University of applied science

Analyse C# component

Authors:
Thijmen Verkerk
Thomas Schmidt
Martin van Haeften
Mittchel van Vliet

Supervisors:
Leo Pruijt
Michiel Borkent
Christian Köppe

June 13, 2012

CONTENTS

CHAPTER 1

INTRODUCTION

The following document describes the functionality and internal workings of the component **analyse C#**. The component is designed to work independently according to component-based architecture. Our component is a part of the analyse component. It consists of multiple components at the moment. The actual analysing of an application happens in the following components:

- Application analyser component.
- Java analyser component.
- C# analyser component.

After the analysing, the analysed data is stored in a language independent model called Famix. This happens in the following components which are also part of the analyse component:

- Model Service component.
- Analyse Domain.

We helped with designing the application analyser component and the first set-up of the Java analyser component. The application analyser component helps to abstract the components in a way that enables future component to be made without changing the other specific components. Every component has to deliver a complete Famix model, so the rest of the application can work with the information independent of a language. This happens in the Model Service and Analyse Domain components

Our team consists of four team members:

Name: **Thomas Schmidt**
Student number: 1549483

Name: **Thijmen Verkerk**
Student number: 1568823

Name: **Martin van Heaften**
Student number: 1572334

Name: **Mittchel van Vliet**
Student number: 1566264

CHAPTER 2

FUNCTIONALITY

This chapter describes the functionality and the internal workings of our component. It will be explained by various diagrams and textual explanations, such as use cases and sequence diagrams.

2.1 Use case diagram

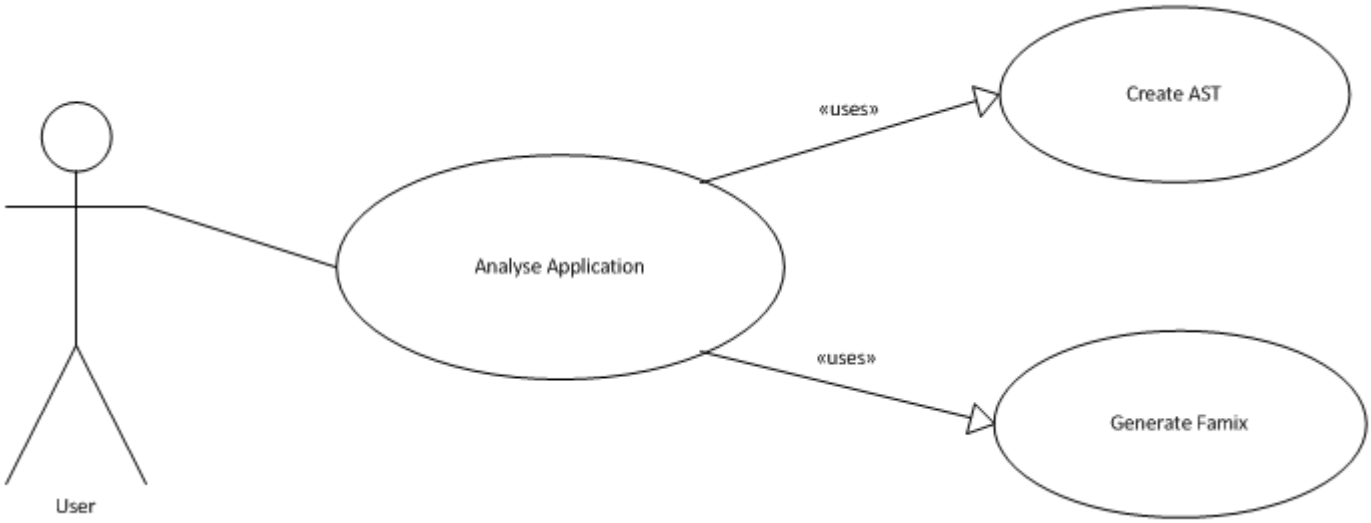

Figure 2.1: Use case diagram

2.2 Use case descriptions

2.2.1 Analyse application

ID	**Use case_1**
Name	Analyse Application
Actors	End-User
Related use case	Subcase Use case_2 (Create AST) Subcase Use case_3 (Generate Famix)
Precondition	Actor submitted the root path or paths of an application and language.
Postcondition	The application is scanned and a FAMIX model is generated.
Description	This use case uses the 'Create abstract Syntax Tree' and 'Generate Famix model' use cases.
Constraints	- The duration of the analyse of a project must be less than 15 minutes. (NF6 architecture Notebook vs 2012-03-08). - The information found should be stored in a language independent model.

2.2.2 Create AST

ID	**Use case_2**
Name	Create Abstract Syntax Tree
Actors	External tool
Precondition	Provided: There is a root path or paths and a language given.
Postcondition	The application is scanned and an AST is generated.
Description	The C# analyser requests to analyse a project, which requires a project path. The analyser will generate an AST, and in this use case it is based on C#. The Analyser Service will return the generated AST back to the C# analyser.
Constraint	- See use case 'Analyse Application'.

2.2.3 Generate Famix

ID	**Use case_3**
Name	Generate Famix
Actors	The code mapper.
Precondition	An AST is generated.
Postcondition	There is a Famix model generated by using an AST.
Description	When there is an AST returned from the analyser, the C# generators will then let the modelService add object to the famix model. Using a Famix model the modelService creates a language independent model of the analysed application. Other components can use information from Famix.
Constraints	- See use case Analyse Application

2.3 Sequence diagrams

2.3.1 Analyse application - class level

This sequence diagram in figure 2.2 and 2.3 shows how the component works on class level. The CSharpTreeConvertController makes an AST of an ANLTR parser. For performance reasons, we go through the AST once. The children of the AST are being sent to the several convertControllers, and they check if that object needs that part of the whole AST.

If a convertController collected enough data, it will be sent to the generator. Sometimes a generator finds another kind of dependency, then it is allowed to send it to that generator. For example, the ClassGenerator is able to find an interface. So it will make the InterfaceGenerator instead. In the generator it will call the ModelCreationService with the right data that it received from the convertController.

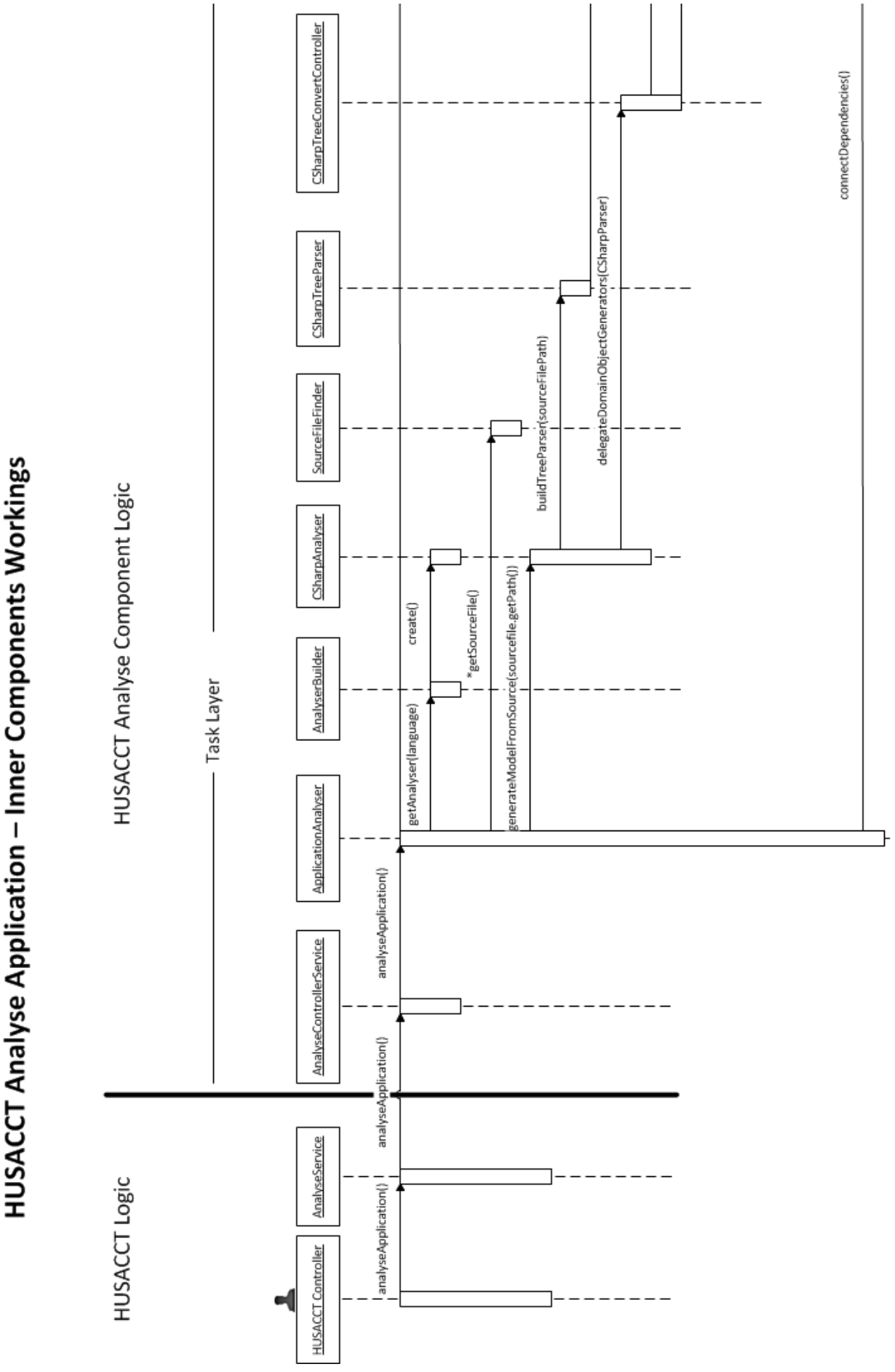

Figure 2.2: Analyse application on class level part 1

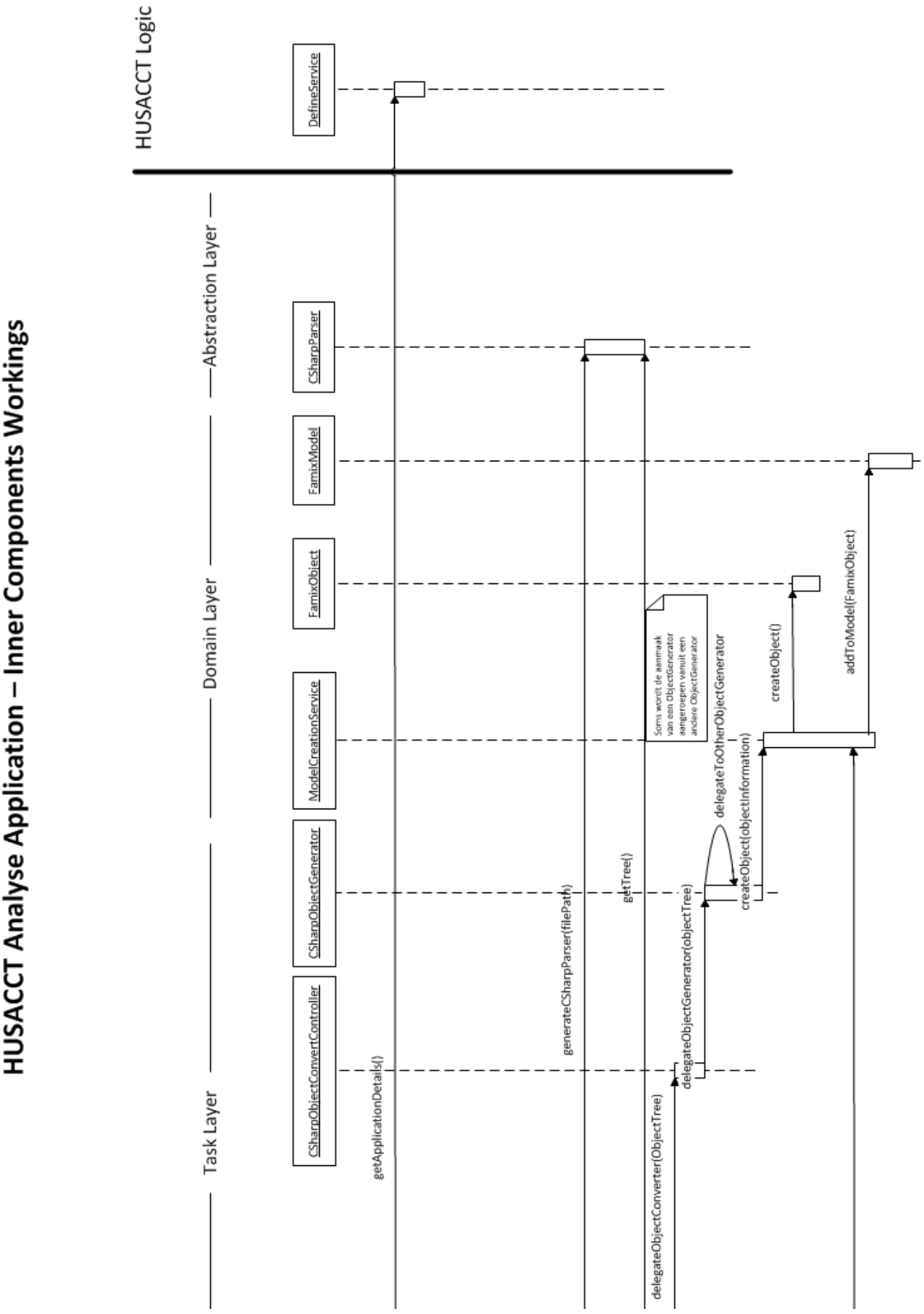

Figure 2.3: Analyse application on class level part 1

2.3.2 Analyse application - Service level

This sequence diagram is designed on service level. It combines the two use cases (Create Abstract Syntax Tree (Use Case_2) and Generate Famix (Use Case_3)). The Analyse component retrieves the AST(Create AST (Use Case_2)) from the Infrastructure service. The application loops through the AST and generates specific Famix objects.

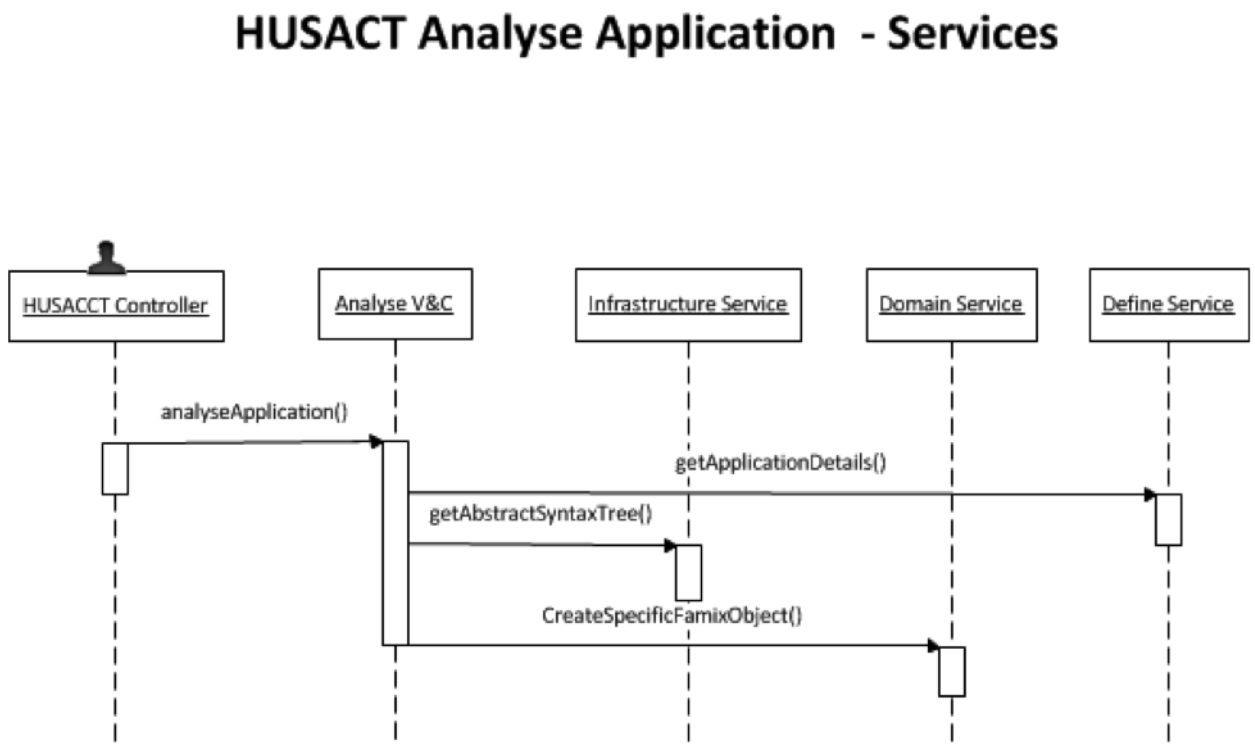

Figure 2.4: analyse application on service level

CHAPTER 3

FUNCTIONAL REQUIREMENTS

3.1 Functional requirements

The following functional requirements are dependency recognitions that the analyse part of the HUSACCT tool must recognize when analysing code. Except for rule number 5.9 these are all rules from the V3ISTO 2012: HUSACCT Requirements document.

5.1	Invocation of a method/constructor
5.2	Access of a property or field
5.3	Extending a class/struct
5.4	Implementing an interface
5.5	Declaration
5.6	Annotation of an attribute
5.7	Import
5.8	Throw an exception of a class
5.9	Indirect dependencies (i.e. an extend of an extend)

Table 3.1: Functional requirements

3.2 Non functional requirement

These non functional requirements (NFR) are taken and filtered out of the V3ISTO 2012: HUSACCT Requirements document.

#	**ISO 9126 attributes**	**Requirement**
2.1	Maturity	The tool must not go down in case of a failure, but generate a meaningful error message.
2.2	Fault Tolerance	There must be no restrictions in the size of the project regarding number of classes, lines of code, components...
4.1	Time behaviour	Tools must not take long (¡= 15 min; 1.000.000 LOC) to analyse/validate the code, and to generate an error report.
5.1	Analyse-ability Testability	Taking over the development of the tool by other development teams must be unproblematic.
5.2	Changeability	The tool must be easily extendible to other code languages.

Table 3.2: Non functional requirements

3.3 Decisions and Justifications

Decision	**Justification** (NFR)
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.	#4.1
Famix will be used as an independent data domain model structure. Famix enables the translation of different programming languages into one complete domain model, which a higher layer can use. As new programming languages are added to the tool with other concepts, the Famix model can be expanded by adding extra data.	#5.2
ANTLR has been chosen instead of PMD. When the project was started, our team had to look at the results from last year. This team used PMD for detecting dependencies. Our team could not use this to detect every possible dependency because of these multiple reasons: - The internal structure of HUSACCT had to be completely different to support PMD. For example: The defining has to be prior to the analysing. - PMD only works with Java. - There was not enough possibilities to modify the process, so the project could use it in a way we defined. These are the reasons why ANTLR has been used to detect dependencies. ANTLR uses grammars to convert source code into Abstract Syntax Trees. 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://www.antlr.org/	#5.1 #5.2
The analyser will process source code in two steps. At first, the code base will be converted in an AST (Abstract Syntax Tree). The second step is to convert the AST to the Famix model through an interface. Using this steps, it will be easier for developers to replace the Famix model with another model.	#5.1 #5.2
HUSACCT loops the tree once instead of multiple times for performance reasons, it is much faster. It is more difficult to generate the right data if you loop once. Because then you can not split up the tree in different parts, which you analyse separately.	#4.1

Table 3.3: Decisions and Justifications

CHAPTER 4

SOFTWARE PARTITIONING MODEL

This is the Software partitioning model, it shows a model of the complete Analyse component with components and layers.

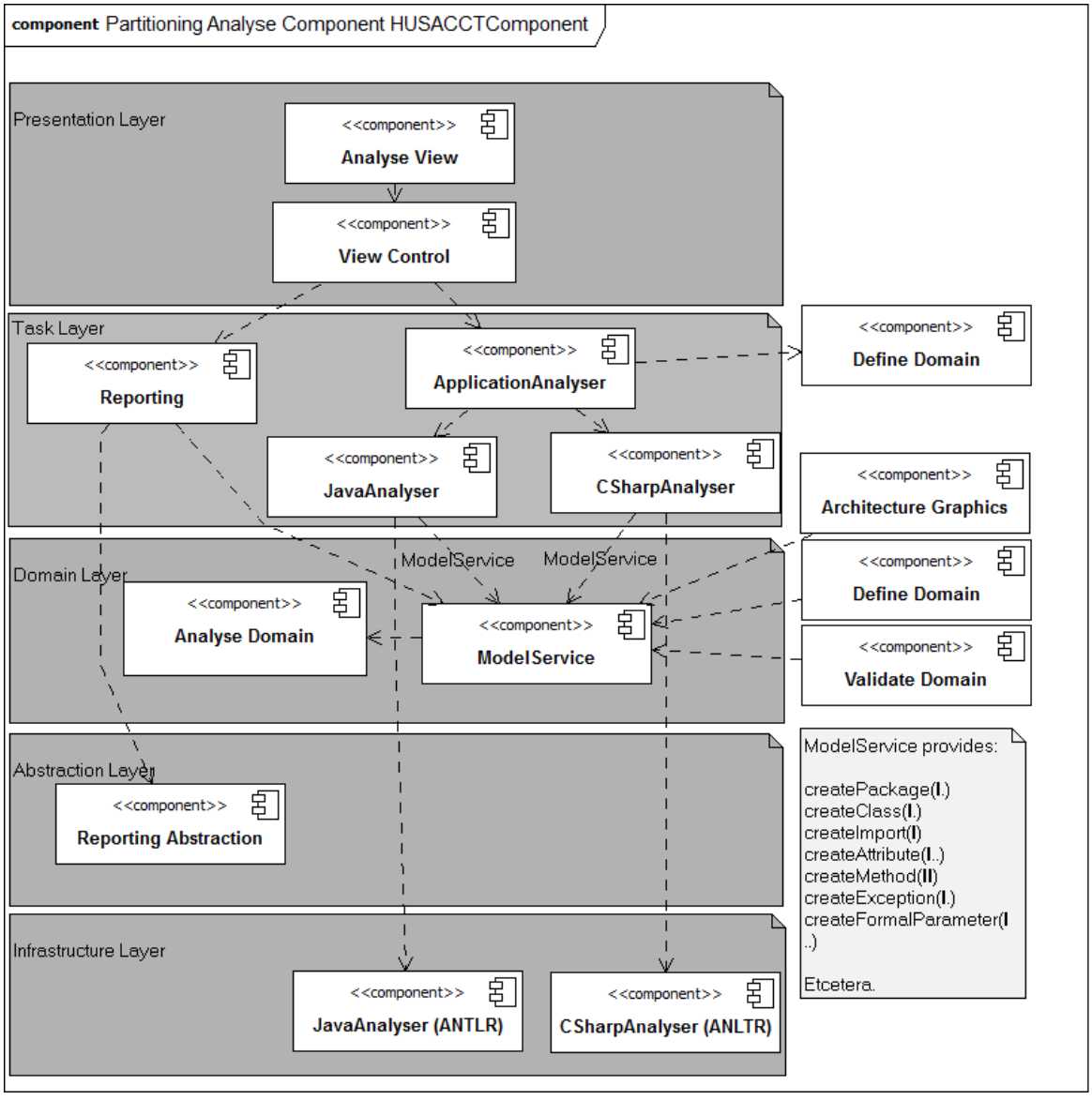

Figure 4.1: Software partitioning model

CHAPTER 5

SUBSYSTEM SPECIFICATION

We designed the C# analyser as a component that is part of the analyse component. Our component does not contain any subsystems because the systems to take care of creating the Famix model, querying the Famix model etc. are a part of the analyse component. For further clarification of our component we added a class diagram figure 5.1 with the needed classes for our component and the Java component.

5.1 Analyse Component class model

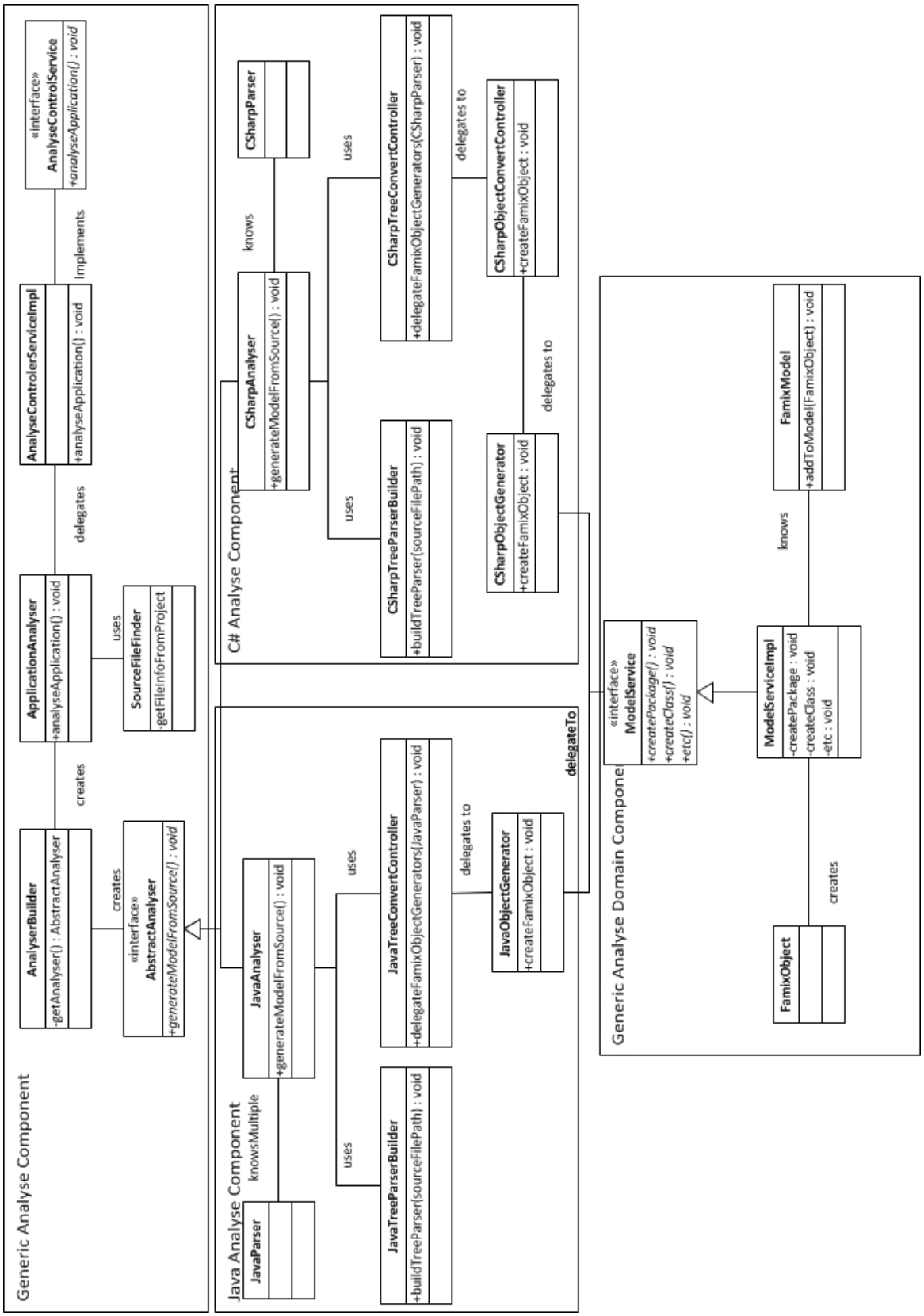

Figure 5.1: Analyse component classes

CHAPTER 6

TESTING

Due to the lack of time given to us in the project, there are no test cases specified. We did some performance tests with a variety of project differing in size and complexity. The test results are given below.
Used machines:

#	Processor	Memory	OS	HDD
1	1.86 GHZ Core 2 Duo	4GB 1067 MHZ DDR3	OSX 10.7.3	SSD
2	1.7 GHZ Core i5	4GB 1333 MHZ DDR3	OSX 10.7.3	SSD
3	2.83 GHZ Quad Core Q9550	4GB 1333 MHZ DDR3	Windows 7	SATA 2

Table 6.1: Used machines

For every software project, we did three tests on each of the machines. Below are the average results for each machine

HU Benchmark - 8729 lines of code			
PC	memory at start	memory at end	time
1	84.6MB	125MB	44 sec
2	97MB	122MB	31 sec
3	70MB	211MB	19 sec
Cosmos - 26413 lines of code			
1	84.6MB	140MB	27 sec
2	94MB	136MB	19 sec
3	69MB	160MB	12 sec
Fspot - 125024 lines of code			
1	84.6MB	192MB	1.45 min
2	95MB	193MB	58 sec
3	69MB	308MB	1.33 min

Table 6.2: Average result

CHAPTER 7

FUTURE WORK

At the end of the project, we noted that some things are not working properly. We also give some advise for the future of the C# Analyser.

7.1 Known bugs

#	Known Bug
1	There are some problems with several encodings, which are not commonly used in C#. They can occur in projects made in a tool which uses a different encoding than Visual Studio in source files. With our grammar, ANTLR gives an OutOfMemoryException with these encodings.
2	System variables, such as console, and variables instantiated in external libraries cannot yet be detected.
4	Static invocations are not being detected properly and are classified as declaration in the dependency overview.
4	Attributes as generics are not detected properly
3	Invocations in for-loops cannot be detected.
4	Lambda expressions and delegates cannot be detected.
5	Sometimes ANTLR gives parsing errors, because the current grammar cannot parse everything of C#.
6	Invocation on exceptions cannot be detected.

Table 7.1: Known bug list

7.2 Future improvements

We tested the scanning capabilities of the software, we saw that the scanning only took one thread while scanning. In the future there should be some research to the impact on the scanning capabilities with multi-threading. There is a good change it would improve the overall performance in the scanning of projects, especially on multi-core computers.

The grammar used in ANTLR was the only one available at the moment, we saw that is was far from perfect. When possible it should be replaced by a better working grammar. While being in the second iterator of the elaboration phase, a new grammar has been released. This seems to be a better grammar. When considering switching to this grammar, every convertController and Generator should be rewritten.

7.3 Future extensions

At the moment our component has some known bugs. To improve the analysing, they should be fixed in the project.

When extending the analyse component for the C# scanner you might have to make several new classes and subclasses. For clarification on how to do this with the current grammar, we will sketch a scenario for

implementing the recognition of a lambda expression. The following steps are required:

1. CSharpTreeConvertController
   In this class there has to be a List of CommonTrees which holds a list that has CommonTrees of the lambdas. There is meta-data and content of every token in the CommonTree.
   The in the CSharpTreeConvertController there is a method called walkThroughAST. This method loops through the primary AST. In here you declare your custom ConvertController. You also need to have a boolean which is true if it is a part of the lambda. The tree will be send to the custom ConvertControlle.

2. CSharpLambdaConvertController
   In the CSharpLambdaConvertController you get every CommonTree from the CSharpTreeConvertController. Then you check for tokens which the lambda uses. For example: "=>" is an example of something specific for lambdas, these are two tokens. So we check for "=" and then for ">". After you have enough tokens, you send it to the CSharpLambdaGenerator

3. CSharpLambdaGenerator
   Here you take everything out of the Tree that has been collected by the CSharpLambdaConvertController. It should extend CSharpGenerator to implement some generic variables. You define every token that has been collected, such as identifier, equals sign, local variable and what class does it belong to. This data is then send to the modelService. Unfortunately, the modelService doesnt have a definition for the Lambda generator.

4. ModelService
   The ModelService is a service which adds an FamixObject to FamixModel. We need to create an FamixLambda for this purpose first. In this class we extend FamixStructuralEntity, and then we add Lambda-specific logic. Next, we add a method to the IModelService, this is an interface that is necessary for the ModelService.