user interface of HUSACCT; 2) select a different code viewer than the build-in code viewer; 3) enable the action logger.



6.1.1.1 ACTION LOG

The action log will show actions you've taken in HUSACCT. For example, after analysis of an application a message appears in the Action Log showing the start and end time, as well as some statistics of the results. The option is disabled by default and can be enabled in Tools -> Options -> General.



6.1.2 VALIDATE – CONFIGURATION

The validate component provides also the possibility to configure the following:

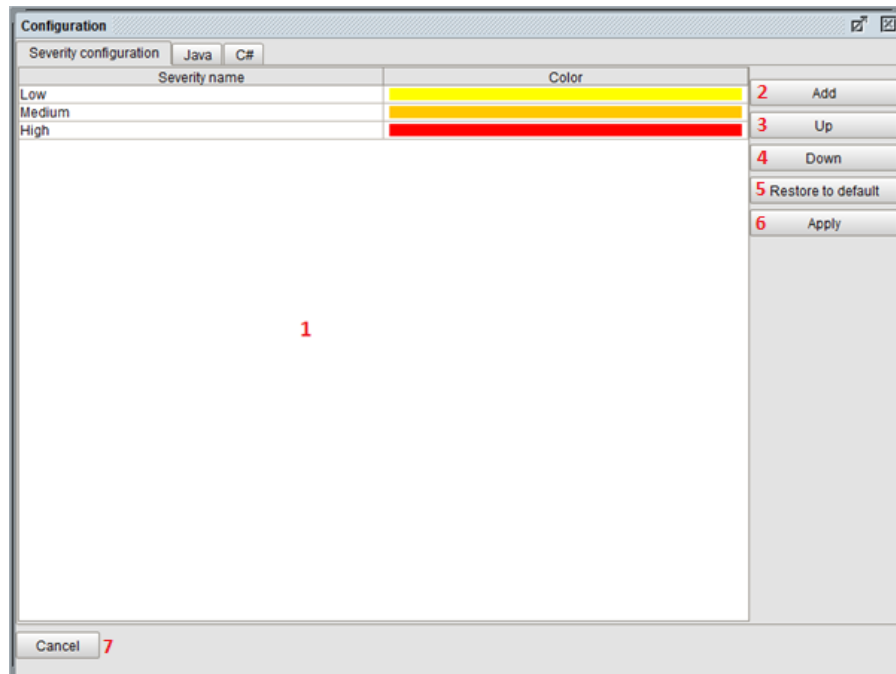
- Severities; adding new severities and change existing severities.
- Severity per rule type; change the severity for a rule.
- Severity per violation type; change the severity for a violation type.
- Active violation types; which violation types should be enabled by default in the filter when adding new rules in the define component. (For more information about filtering in rule types see the manual of the define component.)

The changes that are made are not directly reflected in the GUI. For example when severities are changed in the configuration, there must be revalidated to reflect the changes that are made in the configuration.

The screen is available once a workspace has been created. The configuration screen is available through “menu -> Validate -> Configuration”.

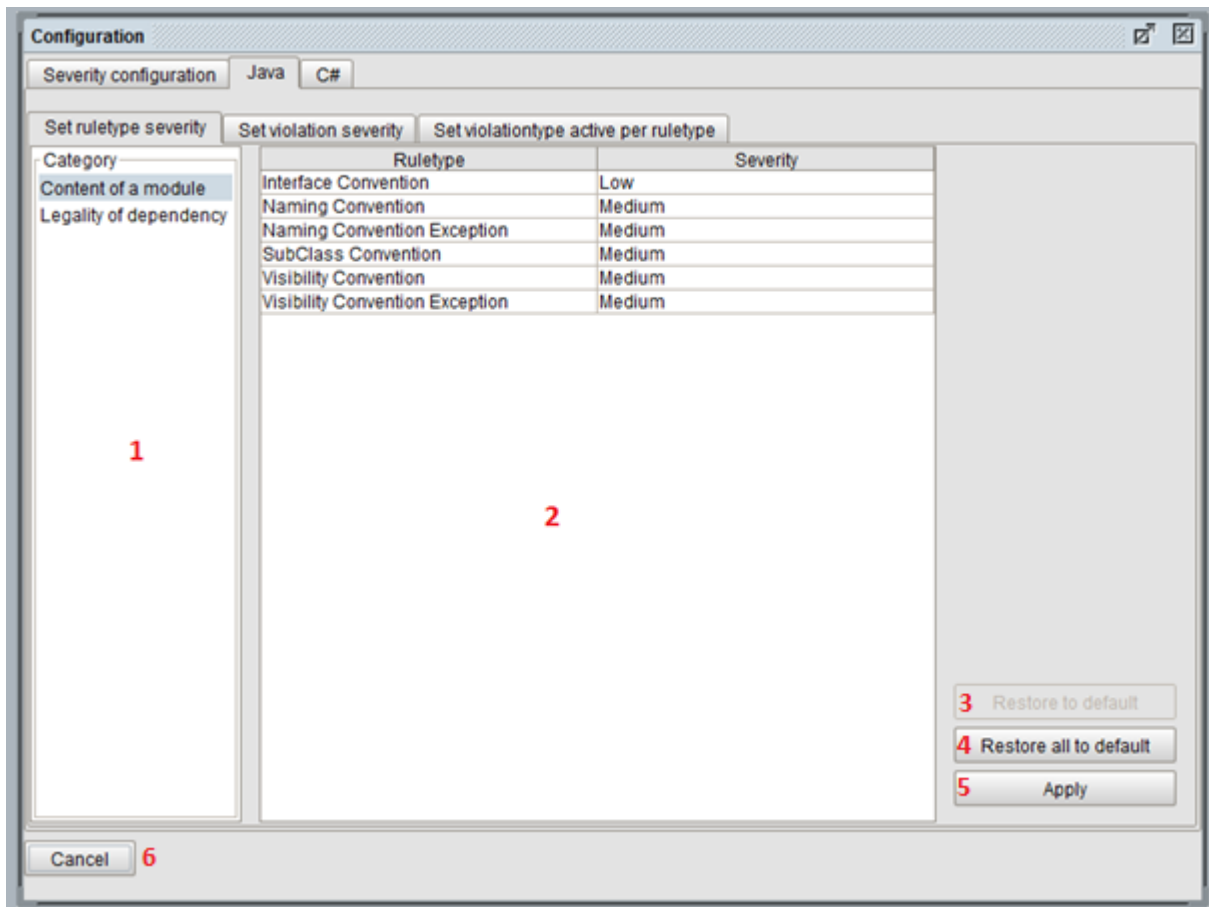
6.1.2.1 *Configure severities*

There are two or more tabs on the screen. The first is to create and edit severities. For every supported programming language a tab will be shown. Inside each of ‘programming language’ tab will contain three other tabs. The first two tabs are for setting the severity per rule/ violation types. The last tab is to set the active violation types.



1. This table lists all the severities. It is possible to change the name and/or color of a severity. The color can be changed by clicking on the color bar and a color picker dialog will pop up. The severities are in order from lowest to highest.
2. When this button is pressed, an empty severity will be added on the lowest place.
3. When this button is pressed, the selected severity will be moved up in the list.
4. When this button is pressed, the selected severity will be moved down in the list.
5. When this button is pressed, all the severities will be restored to the default severities.
6. When this button is pressed, all the changes in this tab will be saved.
7. When this button is pressed, the screen close without saving.

6.1.2.2 *Configure severities per rule type and per violation type*

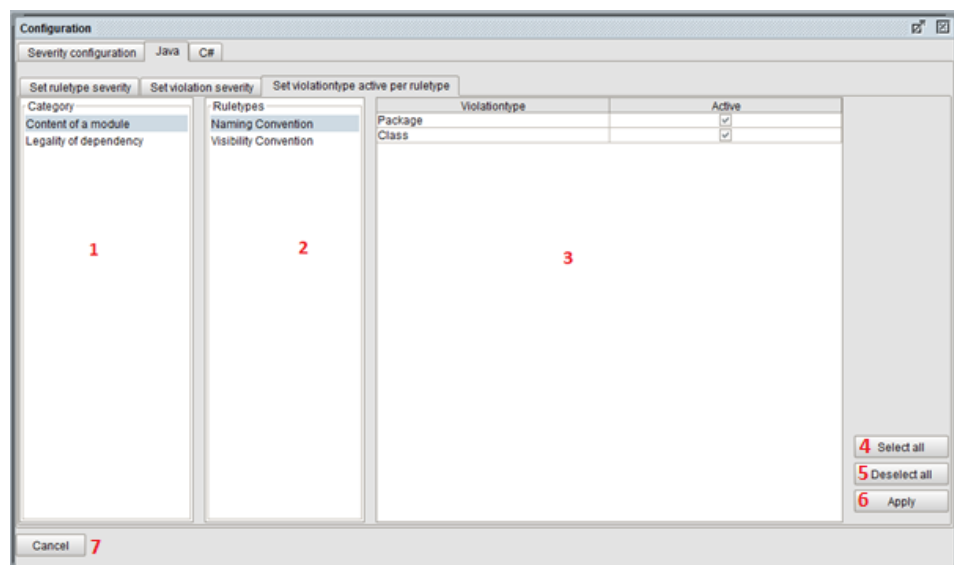


This screen applies for the first two tabs.

1. List of categories.
2. Table with rule types or violation types. Changed by selecting a different category in area 1. You can change the severity by clicking on the severity name.
3. When this button is pressed, the selected rule type will be restored to its default severity level.
4. When this button is pressed, all the rule/ violation types will be restored to their default value.
5. When this button is pressed, all the changes in this tab will be saved.
6. When this button is pressed, the screen close without saving.

6.1.2.3 *Configure active violation types*

1. List of categories.
2. List of rule types.
Changes when you select another category.
3. Table of active violation types.
Changes when

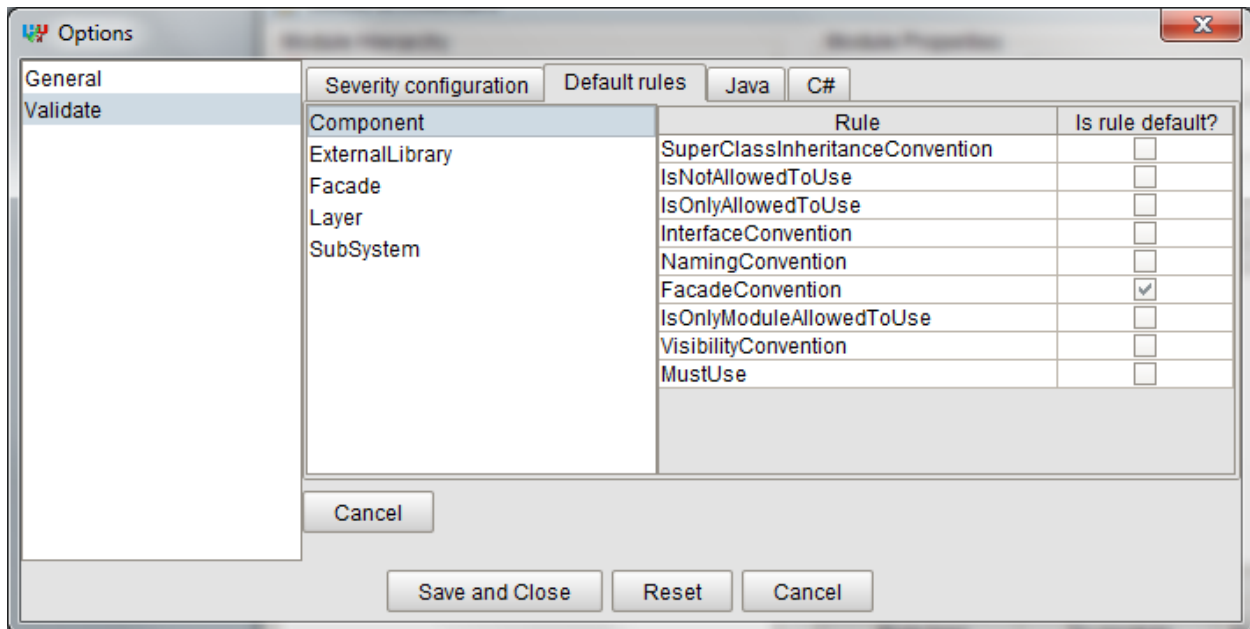


you select another rule type.

4. Select all active violation types.
5. Deselect all active violation types.
6. Save all the changes in this tab.
7. Close without saving.

6.1.2.4 Configuration of the default rules

Within the configuration, you can define what rules should be added by default, to a new component in the software architecture definition. Opening the “Default rules” tab, will lead you to a list of allowed rules per component. Select a component to set or unset the default rules in the component.



There are a few rules selected by default. These can be deselected if required. These settings are instantly saved, so no save and close are required.

7 LITERATURE

1. Pruijt L, Köppe C, Brinkkemper S. Architecture compliance checking of semantically rich modular architectures: a comparison of tool support. In: 2013 IEEE International Conference on Software Maintenance. IEEE, 2013; 220-229. DOI:10.1109/ICSM.2013.33.
2. Pruijt L. Instruments to Evaluate and Improve IT Architecture Work. Utrecht, Netherlands, 2015.
3. Clements P, Bachmann F, Bass L, et al. Documenting Software Architectures: Views and Beyond. Pearson Education, 2010.
4. Pruijt L, Köppe C, van der Werf JM, Brinkkemper S. Husacct: architecture compliance checking with rich sets of module and rule types. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering - ASE '14. ACM, 2014; 851-854. DOI:10.1145/2642937.2648624.
5. Pruijt L, Brinkkemper S. A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking. In: WICSA 2014 Companion Volume. ACM, 2014; 1-8. DOI:10.1145/2578128.2578233.
6. Larman C. Applying UML And Patterns. Prentice Hall PTR, 2005.
7. Parnas DL. On the criteria to be used in decomposing systems into modules. Communications of the ACM 1972; **15**(12):1053-1058.
8. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 1995.
9. Perry DE, Wolf AL. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 1992; **17**:40-52. DOI:10.1145/141874.141884.
10. Pruijt L, Köppe C, Brinkkemper S. On the accuracy of architecture compliance checking: accuracy of dependency analysis and violation reporting. In: 21st International Conference on Program Comprehension. San Francisco, CA, USA: IEEE, 2013; 172-181. DOI:10.1109/ICPC.2013.6613845.
11. Podgurski A, Clarke LA. A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering 1990; **16**(9):965-979. DOI:10.1109/32.58784.
12. Callo Arias TB, Spek P, Avgeriou P. A practice-driven systematic review of dependency analysis solutions. Empirical Software Engineering 2011; **16**(5):544-586. DOI:10.1007/s10664-011-9158-8.
13. Pruijt L, Wiersema W. Dependency related parameters in the reconstruction of a layered software architecture. In: ECSAW '16. , 1916; 1-7.
14. Pruijt L, van der Werf JMEM. Dependency types and subtypes in the context of architecture reconstruction and compliance checking. In: 2015 European Conference on Software Architecture Workshops Proceedings. ACM, 2015; 1-7. DOI:10.1145/2797433.2797491.
15. Wen ZWZ, Tzerpos V. An effectiveness measure for software clustering algorithms. In: 12th IEEE International Workshop on Program Comprehension. IEEE, 2004; 194-203. DOI:10.1109/WPC.2004.1311061.
16. Passos L, Terra R, Valente MT, Diniz R, Das Chagas Mendonca N. Static architecture-conformance checking: an illustrative overview. IEEE Software 2010; **27**(5):82-89. DOI:10.1109/MS.2009.117.

17. Knodel J, Popescu D. A comparison of static architecture compliance checking approaches. In: Working IEEE/IFIP Conference on Software Architecture. IEEE, 2007; 12-21. DOI:10.1109/WICSA.2007.1.
18. Sarkar S, Rama GM, R S. A method for detecting and measuring architectural layering violations in source code. In: Asia-Pacific Software Engineering Conference (APSEC). IEEE Computer Society Press, 2006; 165-172. DOI:10.1109/APSEC.2006.7.