A thin vertical grey line is positioned to the left of the text.

HUSACCT SYSTEM

DOCUMENTATION

ANALYSE JAVA & C-SHARP

CONTENTS

CONTENTS.....	1
1 Introduction Analyse Component.....	3
1.1 FUNCTIONAL REQUIREMENTS	3
1.2 NON-FUNCTIONAL REQUIREMENTS	4
1.3 Architectural DECISIONS & JUSTIFICATIONS.....	5
2 Software Architecture.....	6
2.1 CONTEXT DIAGRAM	6
2.2 TOP LEVEL PACKAGES & API	7
2.3 Software Partitioning	8
2.3.1 LAYERS.....	8
2.3.2 Analyse COMPONENT PARTITIONING	9
2.4 Responsibility Trace Table	11
2.5 LAYERS & CLASSES.....	12
2.6 organization Source code.....	13
2.7 Famix model.....	14
3 USE CASE DESCRIPTIONS.....	15
3.1 Analyse Application.....	15
3.1.1 From GUI-event to AnalyseServiceImp	16
3.1.2 From AnalyseServiceImp to ApplicationAnalyser	16
3.1.3 From ApplicationAnalyser to JavaAnalyser.....	17
3.1.4 From JavaAnalyser to Domain (Famix model)	18
3.1.5 Connect Structural Dependencies	19
3.1.6 Recognizing Dependency Types and Sub Types.....	20
3.2 Search Usages	22
3.3 Save analysed application	23
3.4 Load Analysed Application	24
4 Testing the analyse component.....	25
4.1 TESTING DEPENDENCIES & MODULE-FINDERS.....	26
4.2 TESTING LANGUAGE-SPECIFIC ANALYSERS.....	27
5 Adding support for new programming languages	28

5.1	CREATE A NEW ANALYSER	28
5.2	MAKE YOUR ANALYSER AVAILABLE FOR THE APPLICATION	28
5.3	START CREATING YOUR ANALYSER-FUNCTIONALITY!	29
5.4	CREATING JUNIT TESTS FOR YOUR NEW ANALYSER.....	29
6	HUSACCT Famix Implementation & Description.....	30
6.1	Introduction	30
6.2	Workflow.....	30
6.3	Class Descriptions	31

Note: Currently, this documentation is not completely up-to-date (functionality has been improved and extended and some refactoring has been applied), but this document is still providing a good overview of the Analyse component.

Date	Author	Contents
2015-04	Leo Pruijt	Added: Recognizing Dependency Types and Sub Types
2014-01	Leo Pruijt	Analyse application updated and extended Document structure and some text improved
2013-06	Alex Xia, Gerard Bosma	Some additions of work in 2013
2012-06	Erik Blanken, Asim Asimijazbutt, Rens Groenveld, Tim Muller	First version

The following functional requirements apply:

Analyse Functional Requirements	
F #	Requirement
F1	Analysing java-code into a domain model
F2	Analysing c#-code into a domain model
F3	Find dependencies between modules
F4	All dependencies with the following types of dependencies should be reported
	1 Invocation of a method
	2 Invocation of constructor
	3 Extending an abstract class/struct
	4 Extending a concrete class/struct
	5 Extending an interface
	6 Implementing an interface
	7 Declaration of a type
	8 Annotation of a type
	9 Throw an exception of a type
	10 Imports of a type.

1.2 NON-FUNCTIONAL REQUIREMENTS

The following non-functional requirements are relevant for the analyse-component.

Table 2.1. Analyse Non-functional Requirements		
NF #	ISO 9123 attr.	Requirement
NF1	Analysability Testability	Taking over the development of the tool by other development teams must be unproblematic.
NF2	Changeability	The tool must be easily extendable to other code languages.
NF3	Time Behaviour	Tools must not take long (≤ 15 min; 1.000.000 LOC) to analyse/validate the code, and to generate an error report.
NF4	Maturity	The tool must not go down in case of a failure, but generate a meaningful error message.
NF5	Fault Tolerance	There must be no restrictions in the size of the project regarding number of classes, lines of code, components...

1.3 ARCHITECTURAL DECISIONS & JUSTIFICATIONS

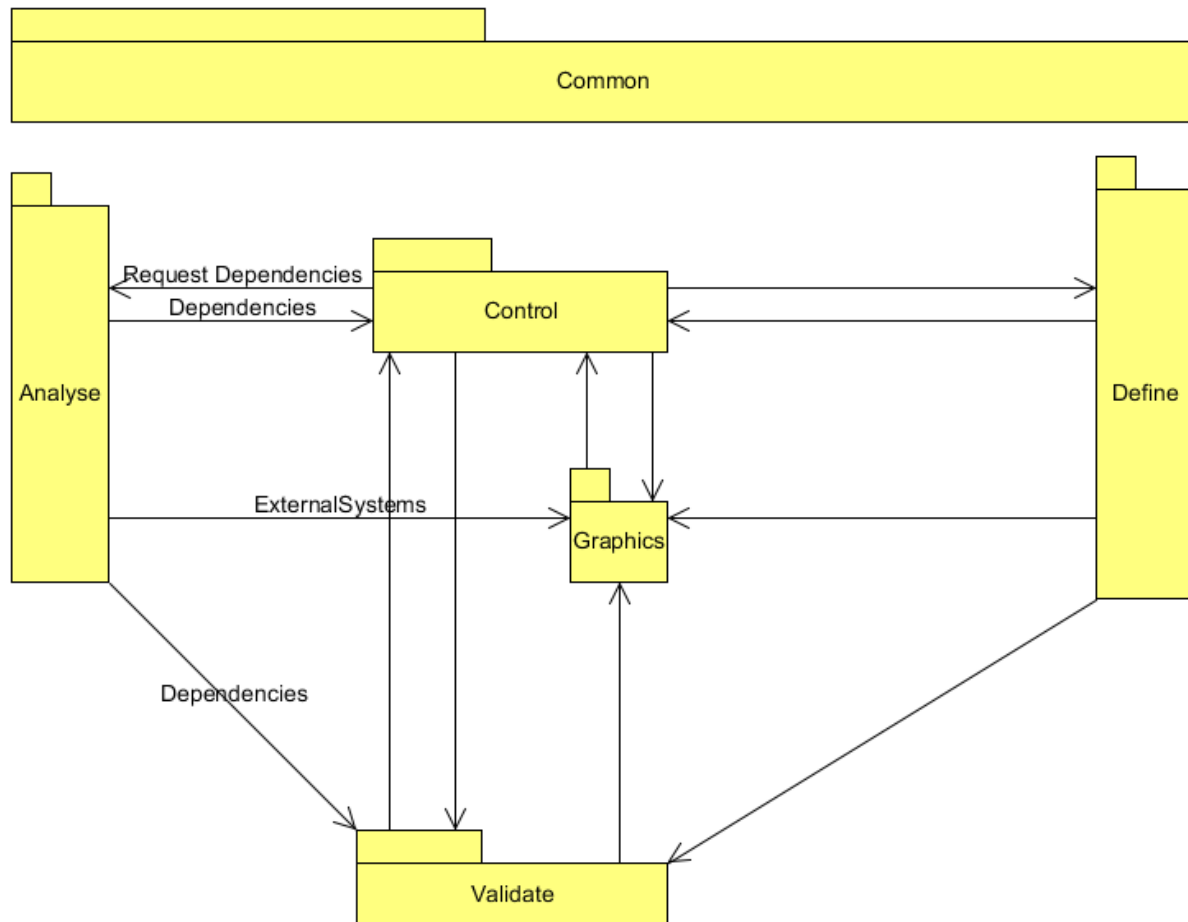
To address the non-functional requirements, some important architectural decisions were made:

Table 2.2. Analyse Decisions & Justification	
Decision	Justification
To enhance the analysability, the analysis process is separated from query processes, and the analysis process is divided into two steps. First, the code base will be converted in an AST (Abstract SyntaxTree). Second, the AST is converted to a code-model. After analysis, the code model can be queried.	NF1.
To enhance the extendibility with other programming languages, Antlr is used for the conversion of program source to Abstract Syntax Tree. Antlr uses so called grammars for this task. Grammars can be written for any programming language, which means that other languages can be implemented in the future. For more information about ANTLR: http://wwwantlr.org/	NF1. NF2.
To enhance the extendibility with other programming languages, Famix will be used to hold the code-model. Famix enables the translation of different programming languages into one complete domain model, which an upper layer can use. If a new programming language is added to the tool, the FAMIX-model can be extended, if needed for types of code constructs not included yet.	NF2.
The language-specific generators can feed the domain model via the IModelCreationService. This allows other language-specific generators to fill the domain via the same service and thus with the same type of parameters and values. Furthermore, it will be easier for developers to replace the FAMIX-model with another model, if needed	NF1 NF2.
To improve performance, string filters will be used for incoming calls, so the validate component can quickly analyse the filter and return the right information. These filters are also used to filter information to certain conditions.	NF 3.

2 SOFTWARE ARCHITECTURE

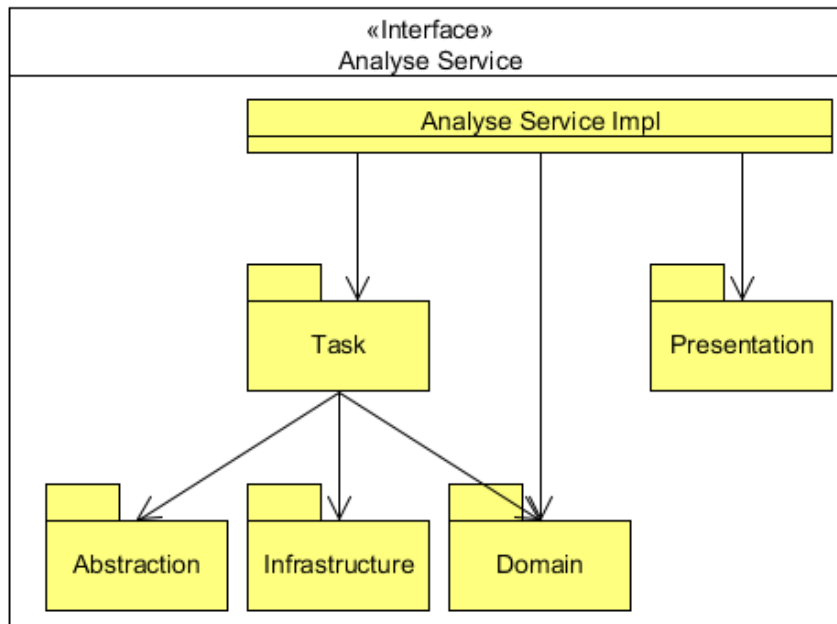
2.1 CONTEXT DIAGRAM

To give a better understanding about the analyse component we need to display it with all the other components. As you can see in the figure below, there are components depending on the analyse component. When requested, analyse can return filtered dependencies and external systems.

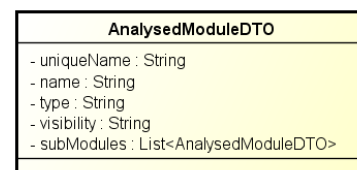
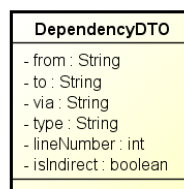
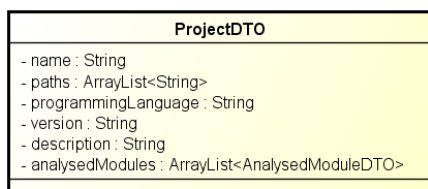
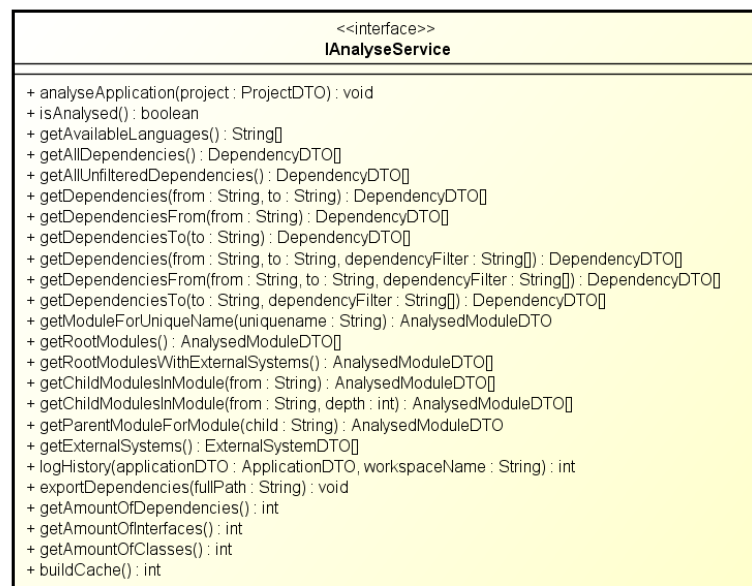


2.2 TOP LEVEL PACKAGES & API

The following diagram shows the packages of the analyse service.



The API represents the implementations of the use cases. Services can be requested via the `IAnalyseService` interface. DTO's are returned when calling services of the API.



2.3 SOFTWARE PARTITIONING

To deliver a maintainable and expandable system, the Analysis component is structured in layers and components.

2.3.1 LAYERS

The software layers and the related dependency rules are specified below.

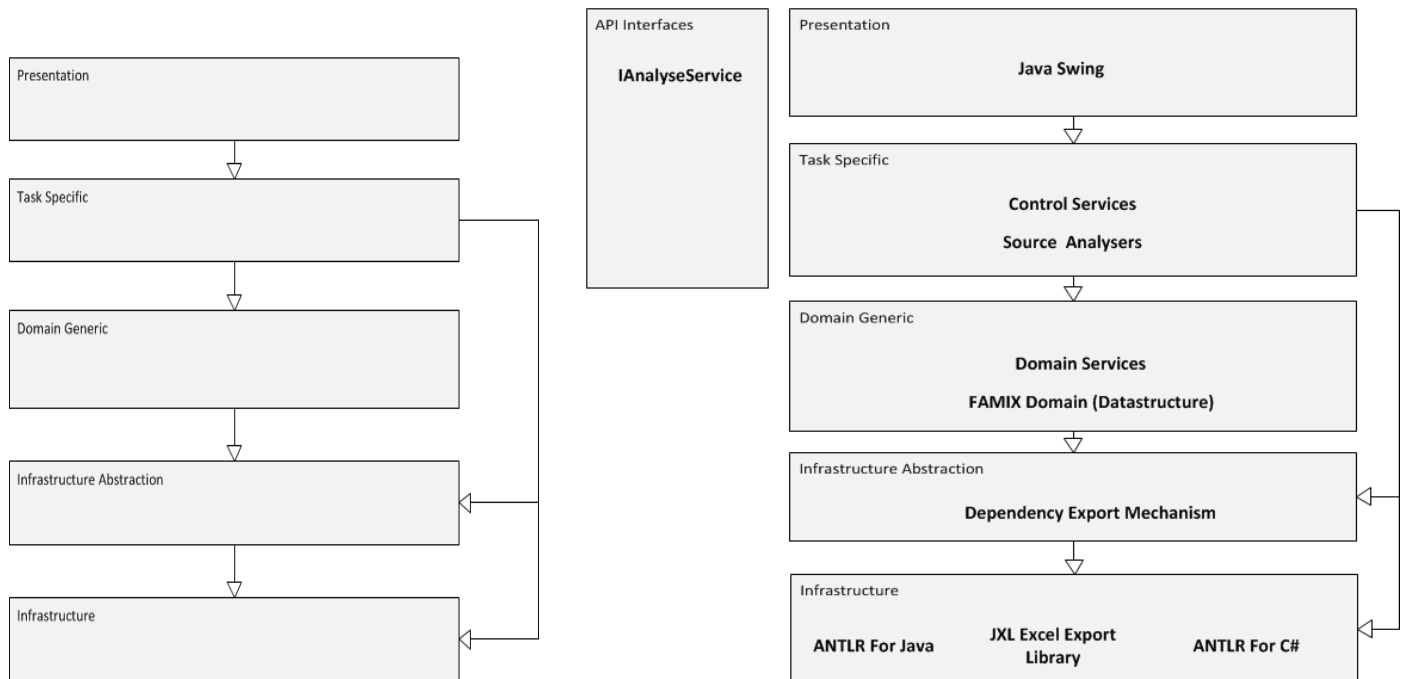
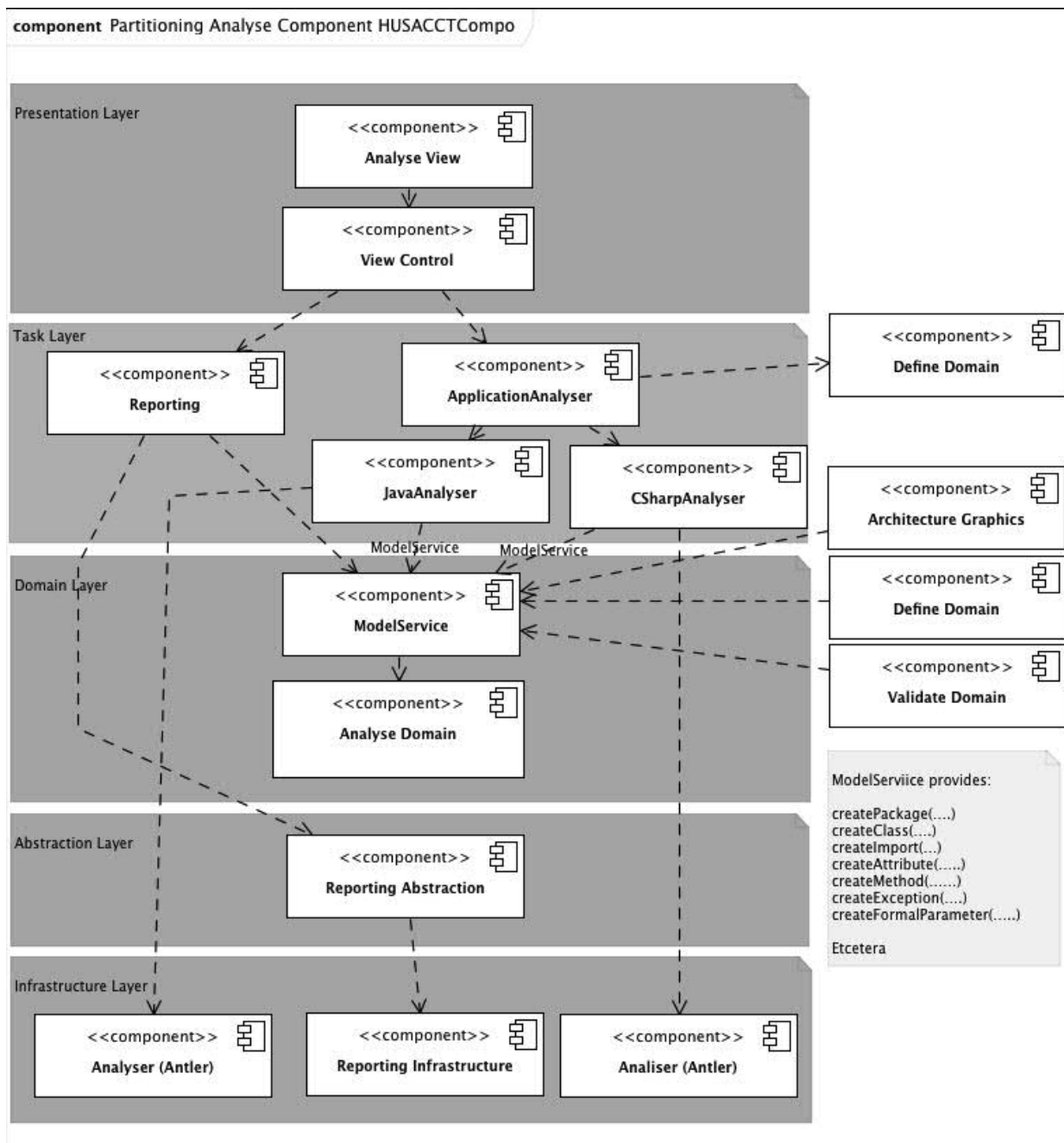


Table 3.1. Architectural Rules of the analyse component

#	Rule
1	The task=specific layer is only allowed to use the domain-layer via IModelCreationService, IModelPersistencyService or IModelQueryService.
2	Task=specific layer can only be accessed via the IAnalyseControlService
3	The task-specific layer is allowed to use the infrastructure layer for external libraries that helps code-translators like the JavaAnalysers to translate code into a specific domain.
4	The task-specific layer is allowed to use the infrastructure abstraction layer, but only to use export mechanisms.

2.3.2 ANALYSE COMPONENT PARTITIONING

In order to follow the layered-models that are created for this component, and to meet the non-functional requirements, the following partitioning has been implemented.



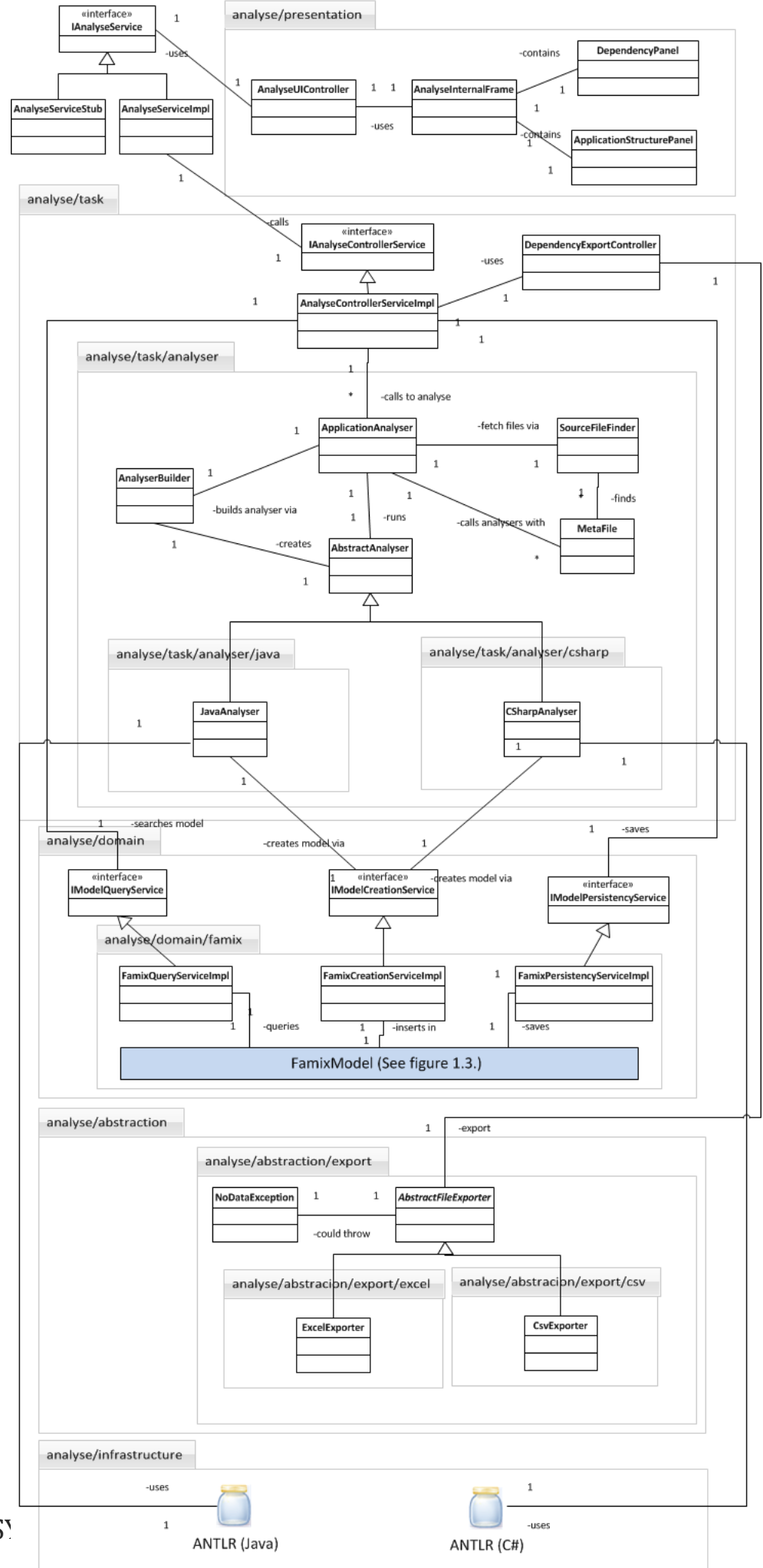
All of the components that can be seen on the previous page are really part of the analyse-component.
To understand to what these component are actually mapped, the following listing will list all mappings.

Husacct Analyse – Software Mapping to physical components		
Analyse	Analyse View	Husacct/analyse/presentation/AnalyseDebuggingFrame.java Husacct/analyse/presentation/AnalyseInternalFrame.java Husacct/analyse/presentation/ApplicationStructurePanel.java Husacct/analyse/presentation/DependencyPanel.java Husacct/analyse/presentation/DependencyTableModel.java Husacct/analyse/presentation/ExportDependenciesDialog.java Husacct/analyse/presentation/FileDialog.java Husacct/analyse/presentation/Regex.java Husacct/analyse/presentation/SoftwareTreeCellRenderere.java Husacct/analyse/presentation/ThreadedDependencyExport.java
	View Control	Husacct/analyse/presentation/AnalyseUIController.java
	Reporting	Husacct/analyse/task/DependencyExportController.java
	Application Analyser	Husacct/analyse/task/analyser/ApplicationAnalyser.java Husacct/analyse/task/analyser/AbstractAnalyser.java Husacct/analyse/task/analyser/AnalyserBuilder.java Husacct/analyse/task/analyser/MetaFile.java Husacct/analyse/task/analyser/SourceFileFinder.java
	Java Analyser	Husacct/analyse/task/analyser/java
	C# Analyser	Husacct/analyse/task/analyser/csharp
	ModelService	Husacct/analyse/domain/IModelCreationService.java Husacct/analyse/domain/IModelPersistencyService.java Husacct/analyse/domain/IModelQueryService.java
	Analyse Domain	Husacct/analyse/domain/famix/*
	Reporting Abstraction	Husacct/analyse/abstraction/storage
	Reporting Infrastructure	JXL-Library
	Analysers(Antler)	Husacct/analyse/infrastructure/anlt/java Husacct/analyse/infrastructure/anlt/csharp Husacct/analyse/infrastructure/anlt/grammars/csharp Husacct/analyse/infrastructure/anlt/grammars/java

2.4 RESPONSIBILITY TRACE TABLE

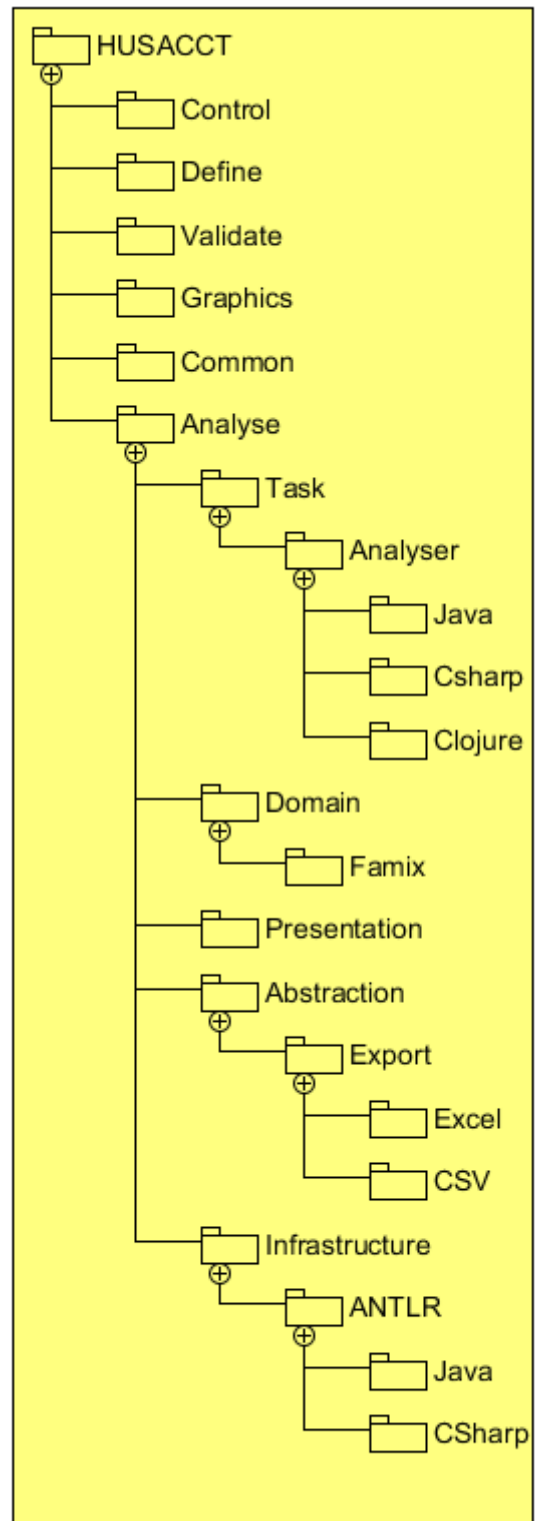
Type of Logic →	Client Interface			Task Specific			Domain Generic			Infrastructure Abstraction		Infrastructure	
Responsibility →	Client Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Application Platform Abstraction	Infrastructure Application Abstraction	Application Platform Service	Infrastructure Application Service
Software Layer / Component ↓													
Analyse View	X												
View Control		X	X										
Reporting						X							
Application Analyser					X								
Java Analyser				X									
C# Analyser				X									
ModelService							X						
Analyse Domain									X				
Reporting Abstraction											X		
Reporting Infrastructure												X	
Analyser(Antlr)													X

2.5 LAYERS & CLASSES



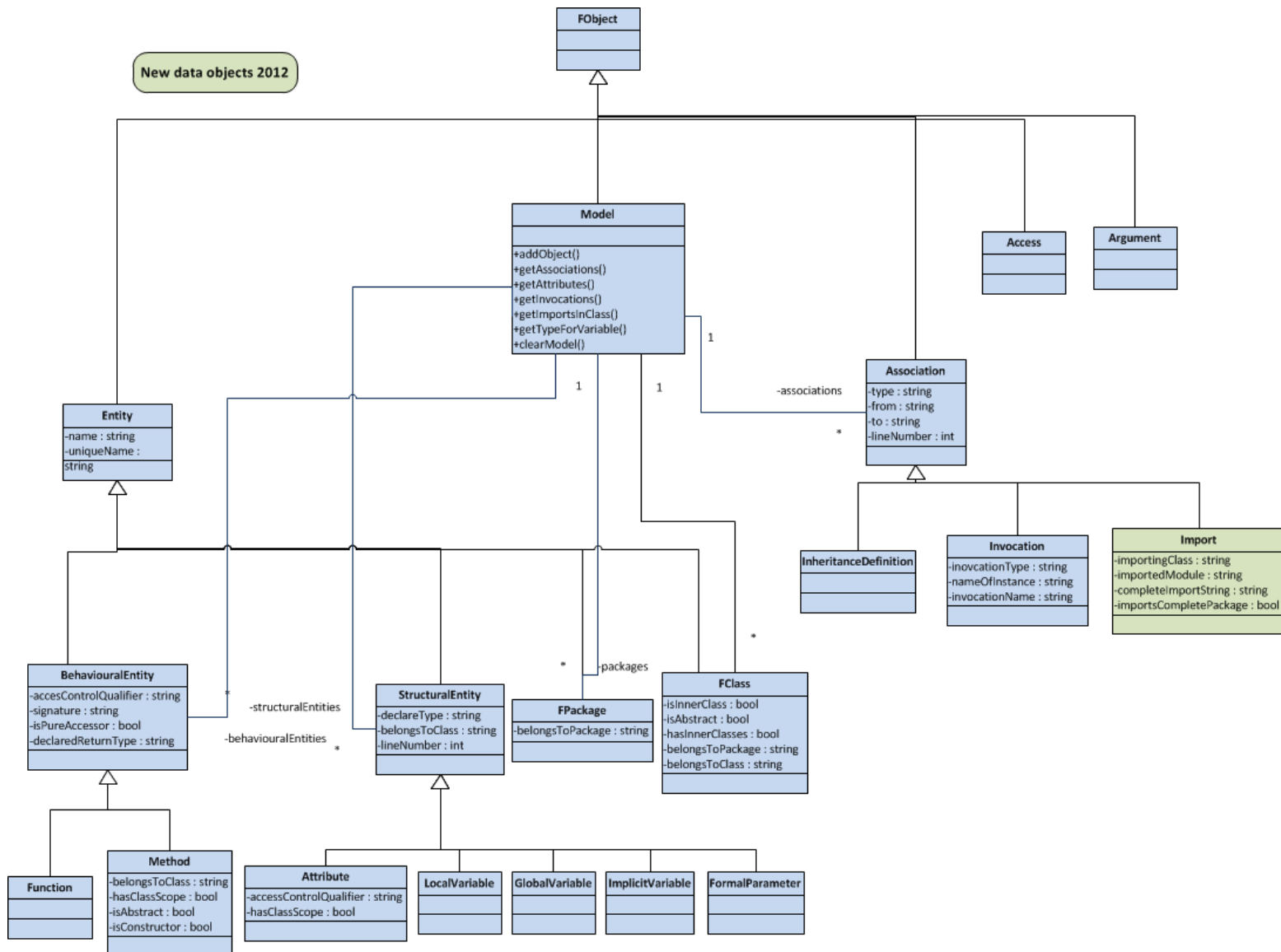
2.6 ORGANIZATION SOURCE CODE

Every important package is listed in this diagram. To improve further functionality for indirect and direct dependencies, see the famix and ANTLR packages.



2.7 FAMIX MODEL

The data of the analysed code is stored in a programming language independent Famix model. More information about the Famix-model or datastructure can be found in appendix 1. Some things about this model are semantically relevant to let the analyse-component work correctly and as expected.



3 USE CASE DESCRIPTIONS

In order to understand the meaning of each use case, this chapter will provide a short motivation for each use case, and provide information on how these use cases are implemented.

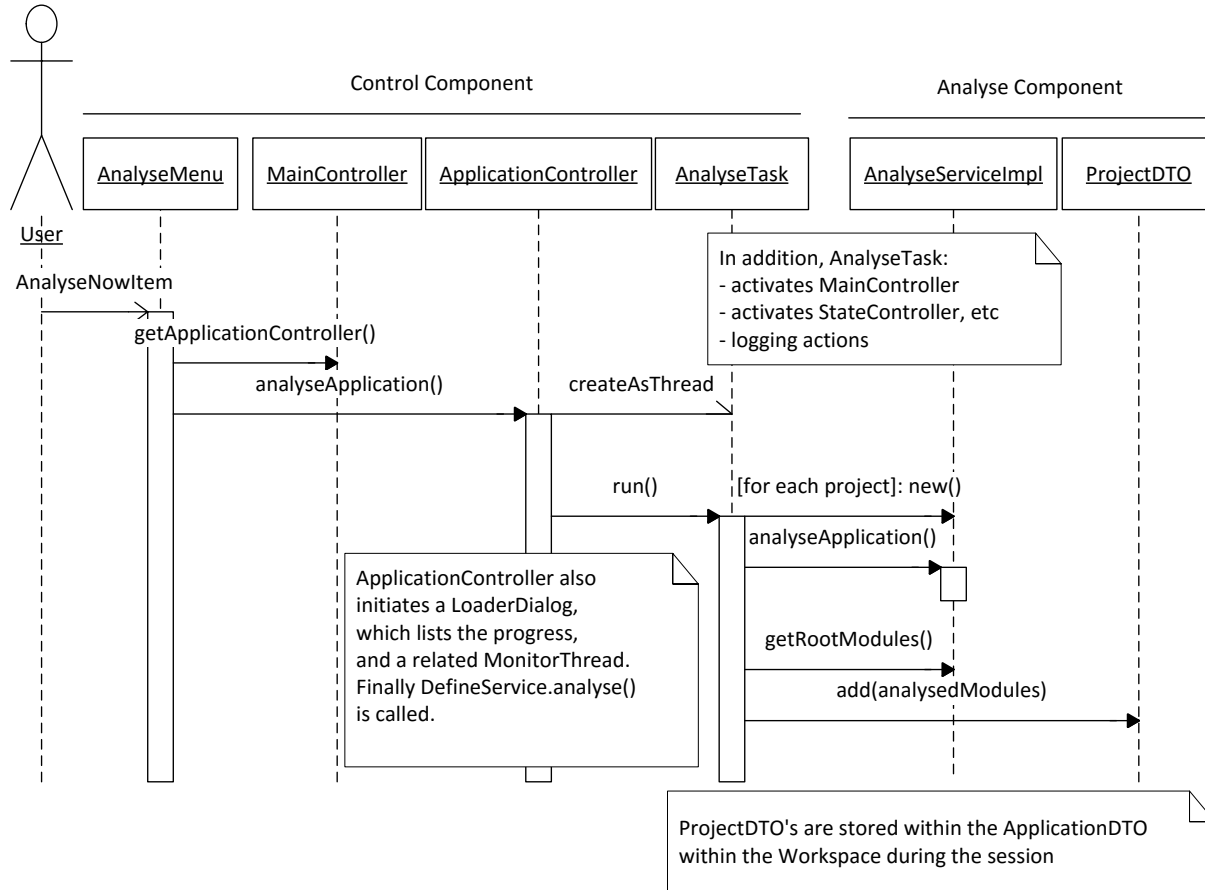
Note: The sequence diagrams do not show all object interaction, only the main line.

3.1 ANALYSE APPLICATION

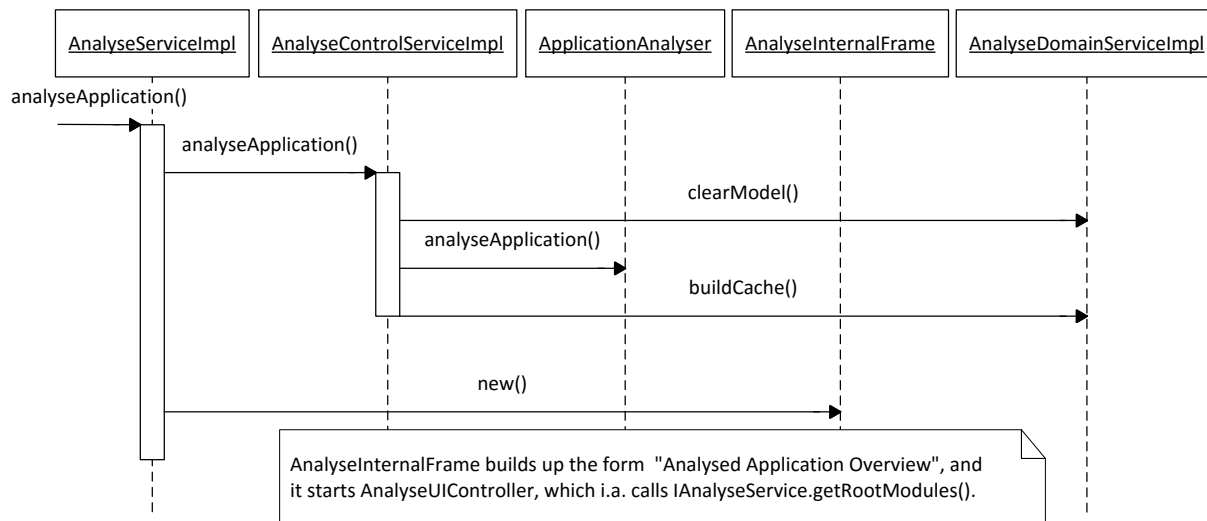
Let's start off with the most important use case: Analyse Application.

Table 1.0. Textual Specification <i>Analyse Application</i>	
Goal	Read source-file from a given directory and turn them to a model that can be used to efficiently search dependencies between different modules.
Implementation Area	<ul style="list-style-type: none">- husacct/analyse/task/analyser/* (Language-specific source-analysers)- husacct/analyse/domain/famix/* (The domain that will be filled)- husacct/analyse/domain/IModelCreationService.java (The API that the analyse-domain provides to fill the model.)- husacct/analyse/domain/famix/FamixCreationServiceImpl.java (Famix-implementation of the IModelCreationService.java)
Extra Info	<p>The source-specific analysers are only allowed to fill the model via the IModelCreationService. The domain-model is encapsulated and by filling it via the IModelCreationService it is ensured that it will work independent from the programming-language in which the source-code is written.</p> <p>The domain-model is wrapped by services similar to the IModelCreationService. This enables developers in the future to create their own implementation of the model, thus they can (for some reason), add or replace the model-implementation of famix by creating a custom domain and implementing those services.</p> <p>This is the only use case that is dependent of another component or situation. The application path must have been set before this function is called, otherwise no source-files will be found. In it's implementation it is dependent on IDefineService.</p>

3.1.1 FROM GUI-EVENT TO ANALYSESERVICEIMP

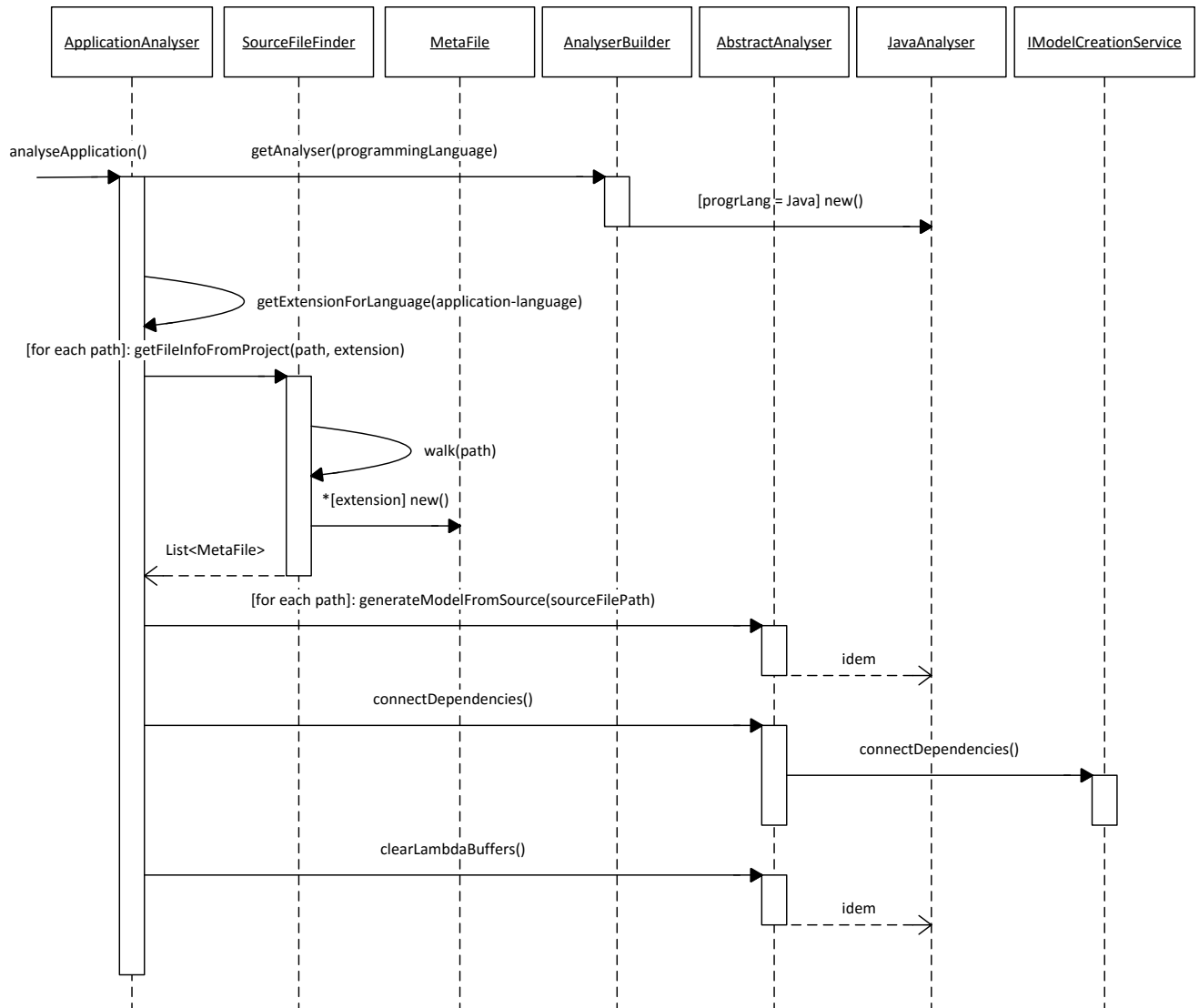


3.1.2 FROM ANALYSESERVICEIMP TO APPLICATIONANALYSER



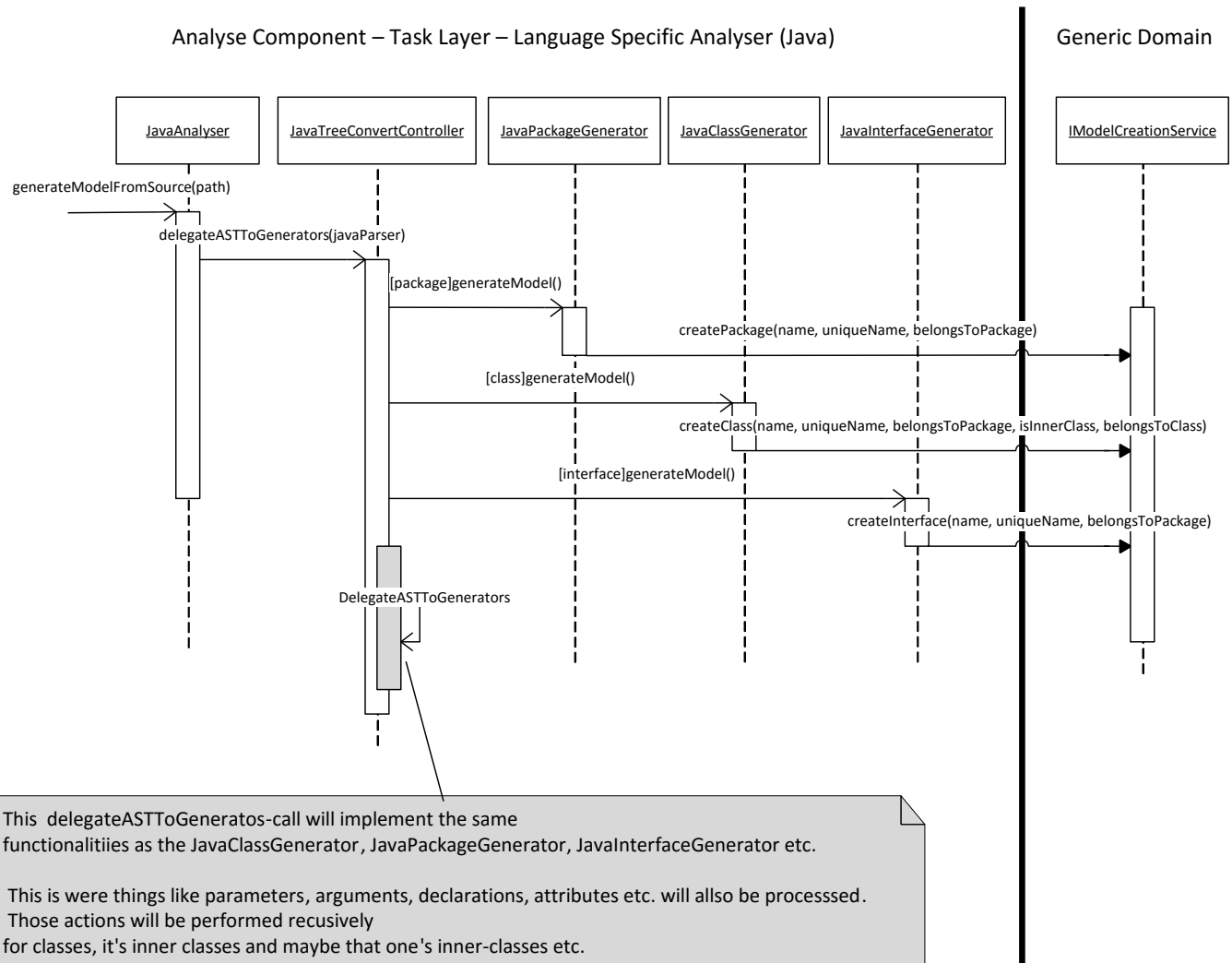
3.1.3 FROM APPLICATIONANALYSER TO JAVAANALYSER

The following figure clarifies how the suiting analyser for a programming Language is created.

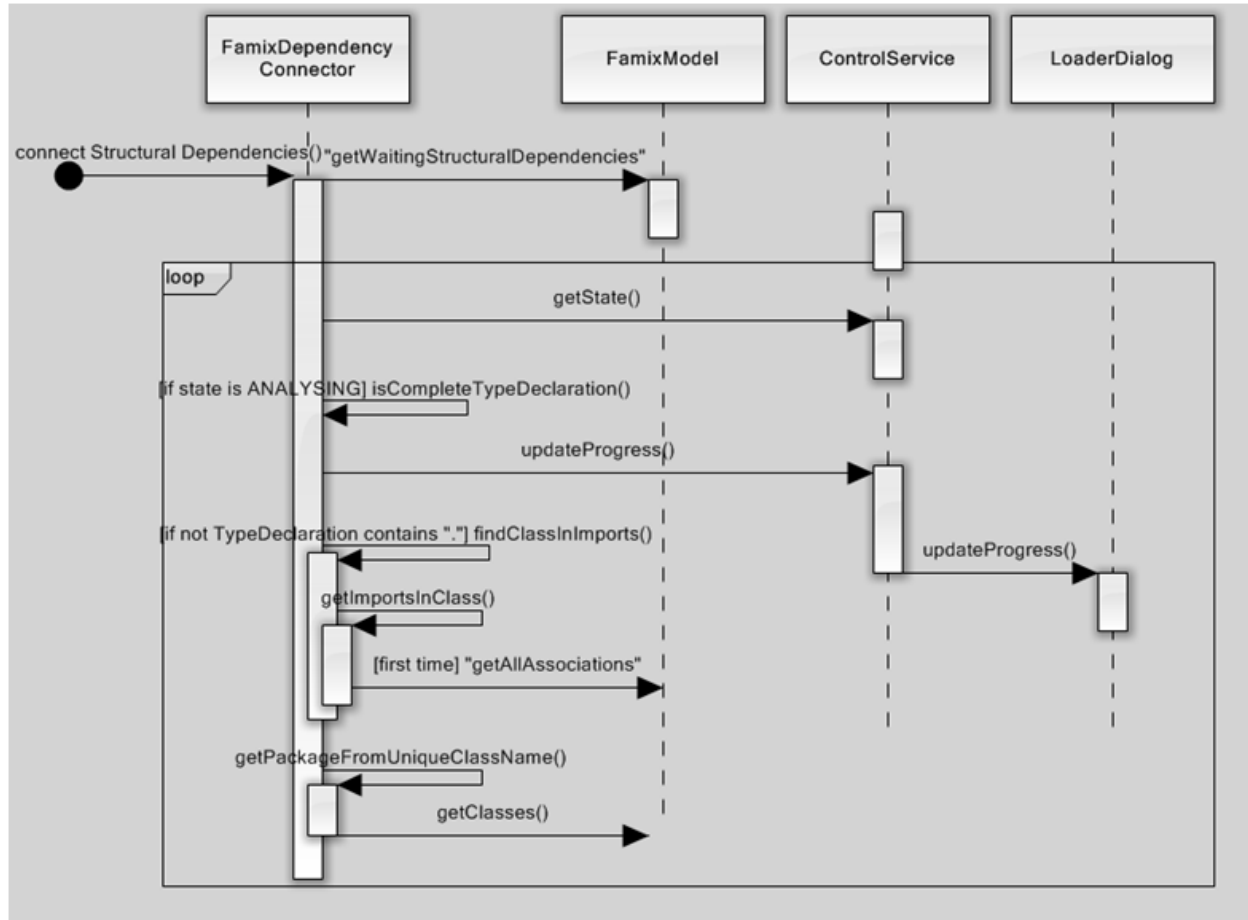


3.1.4 FROM JAVAANALYSER TO DOMAIN (FAMIX MODEL)

To clarify both how a generator can be implemented, and is implemented for java, and how generators can create a generic model via a interface that the domain-model provides, a sequence diagram was drawn up. The following sequence diagram will show how these two things can be successfully implemented.



3.1.5 CONNECT STRUCTURAL DEPENDENCIES



3.1.6 RECOGNIZING DEPENDENCY TYPES AND SUB TYPES

Dependency type represents a category of dependencies.

The following are distinguished: Import, Declaration, Access, Call, Inheritance, Annotation.

SubTypes specify details about the type of usage. Many subtypes are distinguished.

Dependency types and subtypes are determined at different locations in the code.

- Default values are set for dependency type, when an object is created of one of the following Famix objects: FamixImport, FamixException, FamixInheritanceDefinition.
- In case of Import and Annotation, only a type is set, no subtype.
The type is determined in FamixCreationService.
- In case of Declarations, the type and subtype are set in FamixCreationService, based on the code analysis process.
The following subTypes are distinguished: Class variable, Instance Variable, Local Variable, Parameter, Return Type, Type Cast, Exception.
- In case of FamixInheritanceDefinitions:
 - The sub type of direct associations is set in FamixCreationService.
 - Indirect associations of type “inheritance” are added in FamixCreationPostProcessor.indirectAssociations_AddIndirectInheritanceAssociation().
 - The sub types are further determined in FamixCreationPostProcessor.determineDependencySubType().
 - The following subtypes are distinguished: Extends Class, Extends Abstract Class, Implements Class, and Extends Library Class.
- In case of FamixInvocations (used for Call & Access) the specific type and subtype are determined within FamixCreationPostProcessor.processWaitingAssociations() and processWaitingDerivedAssociations(); supported by three methods starting with “determineDependency...”.

3.1.6.1 Access SubTypes

Functional rules:

- 1) Access of attribute owned by the from-class: Usage is not reported as dependency.
However:
 - a) If the attribute is inherited, a dependency to the super class is reported.
 - b) In case that it is the last invocation in an expression usage of the type of the attribute is reported (if the type is found as class or library), as “Reference TypeOfUsedVariable”.
- 2) Access of attribute owned by another class:
 - a) Usage is reported as dependency, while subType specifies the accessed class type or attribute type:
 - i) It is interesting to know the type of the accessed class, so use subtype to report: Library, Interface, or Enumeration Variable.

- ii) it is interesting to know the type of accessed attribute of another class, if the class type is not one of the ones mentioned above, use subtype to report: Class Variable, Instance Variable, combined with Constant or not.
 - iii) For Library, Interface, and Enumeration the type of accessed attribute is not reported, because:
 - (a) In case of Library access the type of attribute is unknown.
 - (b) In case of Enumeration other types are possible.
 - (c) Is it interesting in case of an Interface?
- 3) Access of a Type is reported as with subtype "Reference".
- This is the case in the following situations:
- a) Full type specification, like "domain.direct.Base".
 - b) Name only that refers to an imported class, like "Base".
 - c) Name only that refers to another class in the same package.

3.1.6.2 Call SubTypes

Functional rules:

The same logic applies as described under Access SubTypes, point 1) and 2). The distinguished subtypes are: Enumeration Method, Interface Method, Library Method, Class Method, Instance Method, Local Method.

3.1.6.3 Testing and Notes

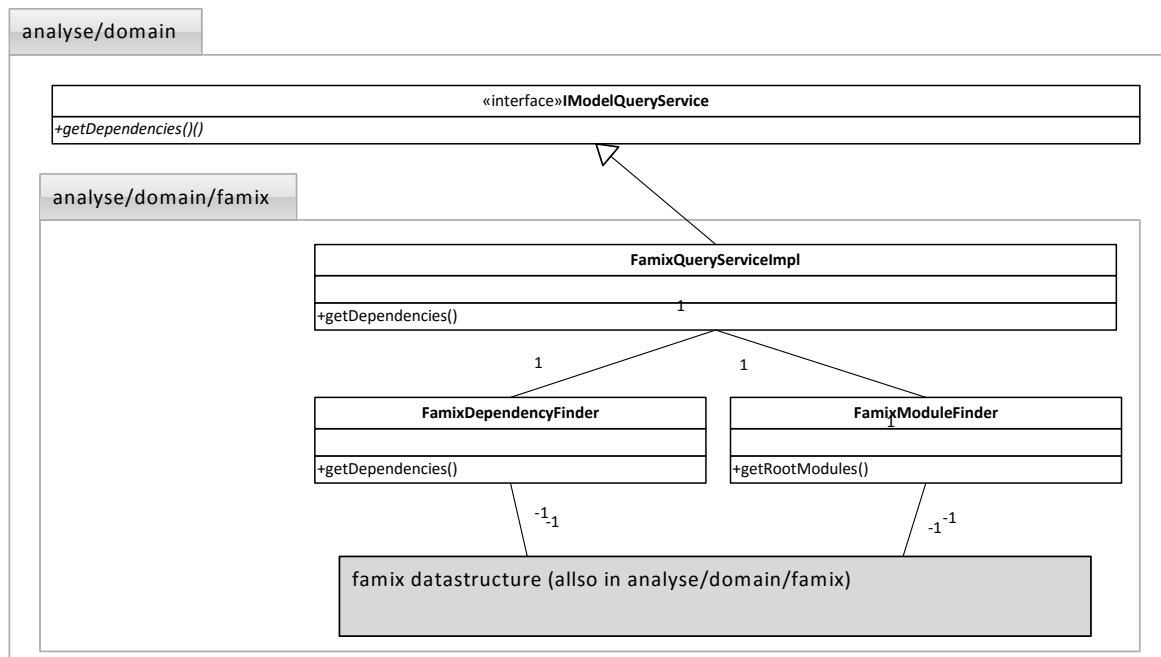
- A) Each subtype, is included in at least two test cases in the accuracy test.
- B) The Freemind test is used to check on the correctness of the dependency types and subtypes.
- C) In addition, it might be usefull to add an extra attribute for the Location?
E.g.: Argument, Return value, Within IF, Within For, ...

3.2 SEARCH USAGES

Another very important use case of this system is searching and returning dependencies between given modules. This chapter gives a short overview of this use case.

Table 1.1. Textual Specification <i>Search Usages</i>	
Goal	Return dependencies, with their types and info, between given modules. Return Modules, at root level or inner modules.
Implementation Area	Available Services: - husacct/analyse/IAnalyseService Important usage implementation area: - husacct/analyse/domain/IModelQueryService - husacct/analyse/domain/famix/FamixQueryServiceImpl - husacct/analyse/domain/famix/FamixDependencyFinder Import module implementation area: - husacct/analyse/domain/IModelQueryService - husacct/analyse/domain/famix/FamixQueryServiceImpl - husacct/analyse/domain/famix/FamixModuleFinder
Extra Info	The code has to be analysed before requesting this use case.

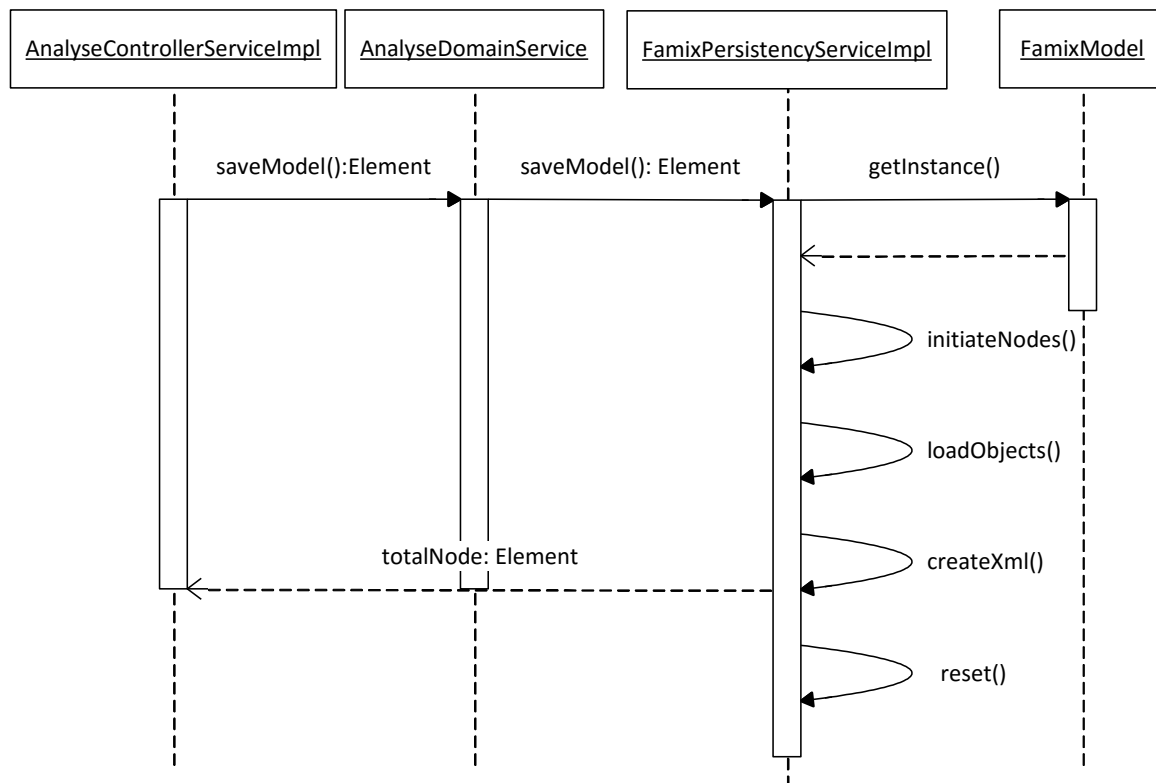
Because of the fact that the implementation of this use case is actually implemented in the famix implementation, and is located in one place, a class model will show how it is implemented.



3.3 SAVE ANALYSED APPLICATION

A functionality to save the analysed domain-model, in this case famix, to an XML-element that can be used in a HUSACCT-workspace, some actions has to been done via a IModelPersistencyService. The global workings are listed in the figure below.

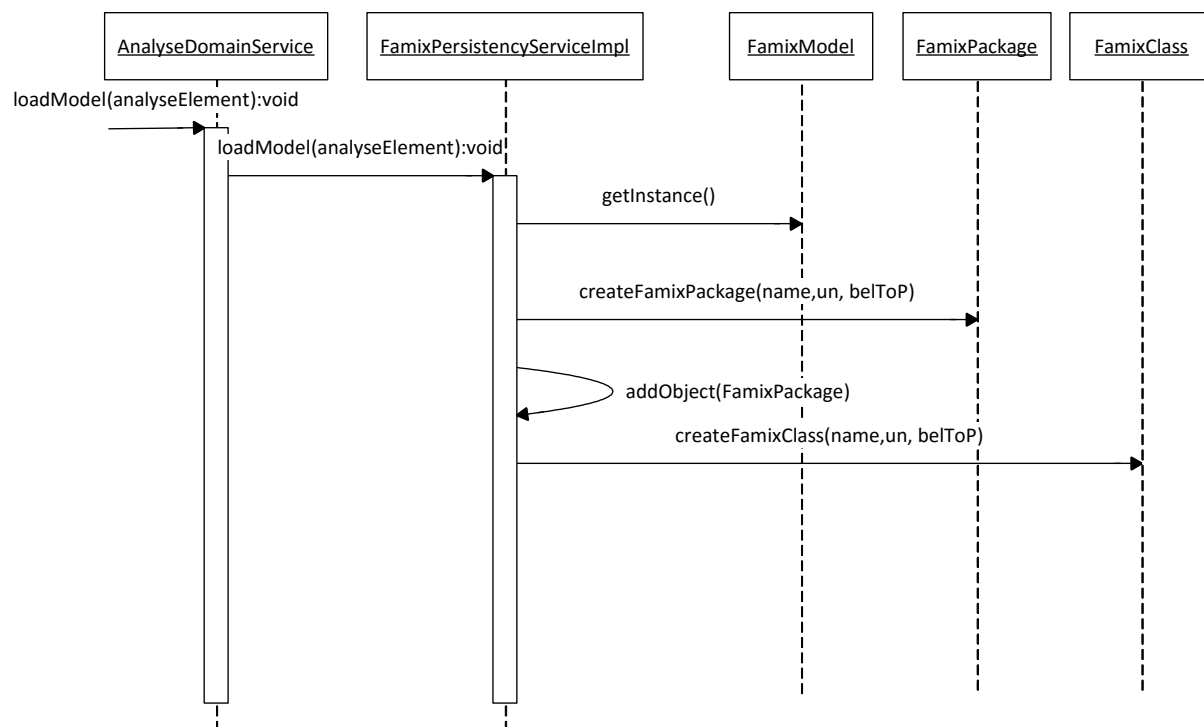
Table 1.2. Textual Specification <i>Search Usages</i>	
Goal	Save the analysed model to an XML-Element.
Implementation Area	Available Services: - husacct/analyse/IAnalyseService Important usage implementation area: - husacct/analyse/domain/IModelPersistencyService - husacct/analyse/domain/famix/FamixPersistencyServiceImpl
Extra Info	This is a service that is called from another component, the Control-component of the HUSACCT.



3.4 LOAD ANALYSED APPLICATION

To load an analysed application back into the model from a given XML-element, this functionality is implemented. To give a global overview of the implementation, a sequence diagram was drawn that shows how the modules and dependencies are created again from an xml element.

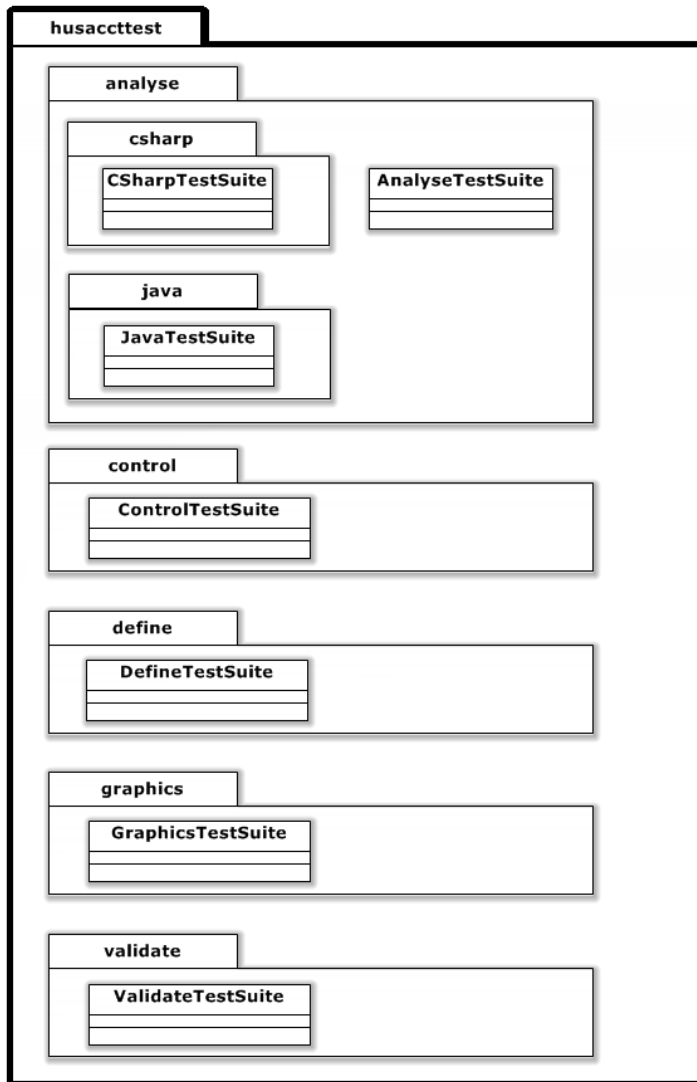
Table 1.3. Textual Specification <i>Search Usages</i>	
Goal	Load an analysed model to an XML-Element.
Implementation Area	Available Services: - husacct/analyse/IA AnalyseService Important usage implementation area: - husacct/analyse/domain/IModelPersistenceService - husacct/analyse/domain/famix/FamixPersistenceServiceImpl
Extra Info	This is a service that is called from another component, the Control-component of the HUSACCT.



4 TESTING THE ANALYSE COMPONENT

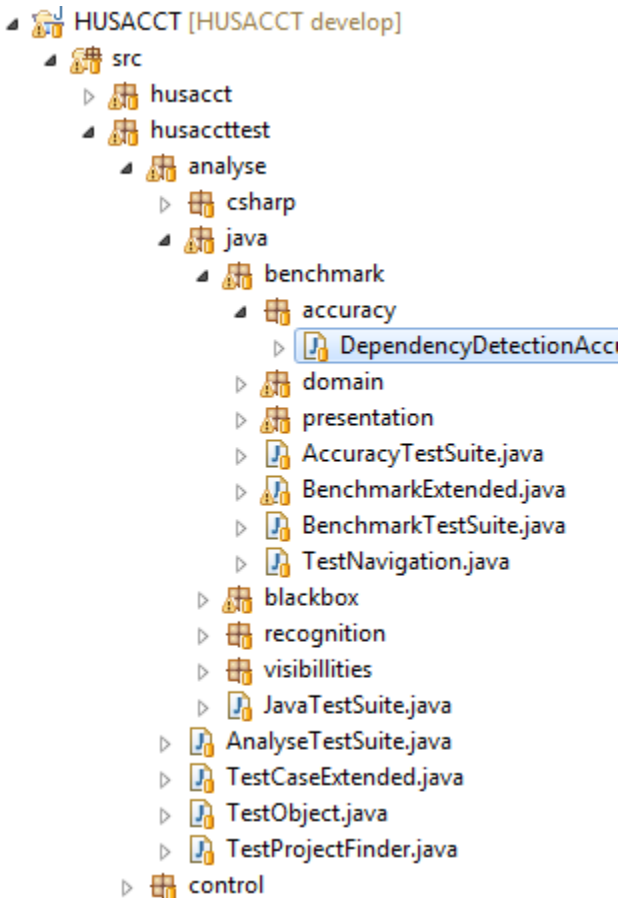
In order to give a short introduction to how to check new analysers and the general part of this component, this chapters explains a few things about the tests.

All tests are included in package husaccttest, subpackage Analyse, etc
To run a certain test, select and run a class as JUnitTest.



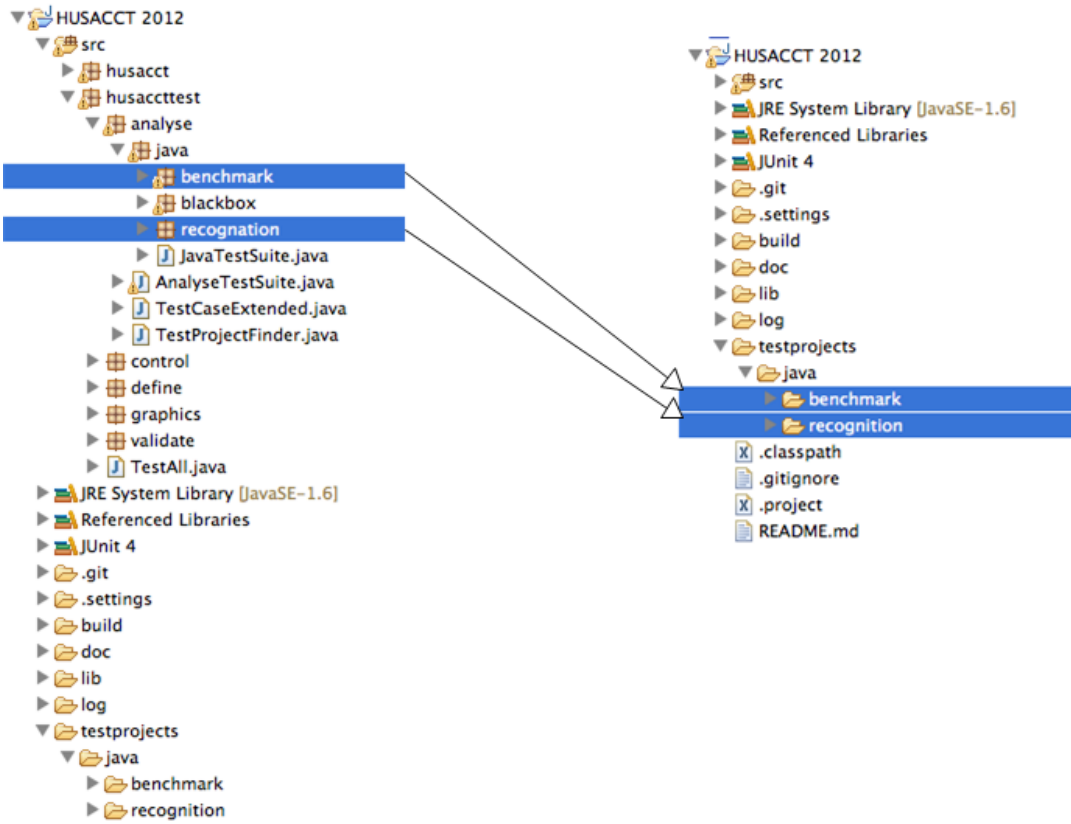
4.1 TESTING DEPENDENCIES & MODULE-FINDERS

The dependencies and modules are tested in `husacttest/analyse/benchmark/accuracy` and `husacttest/analyse/blackbox`. These are general tests for checking main functionalities of the analyse component. To test every specific direct and indirect dependency you will need to open the “`DependencyDetectionAccuracyTest.java`” shown in the figure below.



4.2 TESTING LANGUAGE-SPECIFIC ANALYSERS

In order to test language-specific analysers, an application-structure was made in the root folder of the husacct-project. These test-applications are made to test all different types of declarations in code and see if those are correctly generated from code to the model. The following figure will show which tests applies to which test-applications.



5 ADDING SUPPORT FOR NEW PROGRAMMING LANGUAGES

In order to add support for new Object Oriented programming-languages, some steps have to be followed. This chapter explains those steps.

5.1 CREATE A NEW ANALYSER

- Create a new package in the husacct/analyse/task/analysers and place your analyser in that class, for example husacct/analyse/task/analysers/php.
- Create a new class in the new package, that extends AbstractAnalyser and implement the required functions. (if it's not directly obvious how to do this, check out the other analysers)

5.2 MAKE YOUR ANALYSER AVAILABLE FOR THE APPLICATION

In order to make your analyser available for the HUSACCT-application, after having implemented the previous steps, you have to add some code to the class husacct/analyse/task/analysers/AnalyserBuilder.java.

```
class AnalyserBuilder{  
  
    public AbstractAnalyser getAnalyser(String language){  
        AbstractAnalyser applicationAnalyser;  
        if(language.equals(new JavaAnalyser().getProgrammingLanguage())){  
            applicationAnalyser = new JavaAnalyser();  
        }  
        else if(language.equals(new CSharpAnalyser().getProgrammingLanguage())){  
            applicationAnalyser = new CSharpAnalyser();  
        }  
        else{  
            applicationAnalyser = null;  
        }  
        return applicationAnalyser;  
    }  
}
```



```
class AnalyserBuilder{  
  
    public AbstractAnalyser getAnalyser(String language){  
        AbstractAnalyser applicationAnalyser;  
        if(language.equals(new JavaAnalyser().getProgrammingLanguage())){  
            applicationAnalyser = new JavaAnalyser();  
        }  
        else if(language.equals(new CSharpAnalyser().getProgrammingLanguage())){  
            applicationAnalyser = new CSharpAnalyser();  
        }  
        else if(language.equals(new PHPAnalyser().getProgrammingLanguage())){  
            applicationAnalyser = new PHPAnalyser();  
        }  
        else{  
            applicationAnalyser = null;  
        }  
        return applicationAnalyser;  
    }  
}
```

5.3 START CREATING YOUR ANALYSER-FUNCTIONALITY!

After you have implemented the previous two steps, you can now start developing your new tests. The following table consists of some rules, hints and tips for implementing a new analyser.

Table 4.1. Rules, Tips & Hints for new analysers		
#	Type	..
1	Rule	The model may only be filled via the IModelCreationService!
2	Rule	Let all classes implement an abstract class, just like in the java-generators, which contains a reference to the IModelCreationService. This is a good implementation because of maintainability, expandability and replaceability-reasons.
4	Hint	Carefully test your generators after each step using JUnit tests.
5	Hint	Read the appendix about the FAMIX-model in HUSACCT before starting the development of the new analyser. This document explains each parameter and the semantics of parameters.
6	Tip	If you don't know something, just checkout one of the existing analysers to see how they have implemented their functionality!

5.4 CREATING JUNIT TESTS FOR YOUR NEW ANALYSER

Last but not least some information about how to create new JUnit tests for the new analyser and some rules and important know-hows. Where to put your test-project, is already listed in chapter 4.2. (see the example image).

Table 5.1. Rules, Tips & Hints JUnit-tests for your new analyser		
#	Type	..
1	Rule	Due to build-issues the path to your application has to be set via the a function in the TestProjectFinder.java. Example: <code>String path = TestProjectFinder.lookupProject("java", "recognition");</code>
2	Tip	If you don't know something, just checkout one of the existing analysers to see how they have implemented their functionality!

6 HUSACCT FAMIX IMPLEMENTATION & DESCRIPTION

6.1 INTRODUCTION

The Famix model is a domain that takes care of holding all analysed code information in an organized order, stored in objects. This is made in such a way that this is language independent. There is already Famix documentation, but because the team members have altered this model a bit to suit their needs, this document will serve as a specific guide for the Husacct Tool.

This document provides the workflow of the Famix Model as well as all the classes and it's attributes. Examples will be given with code, but these will be purely Java based.

Remember that a full UML diagram is given at the end of this document. It is very useful to use this as a reference point while you go through this document if you want to fully understand the HusacctFamix Model.

6.2 WORKFLOW

The FamixModel class is basically the center of the domain. Every object that is analysed and put into the domain will go through the addObject() method in the Model class. The model also contains a list of all the attributes and associations, so the queryservice can ask all it's 'get' questions to this Model. It is that the Model is so important, that the decision was made to make this a Singleton. This doesn't have any negative consequences as you can only analyse one application at a time.

There are two kinds of dependencies. Real invocations which belong to the Associations, and there are the declarations which belong to the StructuralEntity types. The StrucuralEntities are purely used for inner workings and should not be seen as a direct dependency. Dependencies are always represented by the Associations. That is the place where the real dependencies are stored and recieved from, once the analysis is over. Here is an example :

```
User testuser = new User() ;
```

The first green part is the declaration of testuser being a User. This will be stored as an Attribute which extends the StructuralEntity. Then the second red part is the actual invocation, in this case a constructor invocation. This will be stored as an Invocation which extends Association. More on this later. For now it is important to know that the Famix Model holds these 2 sorts of dependencies : Associations and StructuralEntities.

6.3 CLASS DESCRIPTIONS

Within this part of the document the most important classes's properties are explained.

FamixEntity	**TODO**
Name	The name of the entity
uniqueName	The whole unique name of the entity beginning with the package it belongs to

FamixBehaviouralEntity	Extends FamixEntity. Containing the Functions and the Methods. The Husacct Tool doesn't see distinction between these two kinds and stores every method and function in the FamixMethod class.
accessControlQualifier	Public, private, protected or package private
signature	The method name including the parameters types. i.edoSomething(String, int)
isPureAccessor	Whether it is static or not.
declaredReturnType	The return type if not void.

FamixMethod	Extends FamixBehaviouralEntity. Contains all the functions and Methods. The Husacct Tool doesn't see distinction between these two kinds and stores every method and function in the FamixMethod class.
belongsToClass	The unique name of the class containing the method.
hasClassScope	**TODO**
isAbstract	Whether the method is abstract or not.
isConstructor	Whether the method is a constructor or not.

FamixPackage	Extends FamixEntity. Represents a physical package.
belongsToPackage	The unique name of the package it belongs to. This doesn't work for the root package, but does work for inner packages.

FamixClass	Extends FamixEntity. Represents a physical class.
isInnerClass	Boolean representing whether the class is an inner class
isAbstract	Boolean representing whether the class is abstract
hasInnerClasses	Boolean representing whether the class has inner classes.
belongsToPackage	The unique name of the package that the class belongs to
belongsToClass	The unique name of the class that the class belongs to. Works only for inner classes.

FamixStructuralEntity	Extends FamixEntity. FamixStructuralEntity is a superclass over attributes, variables and parameters.
declareType	The uniqueName of the class that the entity refers to
belongsToClass	The unique name of the class that the entity belongs to.
lineNumer	The linenumber where the entity can be found in the class.

FamixAttribute	Extends FamixStructuralEntity. Represents an attribute. An attribute looks like as follows: User testUser; testUser is the name of the attribute while it refers to the class called 'User'.
accessControlQualifier	Public, private protected or package-private
hasClassScope	Indicates whether the attribute is static or not

FamixFormalParameter	Extends FamixStructuralEntity. Represents a parameter.
belongsToMethod	The unique name of the method it belongs to.
declaredTypes	The return type of a parameter could simply be a String, int, double etc but it can also be a list containing other declaredTypes such as arrayLists and Hashmaps. In that case, all of the items from that list can be stored in declaredTypes. i.e. if this is a parameter: HashMap<User, HomeAddress> then the returntype is still a HashMap, but now the declaredTypes have 2 properties: a User and a HomeAddress object.
Extra info	There are agreements about how to store the unique name of a parameter. Say we have i.e. the following method with two parameters: doSomething(String varString, intvarInt) the unique name of the first parameter varString looks as followed: 'uniqueclassname.doSomething(String, Int). varString'

FamixLocalVariable	Extends FamixStructuralEntity. Represents an attribute within a method or function.
belongsToMethod	The unique name of the method it belongs to.
Extra info	There are agreements about how to store the unique name of a local variable. Say we have i.e. the following method: doSomething(String varString){ IntvarInt = 0; } The uniqueName of the local variable varInt looks as followed: 'uniqueclassname.doSomething(String).varInt'

FamixAssociation	FamixAssociation is a superclass, that each kind of dependency can extend.
Type	The type of the dependency. Although you can see what kind of dependency it is by checking it's instance of the subclass, there are dependencies which have different kind of types within that same subclass. Examples are: import, declaration, implements, extends etc
From	The unique name (package.classname) of the class which contains the dependency
To	The unique name of the class which the dependency refers.
lineNumber	The linenumber where the dependency can be found in the class.

FamixInvocation	Extending FamixAssociation. Invocation is an invocation of a class. There are three kind of invocations: 1) invocConstructor: This is the type when a new object is created i.e. 'new User();' 2) invocMethod: When a method is called in an object. 3) accessPropertyOrField: When a public attribute is called in an object.
invocationType	**TODO**
nameOfInstance	The name of the method or public attribute that is called.
invocationName	The name of the object within the class that holds the dependency.

FamixImport	Extending FamixAssociation. Imports are usually declared at the beginning of a class and holds a reference point to the class or package some dependencies refer to.
importingClass	The unique name (package.classname) of the class which contains the import
completeImportString	The complete string of the import i.e. husacct.package1 or husacct.package1.*
importCompletePacakge	Boolean. Indicates whether the import imports a single class or a whole package with the * symbol.