System Documentation

Define Architecture

6/7/2012

Hogeschool Utrecht

**Team 4 - Define**  
Henk ter Harmsel  
Alex Schouls  
Dennis van den Waardenburg  
Bob Sanders

Table of Contents

[Introduction 3](#_Toc327351163)

[Functionality 4](#_Toc327351164)

[Use Case Model 4](#_Toc327351165)

[Goal description per Use Case 5](#_Toc327351166)

[Define logical architecture 5](#_Toc327351167)

[Define rules 5](#_Toc327351168)

[Map logical architecture to software units 5](#_Toc327351169)

[Import logical architecture 5](#_Toc327351170)

[Export logical architecture 5](#_Toc327351171)

[Import workspace 5](#_Toc327351172)

[Export workspace 5](#_Toc327351173)

[Details per use case 6](#_Toc327351174)

[Define Logical Architecture 6](#_Toc327351175)

[Define rules 7](#_Toc327351176)

[Map logical architecture to physical architecture 8](#_Toc327351177)

[References 9](#_Toc327351178)

[Define Logical Architecture 9](#_Toc327351179)

[Define rules 9](#_Toc327351180)

[Map logical architecture to software units 9](#_Toc327351181)

[Sequence diagrams 10](#_Toc327351182)

[Define logical architecture 10](#_Toc327351183)

[Define rules 10](#_Toc327351184)

[Map logical architecture to physical architecture 11](#_Toc327351185)

[Decisions and justifications 12](#_Toc327351186)

[Software partitioning 13](#_Toc327351187)

[Physical Class Diagram 14](#_Toc327351188)

[Subsystem specification 15](#_Toc327351189)

[Module Tree 15](#_Toc327351190)

[Classes 15](#_Toc327351191)

[Sequence Diagram 16](#_Toc327351192)

[Rule Details 16](#_Toc327351193)

[Classes 16](#_Toc327351194)

[Class diagram 17](#_Toc327351195)

[Contradictory rule conventions checker 18](#_Toc327351196)

[Classes 19](#_Toc327351197)

[Sequence Diagram 19](#_Toc327351198)

[Domain model 20](#_Toc327351199)

[Classes 20](#_Toc327351200)

[Information model 20](#_Toc327351201)

[Domain Parser 21](#_Toc327351202)

[Sequence Diagram 21](#_Toc327351203)

[How-To-Add 22](#_Toc327351204)

[New Rule 22](#_Toc327351205)

[GUI 22](#_Toc327351206)

[Factory 23](#_Toc327351207)

[Testing 24](#_Toc327351208)

[Future work 25](#_Toc327351209)

[Known bug list 25](#_Toc327351210)

[Ideas for improvement 25](#_Toc327351211)

[Ideas for extension 26](#_Toc327351212)

# Introduction

This document contains the system documentation of the Define component of the HUSACCT tool.

HUSACCT stands for Hogeschool Utrecht Software Architecture Compliance Checking Tool. Hogeschool is Dutch for university of applied sciences.  
This tool is divided in five main components.

* Control architecture
* Define architecture
* Validate architecture
* Analyse application
* Graphics

This document will elaborate on the Define component.  
The Define component is able to define an architecture, import and export that architecture.  
It’s also able to map the defined architecture to a physical architecture.

The development team of this component consisted of four members:

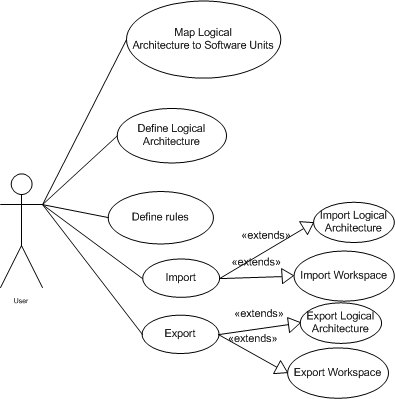
* Bob Sanders
* Henk ter Harmsel
* Dennis van den Waardenburg
* Alex Schouls

# Functionality

This section will elaborate on the use cases.  
There are seven architectural significant use cases:

* Define logical architecture
* Define rules
* Map logical architecture to software units
* Import logical architecture
* Export logical architecture
* Import workspace
* Export workspace

## Use Case Model



## Goal description per Use Case

### Define logical architecture

The goal of this use case is to make the user able to define a logical architecture.  
This means defining architectural components (layers, subsystems etc.)

### Define rules

This use case complements “Define logical architecture” for the goal here is to define rules on the defined components of the architecture.

### Map logical architecture to software units

This use case’s purpose is to map the defined architecture (See also: Define logical architecture) to the analysed software units. Note that this can only be done when an application has been analysed.

### Import logical architecture

Import logical architecture’s purpose is to import an existing architecture for future use or mapping.

### Export logical architecture

Export logical architecture’s purpose is to export the logical architecture for storage, therefor increasing the reusability of the logical architecture, since it can be used for multiple applications.

### Import workspace

Importing the workspace will import all properties of a saved project.

### Export workspace

Exporting a workspace will save all properties of a project.

## Details per use case

Not all use cases require elaboration. This part will elaborate on:

* Define logical architecture
* Define rules
* Map logical architecture to physical architecture

### Define Logical Architecture

|  |  |
| --- | --- |
| Use Case |  |
| Number: | 1.0 |
| Version: | 2.0 |
| Writer | Team 4 - Define |
| Priority | Must |
| Use Case | Define Architecture 🡪 Define Logical Architecture |
| Actors | User |
| Summary | It’s possible to define the desired logical architecture to check the architecture. |
| Precondition | Actor has created a workspace |
| Main scenario | |  |  | | --- | --- | | **Actor actions** | **System actions** | |  | 1. The system shows the GUI to define the architecture. | | 1. Actor clicks “New module”. 2. Actor fills in the module details, and its type. 3. Actor clicks on “Save”. | 1. The system adds the module to the list. | |
| Post condition | The logical architecture is now defined and can be saved or imported. |
| Transformation rules |  |
| Constraints |  |

### 

### Define rules

|  |  |
| --- | --- |
| Use Case |  |
| Number: | 5.0 |
| Version: | 2.0 |
| Writer | Team 4 - Define |
| Priority | Must |
| Use Case | Define Architecture 🡪 Define Rules |
| Actors | User |
| Summary | Define a rule for a specific module |
| Precondition | Modules are defined for applying the rule on |
| Main scenario | |  |  | | --- | --- | | **Actor actions** | **System actions** | |  | 1. The system shows the GUI to define the architecture. | | 1. Actor selects an already defined module. |  | | 1. Actor clicks on “Add” under the “Rules” section. |  | |  | 1. The system shows the popup screen for adding a new rule | | 1. Actor selects the rule type, the module it applies to, whether it’s enabled or not and possible exceptions. |  | | 1. Actor clicks on “Add” | 1. The system creates the rule and adds it to the list of rules | |
| Post condition | The rule is defined and applied to the specific module. |
| Transformation rules |  |
| Constraints |  |

### Map logical architecture to physical architecture

|  |  |
| --- | --- |
| Use Case |  |
| Number: | 2.0 |
| Version: | 2.0 |
| Writer: | Team 4 - Define |
| Priority | Must |
| Use Case | Define Architecture 🡪 Map Logical Architecture to Physical Architecture |
| Actors | User |
| Summary | This use cases describes mapping of the logical defined architecture to the physical architecture of an application. |
| Precondition | An architecture has to be defined and an application has to be analysed |
| Main scenario | |  |  | | --- | --- | | **Actor actions** | **System actions** | |  | 1. The system shows the GUI to define the architecture. | | 1. Actor either defines the logical architecture or imports an existing one 2. Actor selects a module which he/she would like to map. 3. Actor clicks on “Add” under the “Software Units” section. 4. Actor selects the software unit he/she would like to map to the selected module. 5. Actor clicks on “Assign” | 1. The system shows the popup screen for assigning a software unit. 2. The system maps the logical architecture to the software units | |
| Post condition | The defined logical architecture must be used and mapped to the software units present in the application. |
| Transformation rules |  |
| Constraints |  |

## References

### Define Logical Architecture

|  |  |
| --- | --- |
| **Path** | **Type** |
| Define.presentation.jpanel.ModuleJPanel | Class |
| Define.presentation.jpanel.EditModuleJPanel | Class |
| Define.presentation.jdialog.AddModuleValuesJDialog | Class |
| Define.task.DefinitionController | Class |
| Define.domain.services.ModuleDomainService | Class |
| Define.domain.SoftwareArchitecture | Class |
| Define.domain.module | Package |

### Define rules

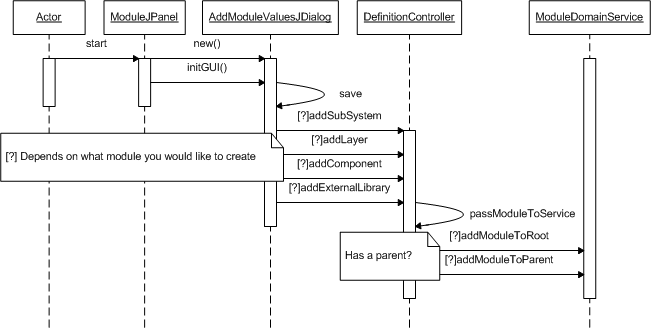
|  |  |
| --- | --- |
| **Path** | **Type** |
| Define.presentation.jpanel. AppliedRulesJPanel | Class |
| Define.presentation.jpanel.ruledetails | Package |
| Define.presentation.moduleTree | Package |
| Define.presentation.jdialog.AppliedRuleJDialog | Class |
| Define.presentation.jdialog.ExceptionRuleJDialog | Class |
| Define.task.DefinitionController | Class |
| Define.task.AppliedRuleController | Class |
| Define.domain.services. AppliedRuleDomainService | Class |
| Define.domain.services.AppliedRuleExceptionDomainService | Class |
| Define.domain.SoftwareArchitecture | Class |
| Define.domain.AppliedRule | Class |

### Map logical architecture to software units

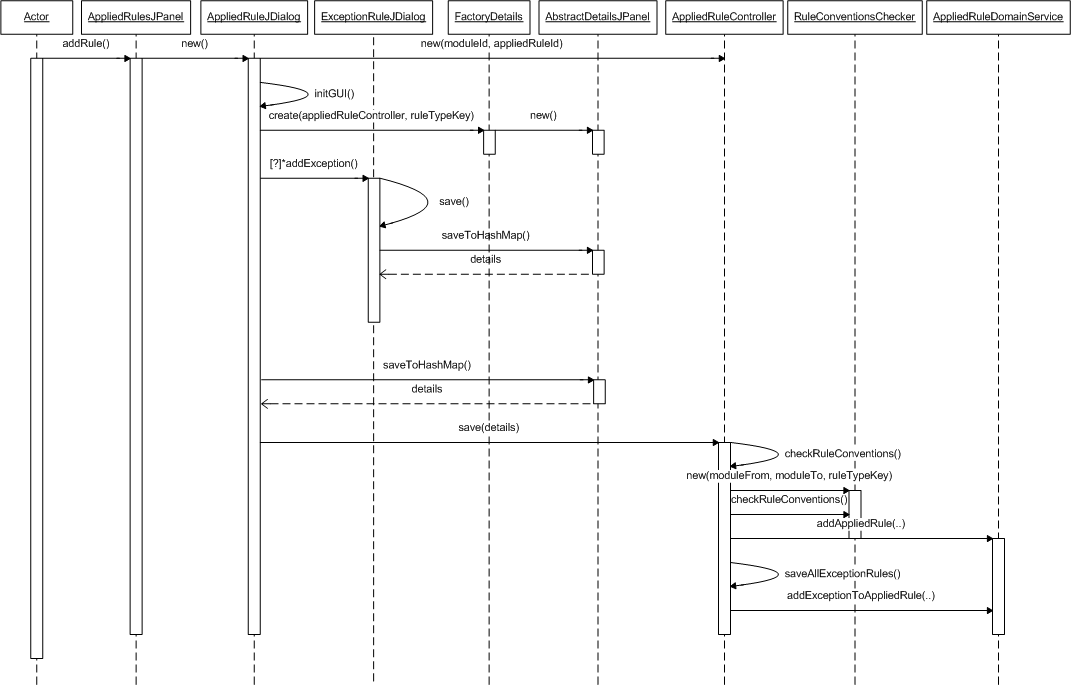
|  |  |
| --- | --- |
| **Path** | **Type** |
| Define.presentation.jpanel. SoftwareUnitsJPanel | Class |
| Define.presentation.jdialog.SoftwareUnitJDialog | Class |
| Define.task.DefinitionController | Class |
| Define.task.SoftwareUnitController | Class |
| Define.domain.services.SoftwareUnitDefinitionDomainService | Class |
| Define.domain.SoftwareArchitecture | Class |
| Define.domain. SoftwareUnitDefinition | Class |

## Sequence diagrams

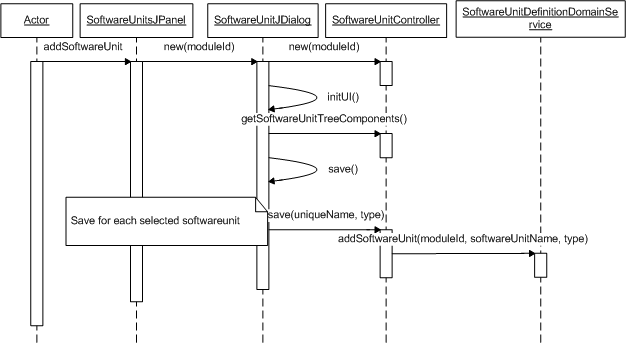
### Define logical architecture



### Define rules



### Map logical architecture to physical architecture



# Decisions and justifications

|  |  |
| --- | --- |
| Decision | Applied rule has a composition with itself |
| Where | Conceptual Domain Model, class AppliedRule |
| Reason | It is now possible to define exception rules on an applied rule making it possible to define different types of AppliedRule exceptions which can store different data depending on the RuleType. |

|  |  |
| --- | --- |
| Decision | Applied rule contains information about the violation types |
| Where | Conceptual Domain Model, class AppliedRule |
| Reason | NFR 3.2 |

|  |  |
| --- | --- |
| Decision | Separate the task from the presentation |
| Where | Package task |
| Reason | NFR 4 (And required by Team 1) |

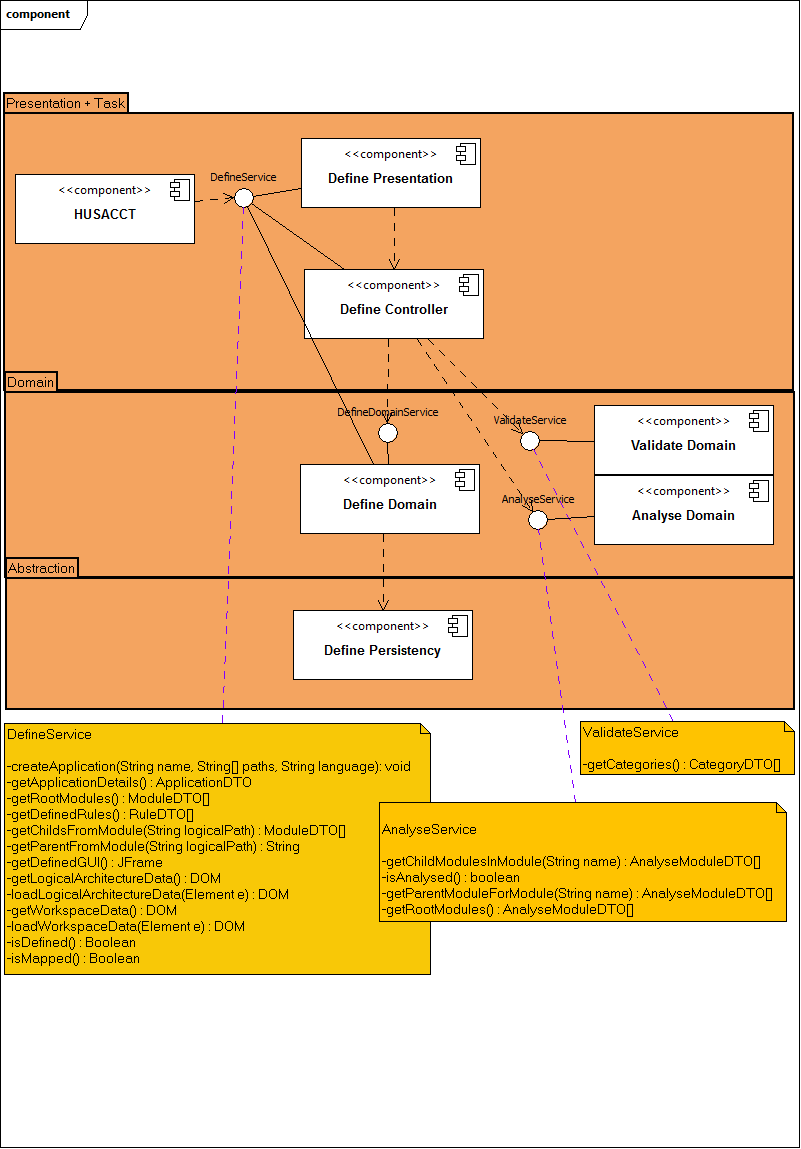
|  |  |
| --- | --- |
| Decision | Using Singleton pattern |
| Where | Package domain |
| Reason | It is better to have only one instance of Architecture Definition, and it must be accessible to clients from a well-known access point. |

|  |  |
| --- | --- |
| Decision | Using Domain Services |
| Where | Domain package |
| Reason | Low coupling between layers. |

|  |  |
| --- | --- |
| Decision | Using Property files for switching language |
| Where | Common package |
| Reason | NFR 1.1, NFR 3.5 (And required by Team 1) |

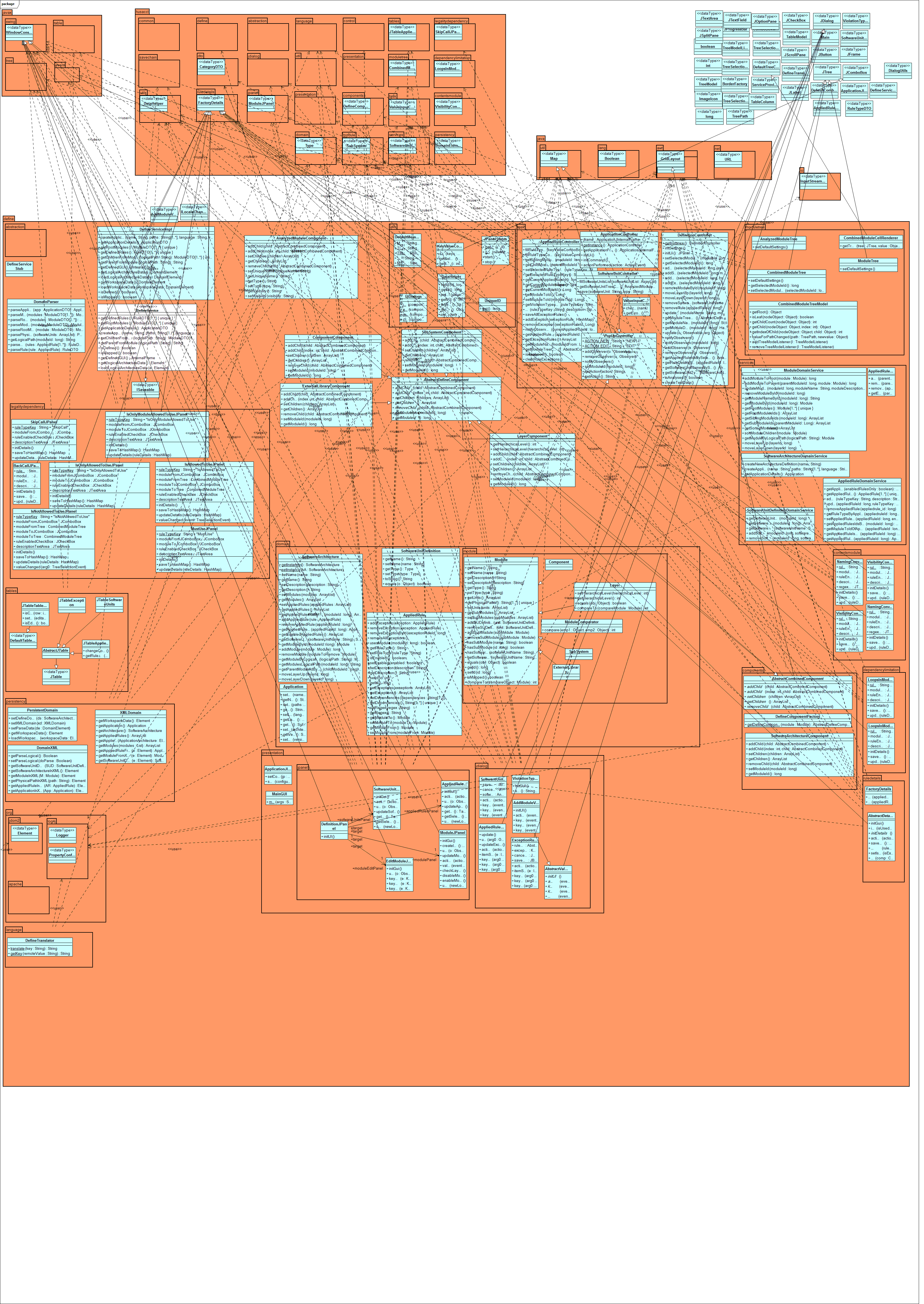
|  |  |
| --- | --- |
| Decision | HUSACCT does not support physical/ghost modules |
| Where | Domain package |
| Reason | We choose not to allow a module to be defined as a ghost module. A ghost module means it is not visible in the GUI and has a sole purpose to combine a rule and a software unit. However a ghost module is unjustifiable in the information model. So we chose to create a logical module instead when a software unit is selected. |

# Software partitioning



# Physical Class Diagram

The physical class diagram shows all the classes and packages used by the define component. It shows links with other packages too. The diagram has been implemented in Topcased 5.2.0.



# Subsystem specification

Some subsystems in the define component are very large and/or hard to understand. For each of this subsystems in the define component we tried to created 1 or more of the following information to get an insight into the subsystem and to explain certain difficulties related to the subsystem.

* Classes
* Sequence diagrams showing important mechanisms
* Other diagrams explaining the subsystem

## Module Tree

This subsystem is very important. It is used 3 times for different purposes. Also the creation of this subsystem is quite difficult to understand if you first see the code. Therefore we did create a sequence diagram to show how the process operates. The module tree itself is placed in the presentation layer. The (view) components used placed in the task layer because the components are filled in the AppliedRuleController which is also placed in the task layer.

There are 3 types of trees. All trees use the same CellRenderer and TreeModel. The TreeModel causes the children of the tree to be displayed in the right manner. The CellRenderer takes care of what has to be displayed. In this tree that will be an icon for each different Component, and its name or path. The 3 tree types are:

* AnalyzedModuleTree

*This shows the path of analyzed software units in a tree.*

* ModuleTree

*Shows all defined modules in a tree.*

* CombineModuleTree

*Combining above-mentioned tree’s*

### Classes

|  |
| --- |
| *hussact.define.presentation.moduletree* |
| AnalyzedModuleTree |
| ModuleTree |
| CombinedModuleTree |
| CombinedModuleCellRenderer |
| CombinedModuleTreeModel |

|  |
| --- |
| *hussact.define.taks.components* |
| AbstractCombinedComponent |
| DefineComponentFactory |
| AbstractDefineComponent |
| ComponentComponent |
| DefineComponentFactory |
| ExternalLibraryComponent |
| LayerComponent |
| SoftwareArchitectureComponent |
| SubSystemComponent |

### Sequence Diagram



## Rule Details

RuleDetails is another very important subsystem. The mechanism is difficult to understand because it has a lot of classes. The UML diagram below shows that it is just a factory pattern.

### Classes

|  |
| --- |
| *hussact.define.presentation.jpanel* |
| AppliedRulesJPanel |
| DefinitionJPanel |
| EditModuleJPanel |
| ModuleJPanel |
| SoftwareUnitsJPanel |
| *PACKAGE - ruledetails* |
| AbstractDetailsJPanel |
| FactoryDetails |
| *PACKAGE - ruledetails.components* |
| AbstractPanelComponent |
| DescriptionPanelComponent |
| EnabledPanelComponent |
| ModuleFromPanelComponent |
| ModuleToPanelComponent |
| RegexPanelComponent |
| *PACKAGE - ruledetails.contentsmodule* |
| InterfaceConventionJPanel |
| NamingConventionExceptionJPanel |
| NamingConventionJPanel |
| SubClassConventionJPanel |
| VisibilityConventionExceptionJPanel |
| VisibilityConventionJPanel |
| *PACKAGE - ruledetails.dependencylimitation* |
| CyclesBetweenModulesExceptionJPanel |
| CyclesBetweenModulesJPanel |
| *PACKAGE - ruledetails.legalitydependency* |
| BackCallJPanel |
| IsAllowedToUseJPanel |
| IsNotAllowedToUseJPanel |
| IsOnlyAllowedToUseJPanel |
| IsOnlyModuleAllowedToUseJPanel |
| MustUseJPanel |
| SkipCallJPanel |

### Class diagram



## Contradictory rule conventions checker

When applying rules on defined modules, it is easily done to define contradictory rules. This subsystem checks the selected from, and sometimes to, module for existing contradictory rules. First, it checks which rule type is selected and second, on the basis of this selection, the correct checks are performed. The following rules are contradictory:

|  |  |
| --- | --- |
| Rule | Forbidden when following rule is defined |
| Naming convention | * “Naming convention” rule in the same module |
| Visibility convention | * “Visibility convention” rule in the same module |
| Subclass convention | * “Subclass convention”: rule in the same module * Same checks as a “must use” rule |
| Interface convention | * “Interface convention” rule in the same module * Same checks as a “must use” rule |
| Is not allowed to use | * “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 | * “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 | * “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 | * “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 | * “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” |
| Skip call | * 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 | * 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. |

### Classes

|  |
| --- |
| *hussact.define.taks.conventions\_checker* |
| LayerCheckerHelper |
| ModuleCheckerHelper |
| RuleConventionsChecker |

### Sequence Diagram



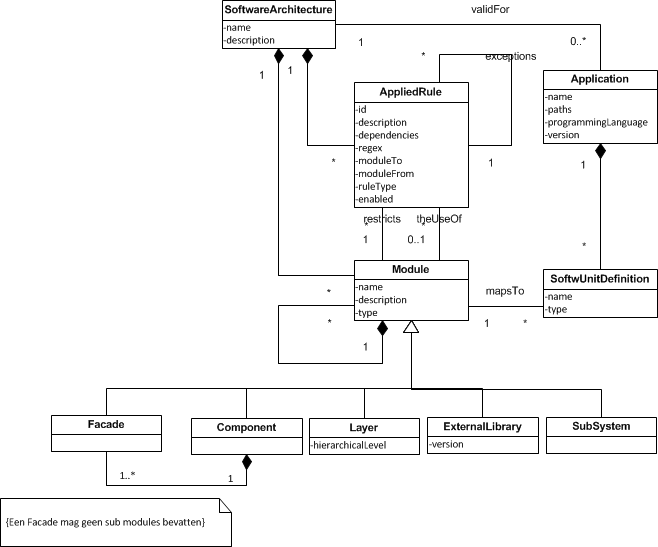
## Domain model

### Classes

|  |
| --- |
| *hussact.define.domain* |
| Application |
| AppliedRule |
| SoftwareArchitecture |
| SoftwareUnitDefinition |

|  |
| --- |
| *hussact.define.domain.module* |
| Component |
| ExternalLibrary |
| Layer |
| Module |
| ModuleComparator |
| SubSystem |

### Information model



## Domain Parser

The domain parser is the part of define that transforms the domain objects to Data Transfer Objects. The DTO’s are placed in a shared package named common, so each component is allowed to use them. This makes it easy to transfer data between components.

In the domain parser you will find has a method to parse all units, which calls a method to parse a single object to its DTO.

### Sequence Diagram



# How-To-Add

## New Rule

To add a new rule a couple of things need to be changed. To start with the basics, the new rule needs to be added in the Validate Component. Once it has been added there it becomes selectable in the dropdown box at the add rule GUI. However this new rule still has no GUI of its own. To do this we need to create a new class in one of the packages in the ruledetails package which is located here: *hussact.define.presentation.jpane.ruledetails*.

### GUI

To create a GUI for the new rule you must make a new class which needs to extend the AbstractDetailsJPanel . Once this is done, you will need to add GUI component. These GUI components, located *hussact.define.presentation.jpane.ruledetails.components,* are simple JPanels which have their own representation and logic to validate their data.

To add these components to the GUI of your new rule you will need to do so as follows:

1. Add a public attribute of the component you wish to add.

**public** **class** TestRuleJPanel **extends** AbstractDetailsJPanel {

**public** EnabledPanelComponent enabledPanelComponent;

1. Initialize the new component in the *InitGUI* method. Modify the GridBagConstraints so it aligns with the rest of your components.

@Override  
**public** **void** initDetails() {  
 enabledPanelComponent = **new** EnabledPanelComponent ();  
 **this**.add(enabledPanelComponent, **new** GridBagConstraints(0, 0, 2, 1, 0.0, 0.0,   
 GridBagConstraints.*FIRST\_LINE\_START*, GridBagConstraints.*NONE*, **new** Insets(0,   
 0, 0, 0), 0, 0));  
**}**

1. Complement the following methods with your added components.

@Override  
**public** **boolean** hasValidData() {  
 **boolean** hasValidData = **true**;  
 hasValidData = hasValidData && enabledPanelComponent.hasValidData(); **return** hasValidData;  
}  
@Override  
**public** **void** updateDetails(HashMap<String, Object> ruleDetails) {  
 **super**.updateDetails(ruleDetails);  
 enabledPanelComponent.update(ruleDetails.get("enabled"));  
}

@Override  
**public** HashMap<String, Object> saveToHashMap() {  
 HashMap<String, Object> ruleDetails = **super**.saveToHashMap();  
   
 ruleDetails.put("enabled", (Boolean) **this**.enabledPanelComponent.getValue());  
 **return** ruleDetails;  
}

Once you followed these steps for all the components you need to define this new rule you have a new GUI for this rule.

### Factory

You are not done yet. Once you added the new GUI you must still add it to the Factory.  
The factory class is located here: husacct.define.presentation.jpanel.ruledetails. FactoryDetails.

To assign the GUI you just made to the new rule you will need to do a couple of things.

First you will need to add a public static attribute to the GUI you just made.  
The value of this attribute should be the same as the ruleTypeKey of your new rule.

**public static final String** *ruleTypeKey* = *”YourRuleTypeKey”*;

Once you have added this attribute you can now modify the FactoryDetails class to return this GUI when this rule is selected. The first thing to do is to add a private attribute of the new GUI.

**private** TestRuleJPanel testRuleJPanel;

After you have done this you need to add the GUI in de create and get method of the factory.  
To do this add the follow piece of code to the *create* method.

**else** **if** (ruleTypeKey.equals(TestRuleJPanel.*ruleTypeKey*)){  
 testRuleJPanel = **new** TestRulelJPanel(appliedRuleController) **return** testRuleJPanel;   
 }

And add this piece of code to the *get* method.

**else** **if** (ruleTypeKey.equals(TestRulelJPanel.*ruleTypeKey*)){  
 **if** (testRuleJPanel == **null**) {  
 testRuleJPanel = **new** TestRulelJPanel (appliedRuleController);  
 }  
 **return** testRuleJPanel;  
}

Congratulations, you have now successfully added a new rule.

# Testing

Testing the application GUI wise in such way that functionality is assured needs to be done by an human hand. This means that all functionality, defined as use cases, needs to be tested as black box tests. The tests are to be confirmed by GUI response.

Tests are not defined for GUI actions and usage. The main reason for this is that the GUI changed during the whole development process and that we’ve tested the GUI throughout the development process. Some of the ideas for improvements that we’ve came up with are listed in the next chapter *Future Work*.

Code, apart from the GUI, has also been tested throughout the development process. This means that the code does have some bugs (see the *Known bug list*). It is however possible to test the define service. These tests are defined in *DefineServiceTests.java*. The tests are written as JUnit tests and can be run individually. Each of the test methods represents a feature in the Define Service.

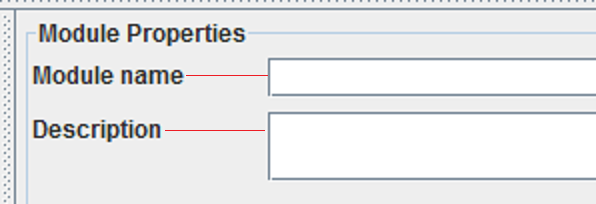
# Future work

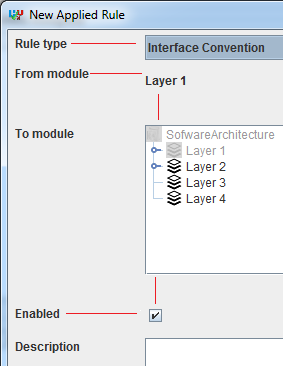
## Known bug list

* The mapped software unit definitions are not checked on multiple logical modules.
* It is impossible to add modules with the same name but it is possible to change a module’s name to a module’s name that is already present.

## Ideas for improvement

* Align GUI
  + A lot of components in the Define GUI are not properly aligned. Especially the module edit panel and the applied rule dialogs. Examples:

1)

2)

* Code quality
  + A lot can be done on code quality, but then again this can always be improved.
* Check mapped architecture
  + This is also a known bug. At this time the architecture is not checked for multiple mappings on the same software unit.

## Ideas for extension

* Implement component module and façade rule
  + At this time, the component module is no different from the Subsystem module. The idea behind a component is that it can only be accessed through a façade. So we could complement this module with automatic creation of a visibility rule. This would mean there is a rule that states this component module cannot be accessed from the outside, with an exception rule on the façade. Note that it should be possible that there are multiple facades.
* Application-Project-Software Unit
  + Currently in the domain there is an Application class which has paths to the location of the application. Furthermore all the software units are falling under one application. It is the intention that the application gets a list of projects. A project is a part of the application that is mapped to one path. This would mean that a project has a name, a path and all the software units falling under this project.
  + This also means that the Software Unit GUI needs to be adjusted to reflex the right changes. The Software Unit GUI now puts everything under one root and this need to be changed to a root for each project.
* Create a right-click menu for the logical modules.
  + Create a right-click menu for the logical modules with the following options.

-Edit module

-Delete

-Rename

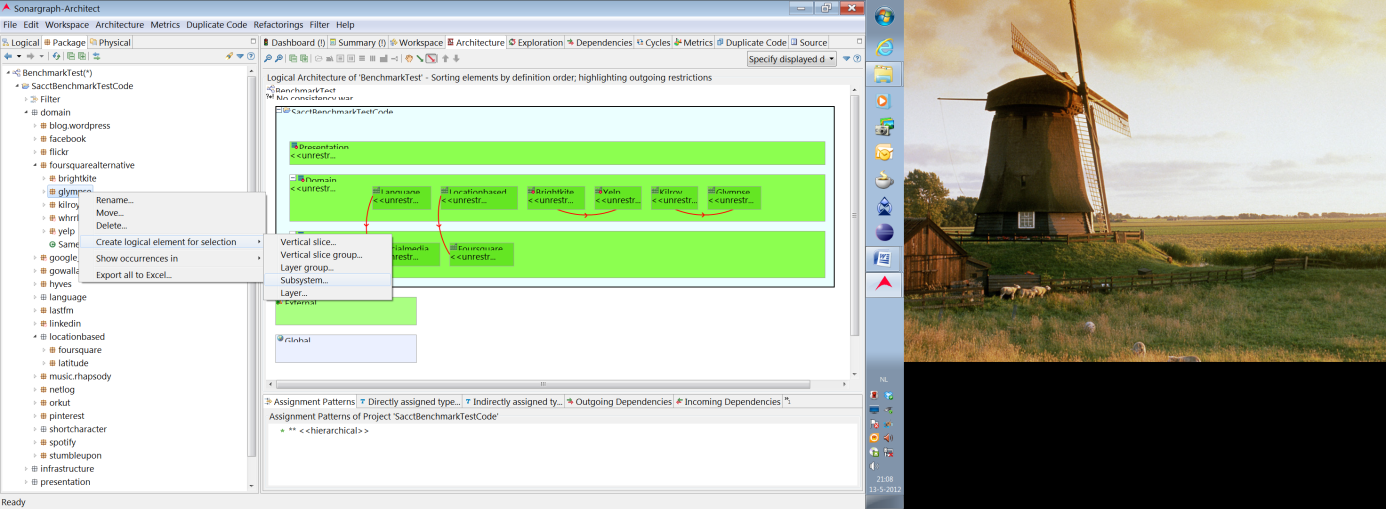
-Move up (if layer)

-Move down (if layer)

-Create logical component based on: -software unit 1

-software unit 2

-software unit 3



* Map software units on module via regular expressions