|  |
| --- |
| HUSACCT SYSTEM DOCUMENTATION ANALYSE JAVA & C-SHARP |

|  |
| --- |
|  |

# CONTENTS

[CONTENTS 1](#_Toc442297090)

[1 Introduction Analyse Component 3](#_Toc442297091)

[1.1 FUNCTIONAL REQUIREMENTS 3](#_Toc442297092)

[1.2 NON-FUNCTIONAL REQUIREMENTS 4](#_Toc442297093)

[1.3 Architectural DECISIONS & JUSTIFICATIONS 5](#_Toc442297094)

[2 Software Architecture 6](#_Toc442297095)

[2.1 CONTEXT DIAGRAM 6](#_Toc442297096)

[2.2 TOP LEVEL PACKAGES & API 7](#_Toc442297097)

[2.3 Software Partitioning 8](#_Toc442297098)

[2.3.1 LAYERS 8](#_Toc442297099)

[2.3.2 Analyse COMPONENT PARTITIONING 9](#_Toc442297100)

[2.4 Responsibility Trace Table 11](#_Toc442297101)

[2.5 LAYERS & CLASSES 12](#_Toc442297102)

[2.6 organization Source code 13](#_Toc442297103)

[2.7 Famix model 14](#_Toc442297104)

[3 USE CASE DESCRIPTIONS 15](#_Toc442297105)

[3.1 Analyse Application 15](#_Toc442297106)

[3.1.1 From GUI-event to AnalyseServiceImp 16](#_Toc442297107)

[3.1.2 From AnalyseServiceImp to ApplicationAnalyser 16](#_Toc442297108)

[3.1.3 From ApplicationAnalyser to JavaAnalyser 17](#_Toc442297109)

[3.1.4 From JavaAnalyser to Domain (Famix model) 18](#_Toc442297110)

[3.1.5 Connect Structural Dependencies 19](#_Toc442297111)

[3.1.6 Recognizing Dependency Types and Sub Types 20](#_Toc442297112)

[3.1.7 UmlLinks as for use in UML Class diagrams 21](#_Toc442297113)

[3.1.8 Declaration of Type Parameters of Paramitrized Types (Generics) 22](#_Toc442297114)

[3.2 Search Usages 24](#_Toc442297115)

[3.3 Save analysed application 25](#_Toc442297116)

[3.4 Load Analysed Application 26](#_Toc442297117)

[4 Testing the analyse component 27](#_Toc442297118)

[4.1 TESTING DEPENDENCIES & MODULE-FINDERS 28](#_Toc442297119)

[4.2 TESTING LANGUAGE-SPECIFIC ANALYSERS 29](#_Toc442297120)

[5 Adding support for new programming languages 30](#_Toc442297121)

[5.1 CREATE A NEW ANALYSER 30](#_Toc442297122)

[5.2 MAKE YOUR ANALYSER AVAILABLE FOR THE APPLICATION 30](#_Toc442297123)

[5.3 START CREATING YOUR ANALYSER-FUNCTIONALITY! 31](#_Toc442297124)

[5.4 CREATING JUNIT TESTS FOR YOUR NEW ANALYSER 31](#_Toc442297125)

[6 HUSACCT Famix Implementation & Description 32](#_Toc442297126)

[6.1 Introduction 32](#_Toc442297127)

[6.2 Workflow 32](#_Toc442297128)

[6.3 Class Descriptions 33](#_Toc442297129)

Note: Currently, this documentation is not in all cases completely up-to-date, but this document provides a good overview of the Analyse component.

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Contents** |
| 2016-02 | Leo Pruijt | Added: Detection of UmlLinks and declarations ofType Parameters. |
| 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 |

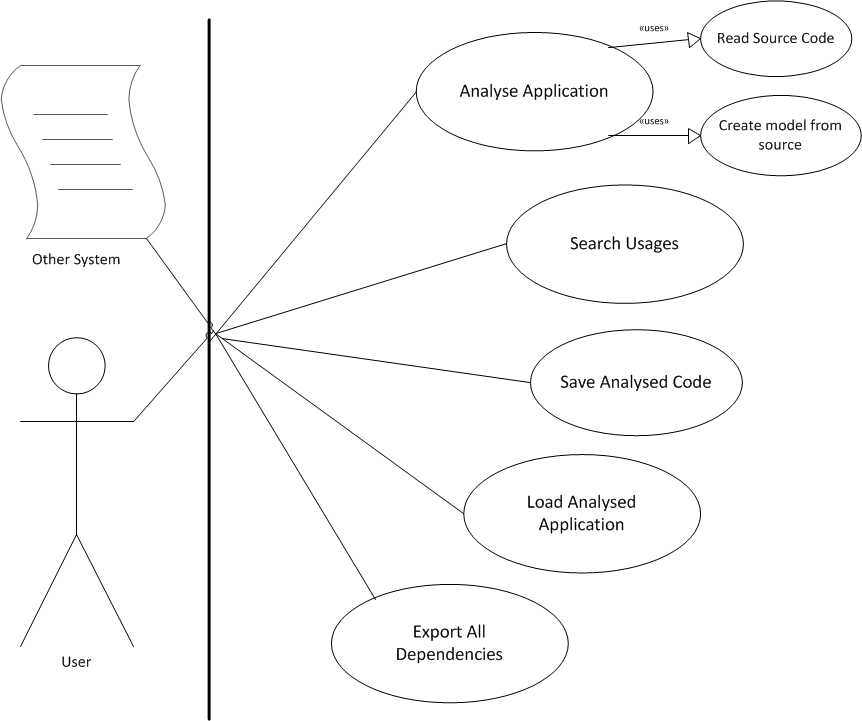
# Introduction Analyse Component

HUSACCT is a tool to support architecture reconstruction and architecture conformance checking. The HUSACCT software is divided into different components. This document is purely about the analyse component.

The most important task for SACC-tools is to be able to compare an implemented project architecture to a given intended architecture. In order to do so, the system must be able to scan source-code. The analyse component scans the source-files in the project directory and generates a model that represents that source. Currently, it supports Java and C#, but is relatively easily extendable by other programming languages.

## FUNCTIONAL REQUIREMENTS

The following diagram shows the use-cases implemented by the analyse component.



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. |

## 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... |

## Architectural DECISIONS & JUSTIFICATIONS

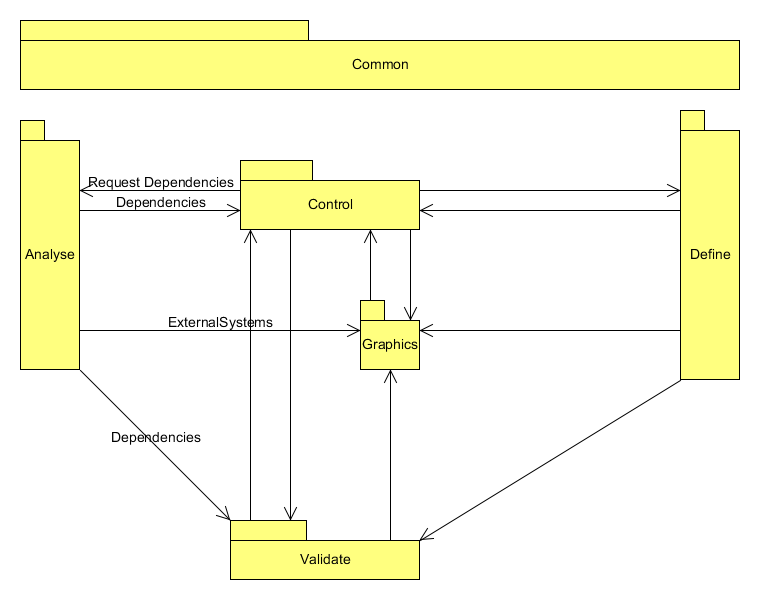
To address the non-functional requirements, some important architectural decisions were made:

|  |  |
| --- | --- |
| **Table 2.2. Analyse Decisions & Justification** | |
| Decision | Justification |
| To enhance the analysibility, 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://www.antlr.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. |

# Software Architecture

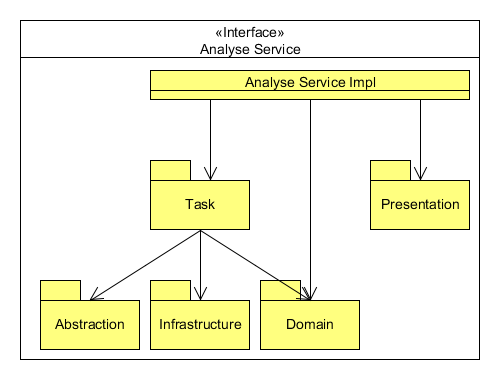
## CONTEXT DIAGRAM

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.



## 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.

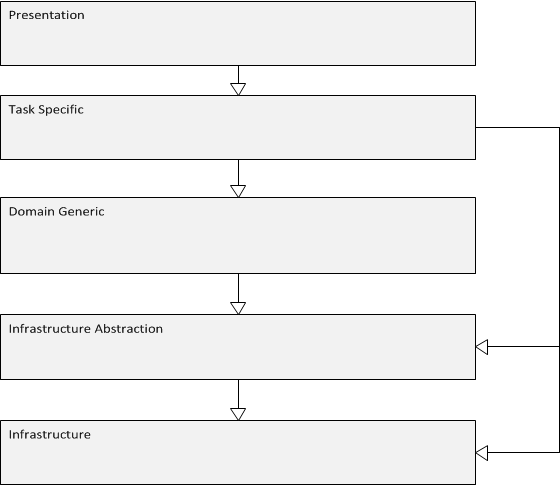
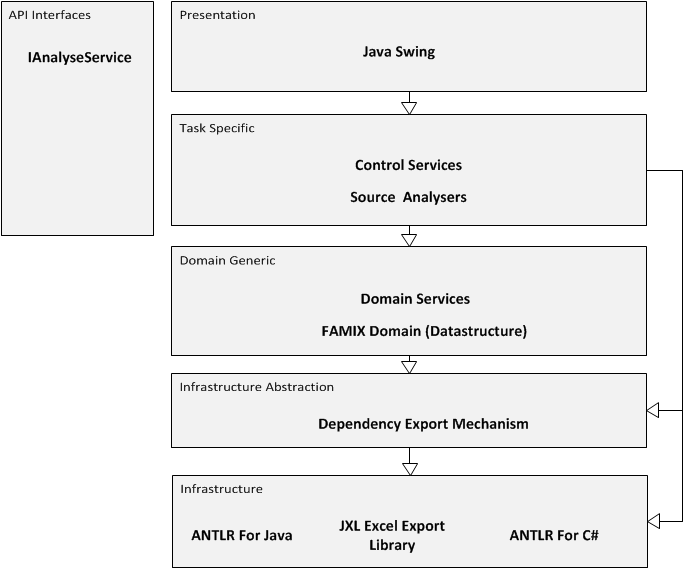


## Software Partitioning

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

### LAYERS

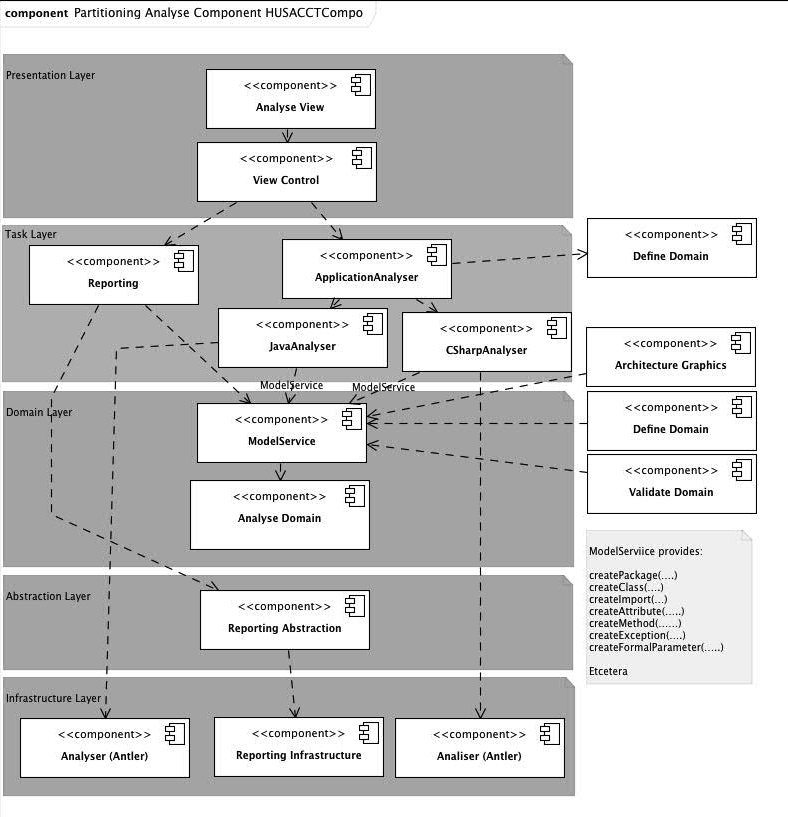
The software layers and the related dependency rules are specified below.



|  |  |
| --- | --- |
| **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-translaters 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. |

### 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 |
| Analyser(Antler) | Husacct/analyse/infrastructure/anlt/java  Husacct/analyse/infrastructure/anlt/csharp  Husacct/analyse/infrastructure/anlt/grammars/csharp  Husacct/analyse/infrastructure/anlt/grammars/java |

## Responsibility Trace Table

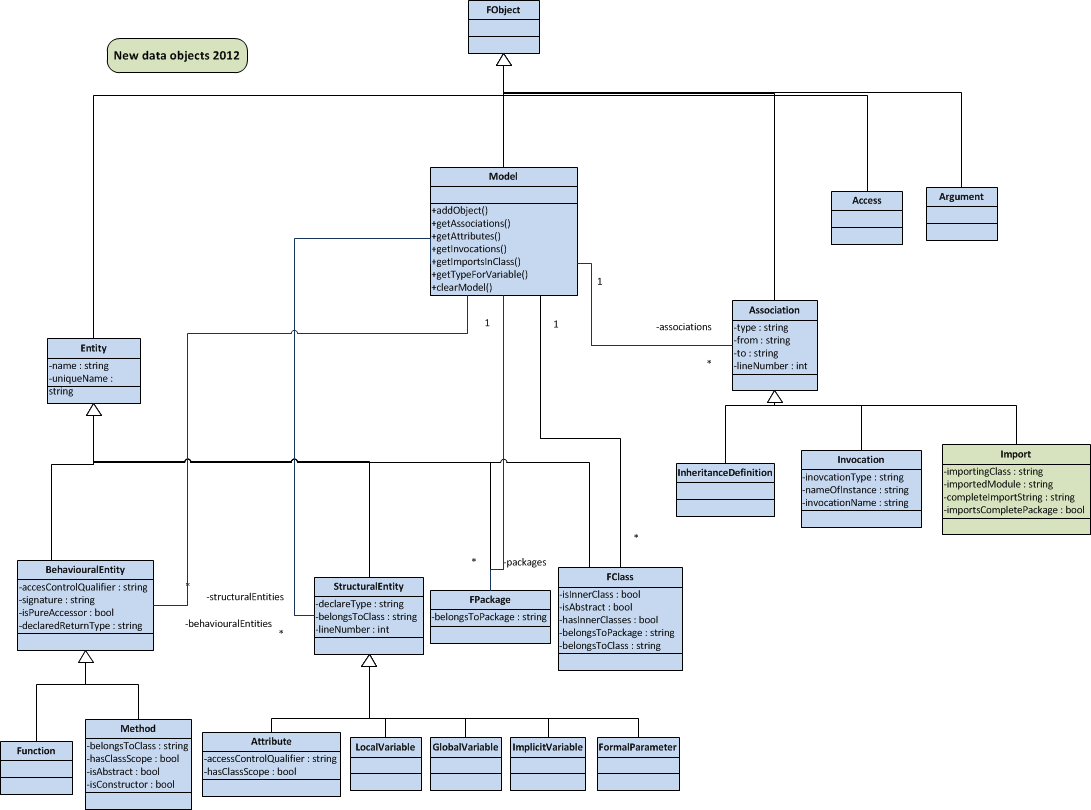
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type of Logic  → | Client Interface | | | Task Specific | | | Domain Generic | | | Infrastructure Abstraction | | Infrastructure | |
| Responsibility →  Software Layer /  Component  ↓ | 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 |
| 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 |

## LAYERS & CLASSES

## C:\Users\Alex Xia\Documents\Dropbox\TCIF-V3TOASE1-12 ANALYSE\HUSACCT Analyse Documentation 2013\Diagrams\DecompositionDiagram.pngorganization Source code

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

## 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.

# 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.**

## Analyse Application

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

|  |  |
| --- | --- |
| **Table 1.0. Textual Specification *Analyse Application*** | |
| 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 | - 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 | 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.  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.  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. |

### From GUI-event to AnalyseServiceImp



### From AnalyseServiceImp to ApplicationAnalyser



### From ApplicationAnalyser to JavaAnalyser

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



### 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.



### Connect Structural Dependencies



### Recognizing Dependency Types and Sub Types

For a definition of terms and for an overview of the Types and subtypes, read paper the following paper (search on title in Google Scholar and download the paper):

Pruijt, L., and van der Werf, J.M.E.M. (2015).   
Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking.   
In Proceedings of the 2015 European Conference on Software Architecture Workshop/ Second Workshop on Software Architecture Erosion and Architectural Consistency, ECSAW ’15, Cavtat, Croatia, Article No. 56. ACM Press.

#### Enums

Enums have been created for DependencyTypes and DependencySubTypes in version 4.5. These enums are not used everywhere yet, but use them where possible.

#### Implementation

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...”.

#### Testing and Notes

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

### UmlLinks as base for Associations in UML Class diagrams

In version 4.5, FamixUmlLinks are added. Such a link represents an uni-directed association between two classes in an UML class diagram. Bi-directional associations may be derived from the set of links between two classes.

Furthermore, FamixAttribute has been extended with two attributes (isComposite, typeInClassDiagram).

Finally, services are added to query UmlLinks and test cases to test the functionality.

#### OMG Definitions Regarding Associations

1. OMG Unified Modeling Language-Version2.5-15-03-01.pdf, especially 11.4, 11.5, 9.9.17 and 10.6.
   1. An Association declares that there can be links between instances whose types conform to or implement the associated types.
   2. A link is a tuple with one value for each memberEnd of the Association, where each value is an instance whose type conforms to or implements the type at the end.
   3. When one or more ends of the Association have isUnique=false, it is possible to have several links associating the same set of instances. In such a case, links carry an additional identifier apart from their end values.
2. Properties of an UML Association:
   1. TypeFrom
   2. TypeTo
   3. 2 Connector End: ToEnd and FromEnd
      1. Attrubute (Property/Role)
      2. Multiplicity
      3. isNavigable
      4. isGeneralization

#### FamixUmlLink

In HUSACCT.analyse.domain.FamixUmlLink, base information on the associations between classes in UML Class Diagrams is stored.

|  |  |  |
| --- | --- | --- |
| **FamixUmlLink Attribute** | **UML term** | **Explanation** |
| from : String | Type (From) | Type of owning object |
| to : String | Type (From), datatype | Referred type of object  Navigable = true at this end. |
| attributeFrom: String | Property (meberEnd), DataType | The short name of the attribute of the from- class that is responsible for the link. |
| isComposite : boolean | isComposite | If isComposite is true, the object containing the attribute is a container for the object or value contained in the attribute.  False: max multiplicity = 1  True: max multiplicity = n (or \*) |
| type : String | superClass, ownedAttribute, nestedClassifier, composite, aggregate | Current values are limited to: inheritanceLink (to superclass),  implementsLink (to interface)  attributeLink (from ownedAttribute) |

#### Rules to create FamixUmlLinks

**Pre-Processing (code analysis)**

1. Container with one Type: When an *instance attribute* has as type a generic type that contains one single type (type with one type parameters), while the type of the type parameter is not-composite, then FamixAttribute.isComposite is set to true, and typeInClassDiagram is set to the single type.
   1. Positive examples: Person[], or ArrayList<Person> In case of s where.
   2. Negative example: HashMap<int, Person>, Map<Person, Address>.
2. Container with two Types: When an *instance attribute* has as type a generic type that contains two types (type with two type parameters), while the type of the second type parameter is not-composite, then FamixAttribute.isComposite is set to true, and typeInClassDiagram is set to the second type.
   1. Positive example: HashMap<int, Person>, Map<Person, Address>.
   2. Negative example: HashMap<String, ArrayList<Person>>.

**Post-Processing**

1. For each FamixAttribute of each FamixClass, determine if it represents a link. If so, create a FamixUmlLink.
   1. Create a link if the following constraints apply:
      1. hasClassScope = false
      2. declareType or typeInClassDiagram refers to a FamixClass (not null, not “”, and not a FamixLibrary (starting with xLibraries)).
   2. If isComposite = false, then the value of declareType must be used to set FamixUmlLink.to.
   3. If isComposite = true, then the value of typeInClassDiagram must be used to set FamixUmlLink.to.
   4. See the Explanation per FamixUmlLink Attribute to determine the values of the attributes.
2. For each inheritance or implements dependency, create a FamixUmlLink
   1. In these cases: attributeFrom = “”, and isComposite = false.

#### Tests

Several test cases have been added to the Accuracy test, recognizable by the prefix UmlLinkType\_.

### Declaration of Type Parameters of Paramitrized Types (Generics)

Version 4.5 adds the detection and declaration of Type Parameters. Type Parameters are not (yet) stored as Famix objects. Only FamixAssociations are created to enable reporting of dependencies on the used types.

#### Rules

How is the type of a variable (parameter, return type) be reported in case of a container?

* As the type of the container.
* Exception: Array, since it is no container type. So report the type of the contained elements.
* In both cases: Add a marking that the attribute may contain multiple values.

How is the dependency on a Type Parameter of a generic be reported?

* Declaration (all subtypes, except Exception): Declaration, specific subtype. In case of multiple Parameter Types, create multiple dependencies.
* Call, Constructor of generic type: Report dependencies on these types.   
  DependencyType = “Declaration”; subType = “Type Parameter” (new subtype).
* Class with Type Parameters: Currently, do not report dependencies on these types, since these parametrized types cannot be identified correctly in all situations.   
  DependencyType = “Declaration”; subType = “Type Parameter” (new subtype).
* Each time a variable of a generic type is used (e.g., acces, or pass it as argument: do nothing (currently), because it is too complex (it even requires an extension of the Famix model), and the contained types are not actively used. When objects are added or retrieved to the container, their usage will be reported by the default functionality.

#### Implementation

* Declaration of Class variable, Instance variable, and Local variable is handled within JavaAttributeAndLocalVariableGenerator.
* Extend approach with the differentiation between one or more included types.
* Declaration of Parameters, Return Types and type parameters in Constructores is handled within JavaInvocationGenerator.getcompleteToString() and an method called in case of a list of argument types.
* The code of the Java and C# generators has been synchronized.

#### Unresolved Issues related to Generic Types

1. In case of a method call with generic type object as argument, no dependencies are reported on the parameter types of the generic type. Exception: Constructor call of a generic type.
   1. The detection mechanism would not work, since a “,” in an argument disrupt the algorithm , currently.
2. No declaration dependencies are reported on Type Parameters when a class is defined.
   1. The class name itself changes by usage of type parameters.
3. In C# generators, a mechanism is included to add parameters e.g. as <p1, p2> to the name of a class to enable dependencies on these classes. However, it is no perfect solution since two classes with the same name and the same number of parameters might be mingled.

#### Tests

Several test cases have been added to the Accuracy test, recognizable by postfixes like \_GenericType\_MultipleTypeParameters.

## 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 *Search Usages*** | |
| 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.



## 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 *Search Usages*** | |
| 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. |



## 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 en dependencies are created again from an xml element.

|  |  |
| --- | --- |
| **Table 1.3. Textual Specification *Search Usages*** | |
| Goal | Loadananalysed 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. |

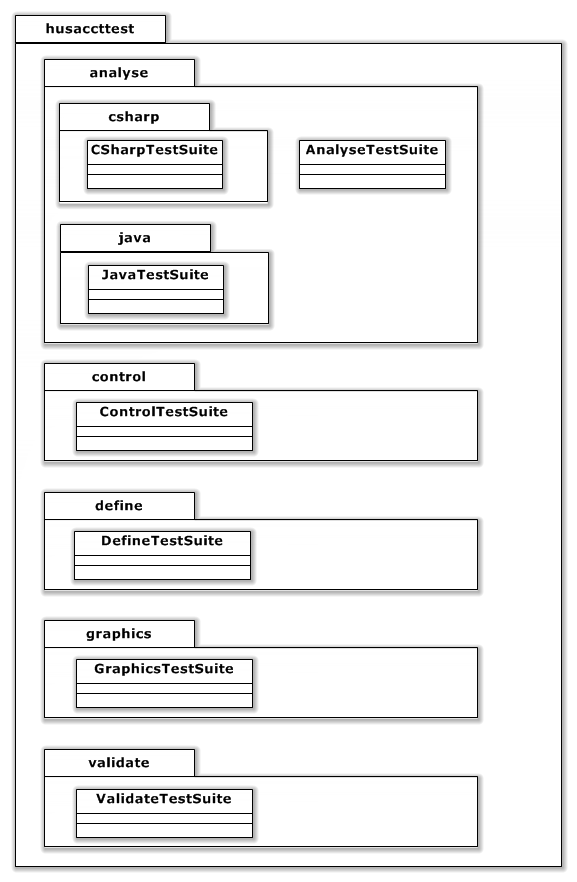


# Testing the analyse component

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.

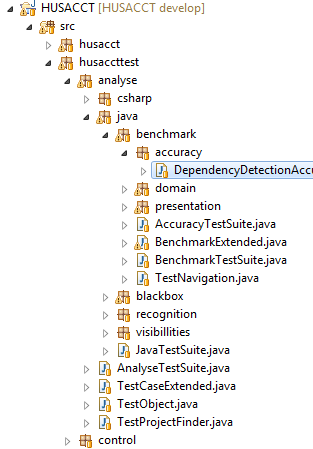
All tests are included in package husaccttest, subpackage Analyse, etc

To run a certain test, select and run a class as JUnitTest.



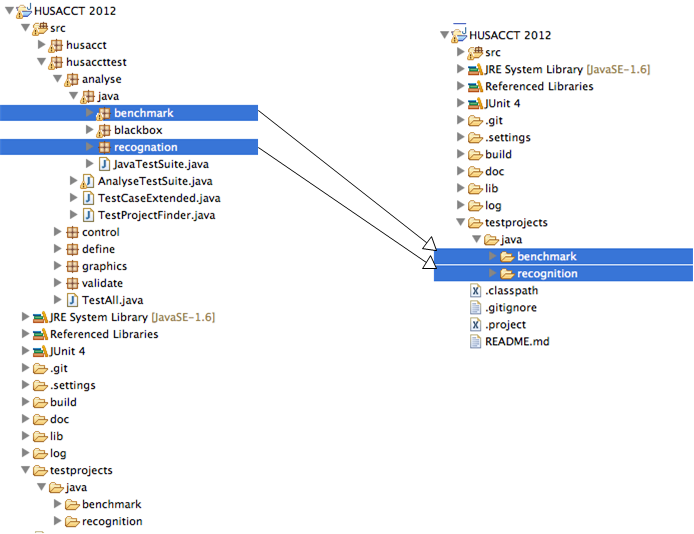
## TESTING DEPENDENCIES & MODULE-FINDERS

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.



## 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 al 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.



# Adding support for new programming languages

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

## 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)

## 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.



## 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 replaceabillity-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 sematics 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!** |

## 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:** String path = TestProjectFinder.*lookupProject*("java", "recognition"); |
| 2 | **Tip** | **If you don’t know something, just checkout one of the existing analysers to see how they have implemented their functionality!** |

# HUSACCT Famix Implementation & Description

## 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.

## 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 descision 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 dependecies are stored and recieved from, once the analysation 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.

## 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 uniquename of a parameter. Say we have i.e. the following method with two parameters:  doSomething(String varString, intvarInt)  the uniquename 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 uniquename 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:  ‘uniqeclassname.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 dependcy |
| 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. |