|  |
| --- |
| HUSACCT Tool |
| HU Architecture Compliance Tool  19-04-2012 |
| Architecture Notebook Analyse Component |

Team 3

Erik Blanken

Asim Asimijazbutt

Rens Groenveld

Tim Muller

Team 6

Thijmen Verkerk

Thomas Schmidt

Martin van Haeften

Mittchel van Vliet

# Table of Contents

1. Introduction 2

2. Architectural Significant Requirements 3

2.1. Functional Requirements 4

2.2. Non Functional Requirements 4

3. Architectural Significant Requirements 5

4. Usage of Design Patterns 6

5. Architectural Significant Requirements 7

6. Architectural Significant Requirements 9

6.1. Physical Components and Layers 9

# Introduction

HUSACCTS software is software with which you can check the architecture of software development projects. You can define an architecture on pre-hand, analyse an existing project, and compare these two to see if there are any architecture flaws in your project.

The software is divided into different pieces, each having a team of students assigned to develop. This document is purely about the analyse part of the software. The analyse part means the scanning of an existing project on a file system, and convert it into a domain model the entire application can use. We will describe how we do this in this document.

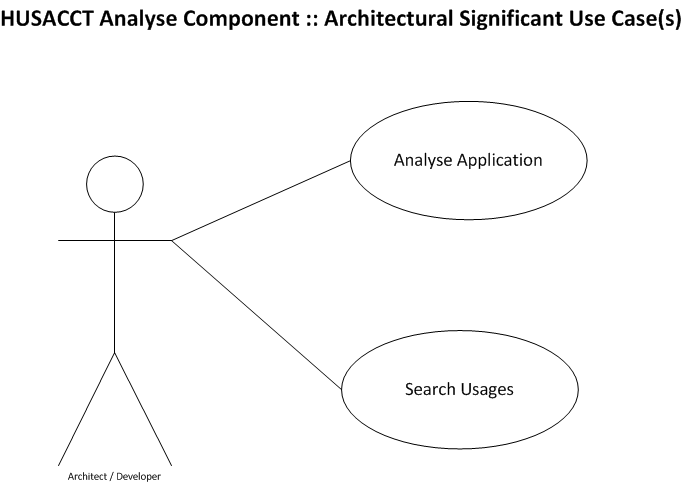
Because there are multiple teams working on this project, the architecture notebook is a good communication document for communication with other teams.

Finally, it are two teams that will have to develop the analyse part of the software. There is one Java team and one C# team. Because of the research about the tools that we can use to generate AST’s\*, the first part of the design and development of the HUSACCT analyse-component will be performed by both teams. When the decisions about the tools to use are made, each team will split up and proceed with the development of their own component.

# Architectural Significant Requirements

The requirements for the Analyse part of the HUSACCT 2012 tool are described in this chapter. Figure 1.0.

gives a basic overview of the architectural significant requirements for this component.



*Figure 1.0. Architectural Significant Functional Requirements for the HUSACCT-tool Analyse-component*

### Functional Requirements

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

|  |  |  |
| --- | --- | --- |
| Listing 1.2. HUSACCT Analyse Non-Functional Requirements | | |
| # | ISO 9123 attr. | 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 behavior | Tools must not take long (<= 15 min; 1.000.000 LOC) to analyse/validate the code, and to generate an error report. |
| 5.1 | Analyseability  Testability | Taking over the development of the tool by other development teams must be unproblematic. |
| 5.2 | Changeability | The tool must be easily extendable to other code languages. |

|  |  |
| --- | --- |
| Listing 1.1. HUSACCT Analyse Functional Requirements | |
| 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 |

### Non Functional Requirements

These non functional requirements are filtered out of the ‘V3ISTO 2012: HUSACCT Requirements’ document.

# Architectural Significant Requirements

This chapter explains the decisions that were made and their justifications. They are based on the non funtional requirements. They will give a clear overview of the decisions made to meet some of the requirements. The requirements that are not explained in the table below are usually self-explanatory and thus not reviewed.

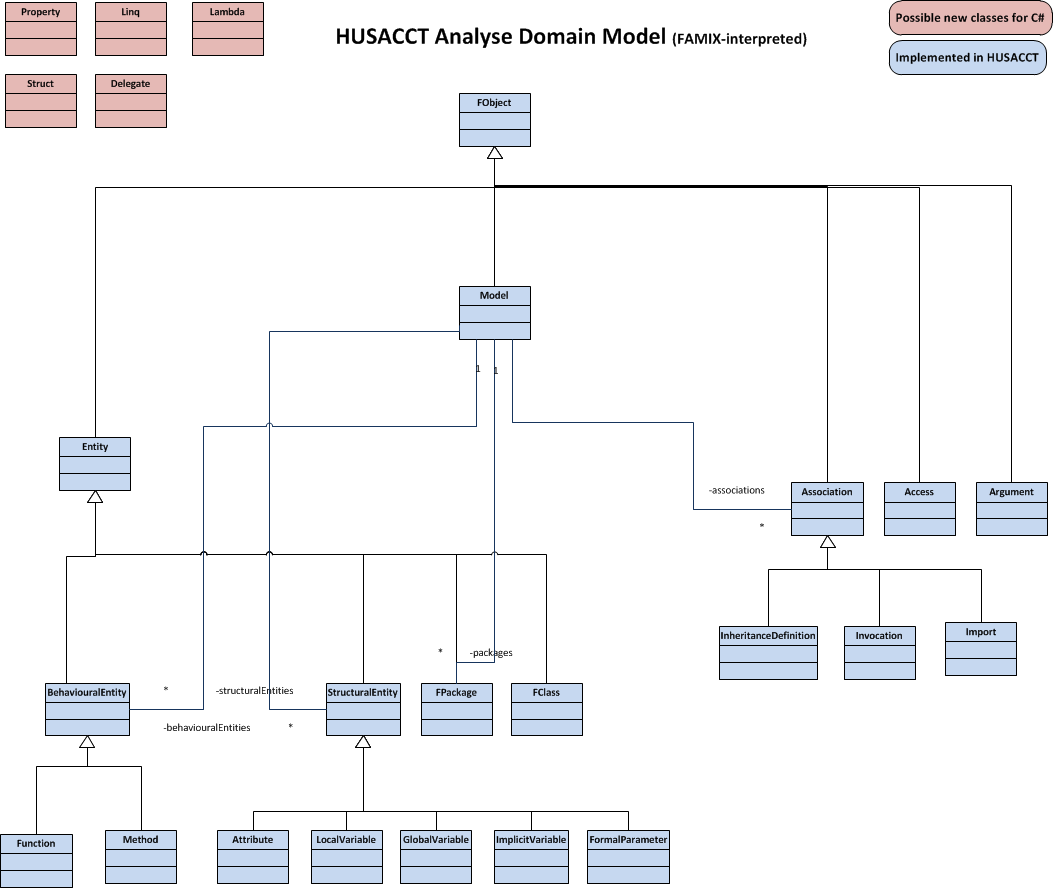
|  |  |
| --- | --- |
| Listing 2.1. HUSACCT Analyse Decisions & Justifications | |
| Decision | Justification (Non-Functional Requirement) |
| String filters will be used for incoming calls, so the validate component can quickly analyse the filter and return the right information. These filters are 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 different programming languages into one complete domain model, which an upper layer can use. As new programming languages are added to the tool, the FAMIX-model can be modified by adding extra data. | #5.2 |
| Because there was a lot of confusion about the workings of PMD which couldn’t be altered in the team’s favourite, Antlr has been used for the convertion from source files to Abstract Syntax Tree. Antlr uses so called grammars for this. Grammars can be written for any programming language, which means that other languages can be implementend in the future.    For more information about ANTLR: http://www.antlr.org/ | #5.1  #5.2 |
| The mappers will process codebases in two steps. At first, the codebase will be converted in an AST (Abstract Syntax Tree). The second step is to convert the AST to the FAMIX-model. Using this steps, it will be easier for developers to replace the FAMIX-model with another model. | #5.2 |

# Usage of Design Patterns

|  |  |
| --- | --- |
| Listing 3.1. HUSACCT Analyse Design Patterns | |
| Pattern | Reason |
| Service Layer Pattern | When a component of another team asks us for a piece of analysed information from the the Famix Domain, we will have to give this back. The call to fetch data is handled by a wrapper around the domain. This wrapper will know about the domain, fetch the correct data from it’s objects, transforms it into a DTO and give it back upwards to where the call came from. |
| Builder Pattern | The analyse part has to be able to analyse Java and C# code. In the future though, extention towards analysing other programming languages must be easy. A builder pattern will be used to instantiate or construct the correct mapper, based on a String-parameter that represents the programming-language. |
| Strategy Pattern | In order to enable different implementations of analysing source-code, the strategy-pattern will be used in combination with the builder=pattern. All existing mapper implement the analyse-functionality in their own way. |
| Assembler Pattern | To have a somewhat loose coupled architecture, the assembler pattern will be used to decouple DTO’s from the controlling service. The idea is that each DTO has it’s own controller class, that controls the DTO. |
| Singleton | The Singleton pattern is used for the famix model. This is because there is only one project analysed at a time. A Singleton makes it very easy for all classes that need to know or fill the model. It saves a lot of dependencies betweend classes. |

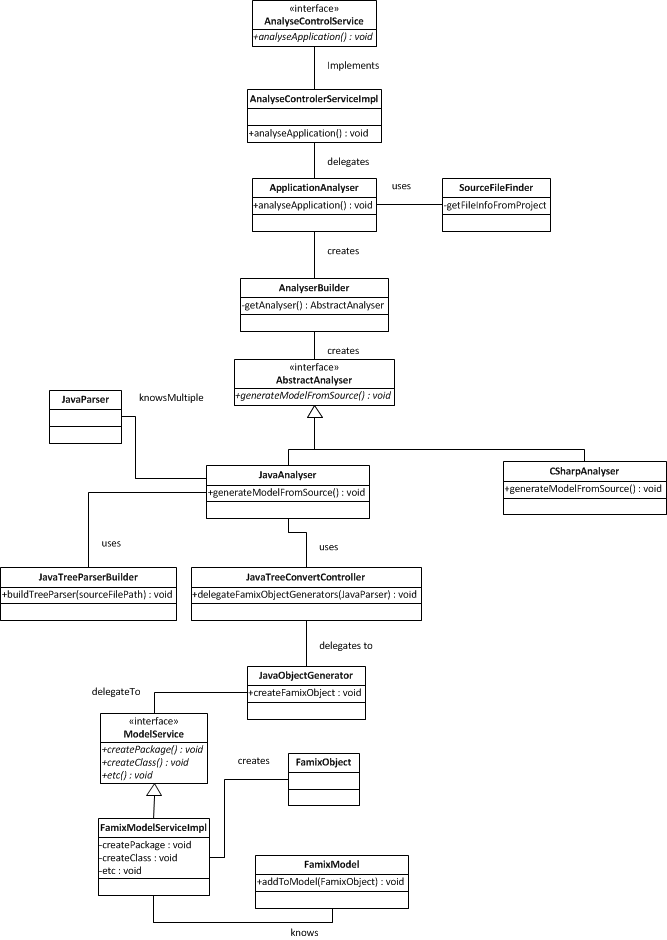
# Architectural Significant Requirements

The physical classes describe the classes that are in the analyse part of the HUSACCT software. This is purely based on the FAMIX model. The blue classes describe the classes that are needed for Java and C#. The red classes describe extra classes that might be needed for C#. For a new programming language to be added, it is very possible it just fits within this domain. If not, the domain must be expanded for that specific programming language. Although this is not desirable, the domain has a certain abstraction factor which makes it easy to implement new classes.



*Figure 5.0. HUSACCT FAMIX-Classes with possible new classes*

Figure 5.1 shows an overview of all the physical classes that will be implemented in the CodeMappers. This class-diagram also shows the implementation of a builder-pattern in combination with a strategy-pattern.



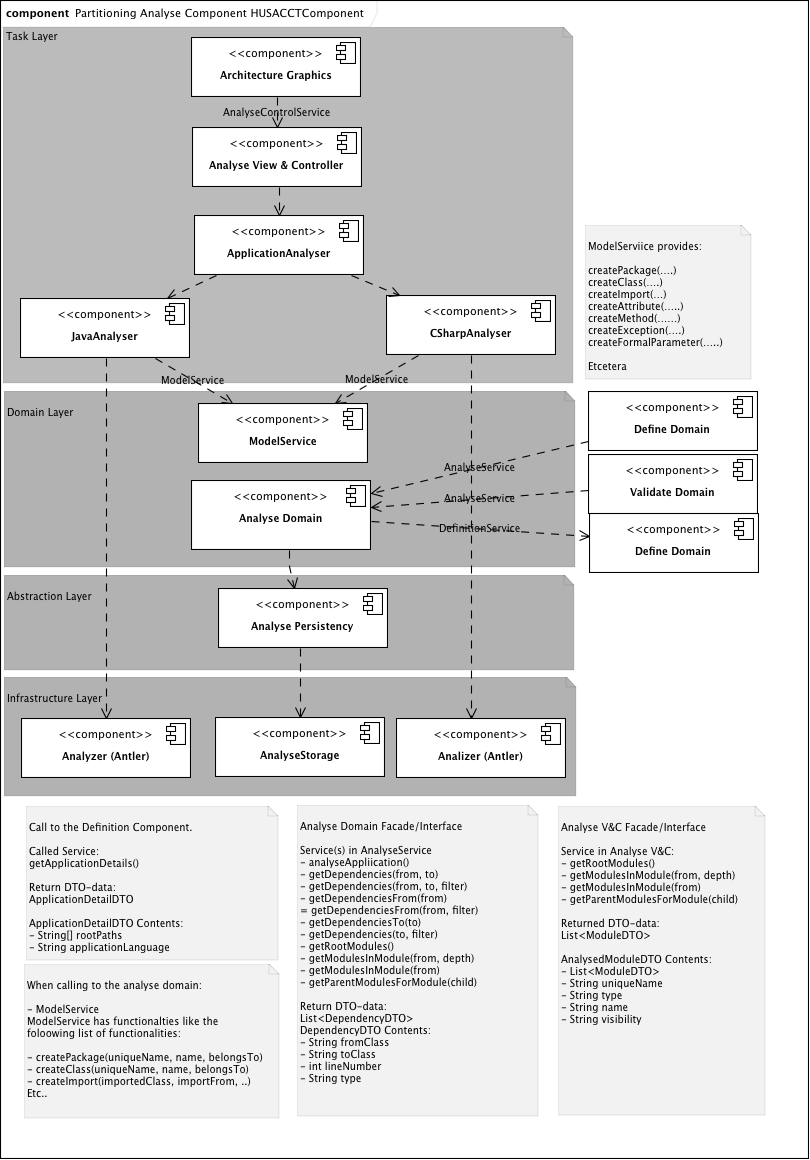
*Figure 5.1. HUSACCT Analyse JavaAnalyser-classes*

# Architectural Significant Requirements

**Partitioning the system** helps to manage its complexity by using the well-known "divide and conquer" strategy. Breaking the process into smaller and more manageable pieces, makes development easier.

**Layers**are hierarchical partitions of the functionality of the system with certain rules regarding how relationships can be formed between them. **Components** provide a way to isolate specific sets of functionality within units that can be distributed and installed separately from other functionality.

### Physical Components and Layers

 Logical Layers and their mapping to the Physical environment. All of the components will be build with Java.

*Figure 6.0. HUSACCT Analyse Partitioning, Interfaces & Data Transfer Objects*