HU University Of Applied Science
Nijenoord 1
Utrecht, Holland 3500 AD
Phone: **088 481 82 83**
E-Mail: info@hu.nl Web: www.hu.nl

HUSACCT Tool

HU Architecture Compliance Tool

19-04-2012

Functional Design Analyse Component

Team 3
Erik Blanken
Asim Asimijazbutt
Rens Groenveld
Tim Muller

Table of Contents

1. Introduction

The HUSACCT-tool stands for ‘HU Software Architecture Compliance Checking Tool’. The tool has to be designed and developed in order to enable developers and architects to compare the defined architecture with the actual developed code-base.

Based on a research about the existing SACCT-tools, a new SACCT-tool has to be designed, developed and delivered. The new tool must be able to check almost every dependency that can exist in a software-product. The tool should also be designed for new programming-languages to integrate easily.

One important component of a SACCT-tool is the *Analyze-component*. This component enables the tools to change a code-base of an application to data that can be used by the program. This document is the functional design for the *Analyze*-component of the HUSACCT-tool, which includes screen designs, use case models, explanations and more information on the several use cases that this component will inherit.

2. Detailed Use Case Model

In order to get a good overview of the global use cases that the analyse-component of the HUSACCT-tool must provide, a model was made, which gives a good overview of all the use cases. This *Use Case Model* is drawn in figure 1.0.

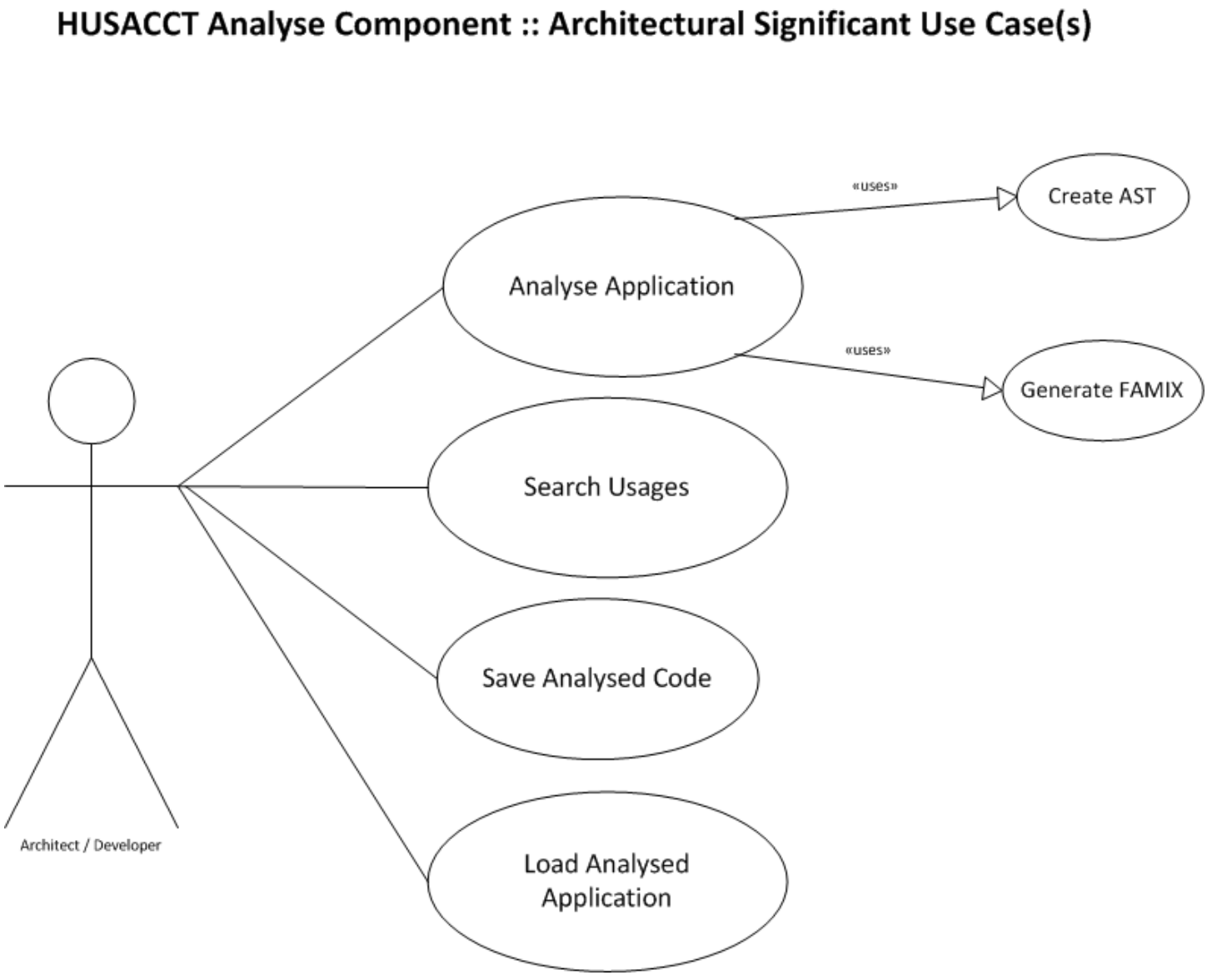

Figure 1.0. Detailed Use Case Model of the HUSACCT Analyze Component

3. User Interface (Representation of functionality)

The Analyze-component of the HUSACCT-tool will work as an information-supplier for other components within the tool. For debugging-purposes, it will be a good feature to be able to provide them with an overview of all contents of the code-base that they have loaded into the HUSACCT-tool. In order to give an overview of the functionalities of the HUSACCT analyse-component, some example screenshots are listed in figures 2.0, 2.1 and 2.2.

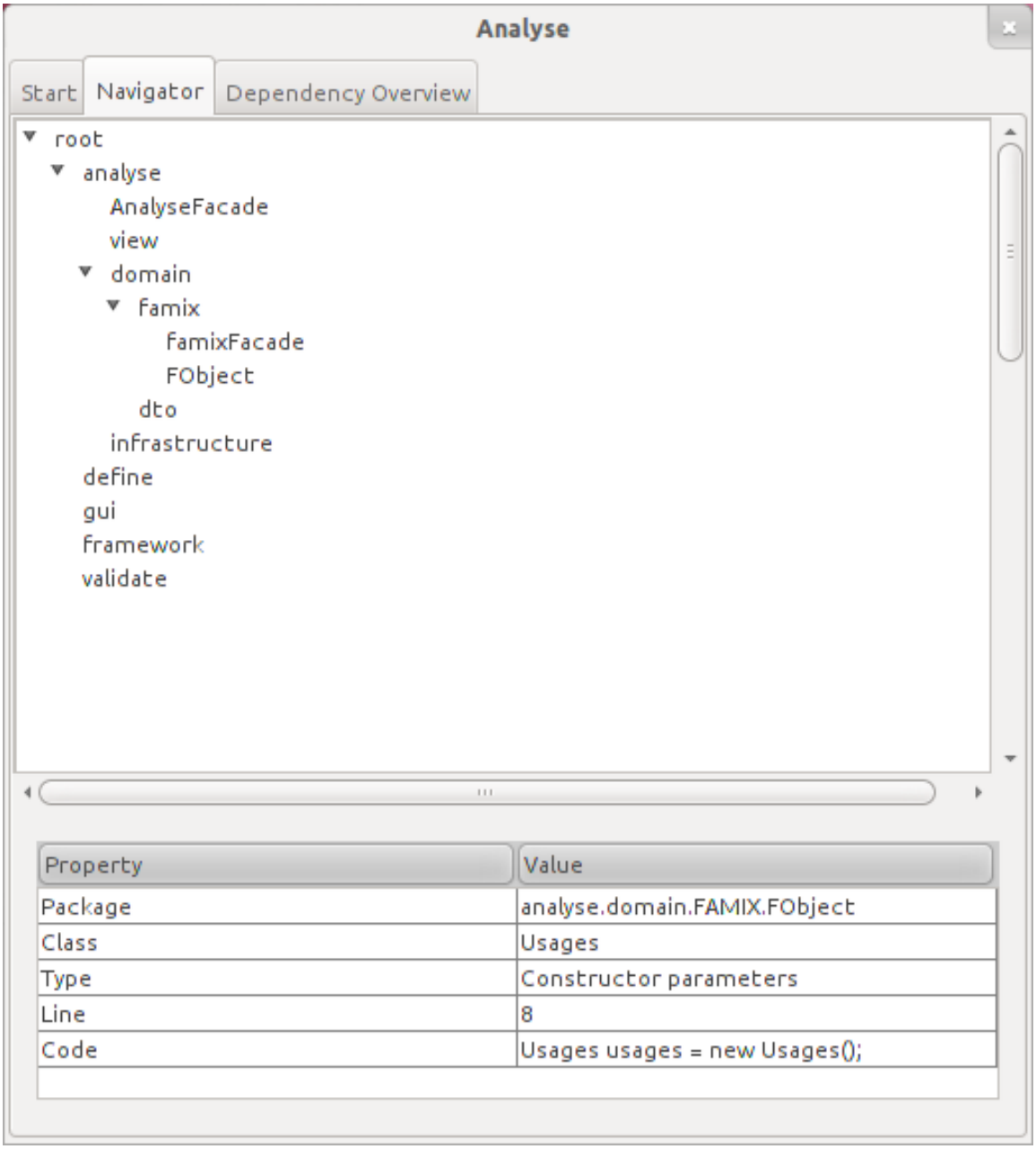

Figure 2.0. HUSACCT Analyse: Codebase Elements

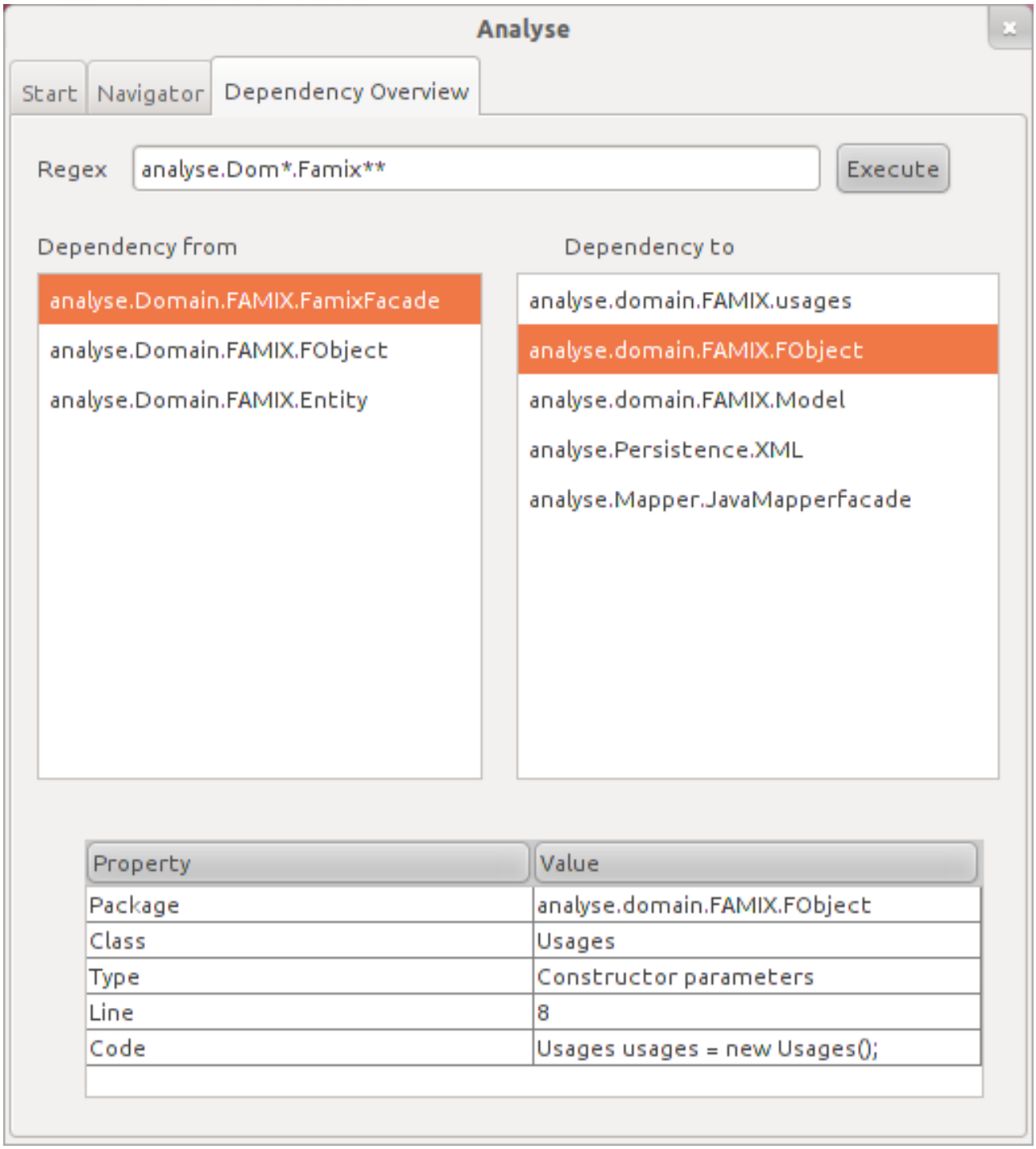

Figure 2.1. HUSACCT Dependency Overview 1

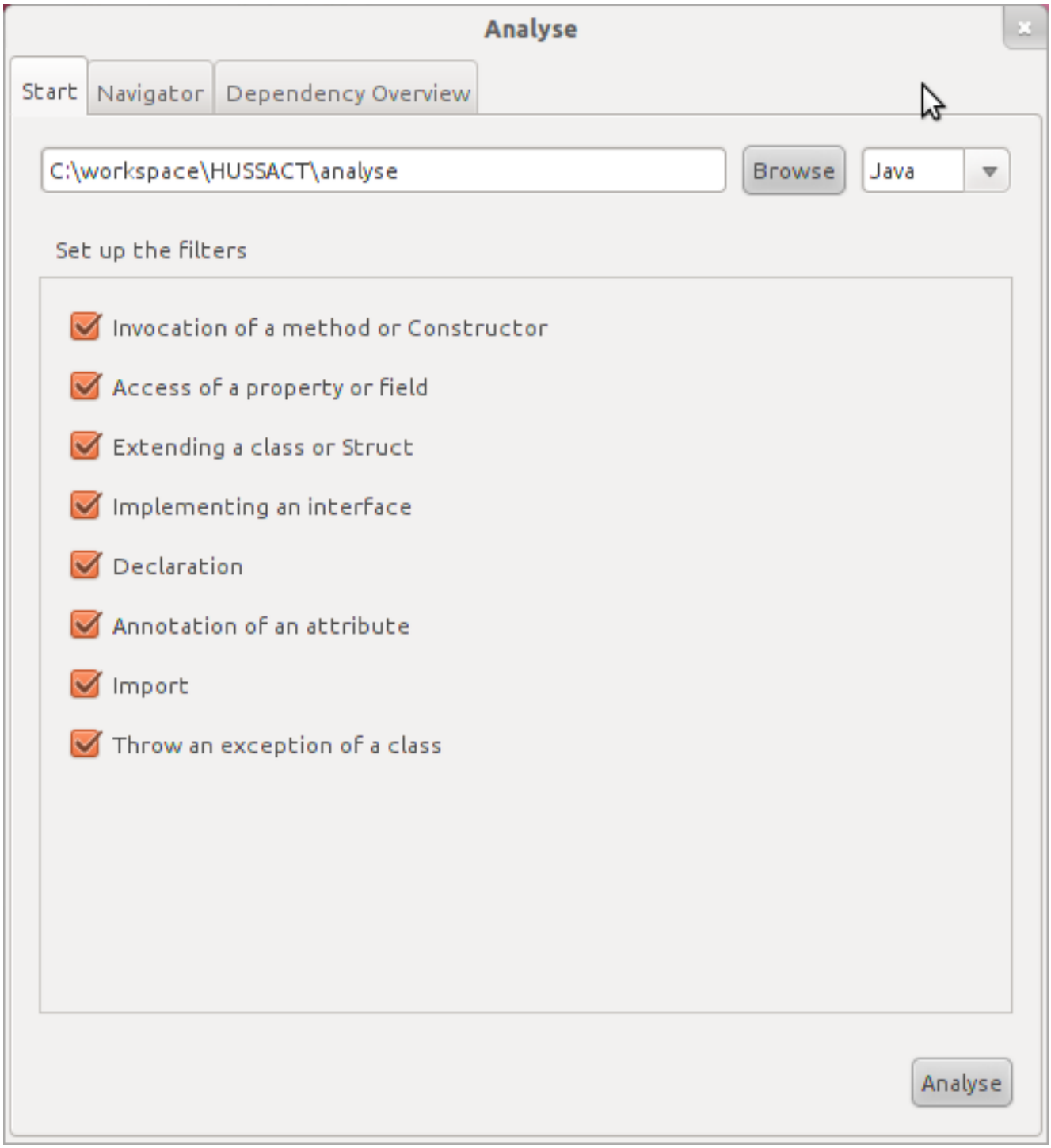

Figure 2.2. Possible configuration screen of dependency types

4. Use Case Specifications

To make the use cases more detailed, textual specification are written for each use case the *Analyze-component* of the HUSACCT-tool provides.

4.1. Analyse Application

Listing 2.1. Textual specification of the use case *analyse application*	
ID	U1
Name	Analyse Application
Actors	End-user
Related Use Cases	Subcase U2 (Create AST) Subcase U3 (Generate FAMIX)
Precondition	End-user submitted the root path of the application in the form of the user interface.
Postcondition	The application is scanned and the AST & FAMIX model are generated.
Description	Analyses the given path by an external tool (U2). This tool provides an "Architecture Syntax Tree" (AST) which contains the structure of the analysed application. The generated tree will be converted to a generic Model (like FAMIX) (U3).

4.2. Create AST (Abstract Syntax Tree)

Listing 2.2. Textual specification of the use case *Create AST (Abstract Syntax Tree)*	
ID	U2
Name	Create AST
Actors	external tool: Antler
Precondition	Provided: the root path of the application
Postcondition	The application is scanned and the AST is generated.
Description	The root path of the application will be given to an external application (Antler). The external application analyses the application and provides a kind of a model. This model must be converted to an "architecture Structure Tree" (AST)

4.3. Generate FAMIX

Listing 2.3. Textual specification of the use case *Generate FAMIX*	
ID	U3
Name	Generate Famix
Actors	The code mapper.
Precondition	Use case "Create AST" is finished and provides the AST
Postcondition	Famix model is created.
Description	By parsing the AST, the Generic Model (Famix) can be filled. The Generic Model contains the structure of the analysed application.

4.4. Search Usages

Listing 2.4. Textual specification of the use case *Search Usages*	
ID	U4
Name	Search Usages
Actors	Validation Service
Precondition	Use case “Analyse Application” (U1) is finished
Postcondition	Usage is found, or no usage is found and an error is returned.
Description	The Validation Service provides a class path or a package path in the application. By searching through the generic model (like Famix) the use case retrieve all packages and classes which have a dependency with the given path. The list with all the dependencies will be returned to the asker in the form of a DTO.

4.5. Save Analyzed Code

Listing 2.5. Textual specification of the use case *Save Analyzed Code*	
ID	U5
Name	Save Analysed Code
Actors	End User
Precondition	Use case “Analyse Application” (U1) is finished
Postcondition	Analyzed code is stored, or an error is returned.
Description	The End User wants to save the result of the analysed code. Through the user interface ‘save’ was chosen and the analysed code will be saved in a file (standard filetype)

4.6. Load Analyzed Application

Listing 2.6. Textual specification of the use case *Load Analyzed Application*	
ID	U6
Name	Load Analysed Application
Actors	End User
Precondition	Use Case “Save Analysed Application” is finished
Postcondition	Analyzed code is loaded from external file.
Description	The end user can choose for ‘load analysed file’ in the user interface. This file is provided by the use case “Save Analysed Application” (U5). By loading this file the Generic Model will be filled with the information of the loaded file. This way the code isn’t analysed and it’s faster to fill the Generic Model.

5. Transformation Rules

No real transformation rules are defined for this project. A few things to keep in mind while developing the HUSACCT analyse-component are listed in listing 3.1.

Listing 3.1. Transformation in the HUSACCT analyse-component

Number	Description
1	Packages and classes will be seen as module. Each module will have a uniqie name. An example of such a unique name: *application.presentation.MainGUI*
2	When the analyse-component is asked for a list of dependencies that a module has, the return-value has to contain a line-number and a type The following types are possible: “abstract”, “class”, “package”, “interace”
3	**String Represenations of violation types** In order to enable filtering dependency types from the analysed project-code, and thus enabling other modules to provide filters when searching dependencies via the analyse-service, some decisions are made about naming the dependency-types in the code as parameter. These names are called the *String representation*. Those decisions are listed in listing 3.1.1.

Listing 3.1.1. String representations of violation types

Nr	String representation	Dependency Type
1	“isNotAllowedToUse”	Is not allowed to use
2	“invocMethod”	Invocation of a method
3	“invocConstructor”	Invocation of a constructor
4	“accessPropertyOrField”	Access of a property or field
5	“extendsConcrete”	Extending a concrete class/struct
6	“extendsAbstract”	Extending an abstract class
7	“implements”	Implemeting an interface
8	“declaration”	Usage of a class/struct/enum as a type
9	“annotation”	Annotation of an attribute
10	“import”	Import Declaration
11	“exception”	Throw an exception of a class

4	**String representation of visibilty** In order to enable filtering visibilities from the analysed project-code, and thus enabling other modules to provide filters when searching dependencies via the analyse-service, some decisions are made about naming the visibilties in the code as parameter. These names are called the *String representation*. Those decisions are listed in listing 3.1.2.

Listing 3.1.2. String representations of visibilities

Nr	String representation	Visibility
1	**“public”**	**Public**
2	**“private”**	**Private**
3	**“protected”**	**Protected**
4	**“default”**	**Default**

6. Constraints & Goals

6.1. Constraints of the *Analyze-component*.

While the product is still in the design-phase, a list of constraints has been set up. These constraints are to be kept in thought during the development process of the HUSACCT-tool. Listing 4.1. shows an overview of all detected constraints or possible constraints for the development of the Analyse-component of the HUSACCT software.

Listing 4.1. Possible Constraints for the *Analyse-component* of the HUSACCT-tool	
Number	Constraint Description
1	The existing tool, developed in 2011, combines multiple components. Seperating these component will deliver extra work.
2	The tool that is used to create an AST (Abstract Syntax Tree) and/or to convert the AST to a FAMIX model, does not implement or detect all ruletypes.

6.2. The goal of the *Analize-component*

The HUSACCT analyze-component has one goal, which is *'Convert code-bases to objects, which enables other components to access the members of that code-base'*. When taking some other requirements under the loop, some other smaller goals can be defined. Listing 4.2. gives an overview of those goals.

Listing 4.2. Goals of the HUSACCT Analyse-component	
Number	Constraint Description
1 (Main Goal)	Convert code-bases to objects, which enables other components to access the members of that code-base
2	Enable the HUSACCT-tool to check the conformance for multiple programming-languages.
3	Enable users to save and load in their analyzed applications.

7. Conceptual Domain Model

To enable the system to work correctly, some changes were made to the conceptual domain-model. The domain model can be seen in figure 3.0.

HUSACCT Conceptual Domain Model (Analyse Component)

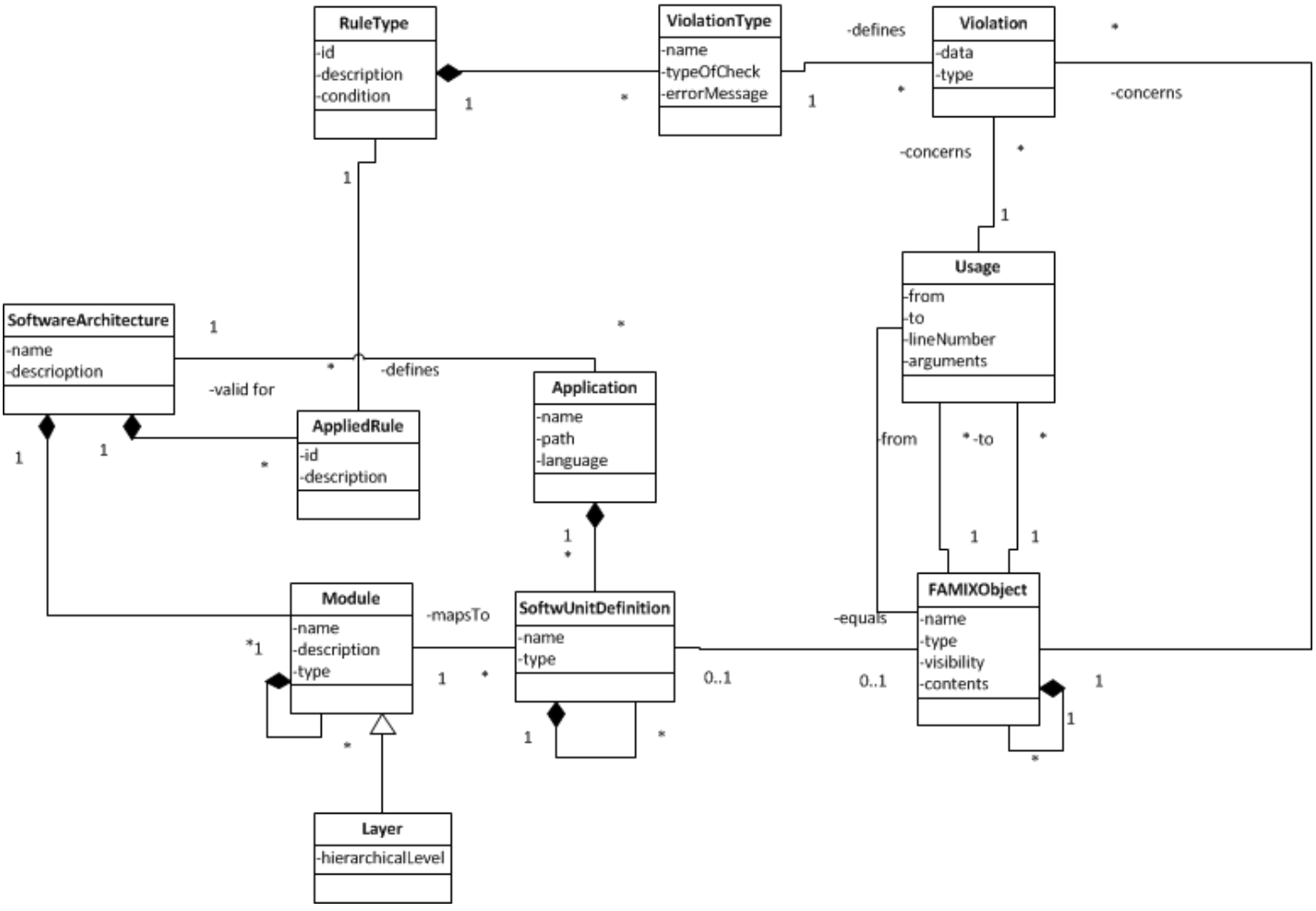

Figure 3.0. HUSACCT Conceptual Domain. Note that the FAMIX-object was created. Appendix 1 explains the setup of the FAMIX-domain. Also note that the Usage has now got the attributes from, to, lineNumber and arguments. These are the data that will be available for a usage.

Appendix 1. FAMIX Class Diagram