|  |
| --- |
| SYSTEM DOCUMENTATION  HUSACCT – VALIDATE |

Contents

[1. Introduction 4](#_Toc365533639)

[1.1 Terminology 4](#_Toc365533640)

[2. Functionality 7](#_Toc365533641)

[2.1 Use case diagram 8](#_Toc365533642)

[2.2 Check Conformance 9](#_Toc365533643)

[2.2.1 Textual description 9](#_Toc365533644)

[2.2.2 Check conformance mechanism 10](#_Toc365533645)

[2.2.3 Callback to the code viewer 12](#_Toc365533646)

[2.3 Configure configuration 13](#_Toc365533647)

[2.3.1 Configuration data repositories SeverityPerTypeRepository 13](#_Toc365533648)

[2.3.2 Configuration of the default and allowed rules 13](#_Toc365533649)

[2.4 Export report 14](#_Toc365533650)

[2.4.1 Adding a new Reporttype 16](#_Toc365533651)

[2.5 Export workspace 18](#_Toc365533652)

[2.6 Import workspace 19](#_Toc365533653)

[3. Decisions and justifications 21](#_Toc365533654)

[4. Software partioning 22](#_Toc365533655)

[4.1 Mapping of the physical classes to the software partitioning 23](#_Toc365533656)

[4.2 Relations with other components 24](#_Toc365533657)

[5. Dependency types 25](#_Toc365533658)

[5.1 Call 25](#_Toc365533659)

[5.1.1 Example 25](#_Toc365533660)

[5.2 Access 25](#_Toc365533661)

[5.3 Inheritance 25](#_Toc365533662)

[5.3.1 Example 25](#_Toc365533663)

[5.4 Declaration 26](#_Toc365533664)

[5.4.1 Example 26](#_Toc365533665)

[5.5 Annotation 27](#_Toc365533666)

[5.5.1 Example 27](#_Toc365533667)

[5.6 Import 27](#_Toc365533668)

[5.6.1 Example 27](#_Toc365533669)

[6. RuleViolation TypeKeys 28](#_Toc365533670)

[6.1 Rulekeys 28](#_Toc365533671)

[6.2 Violationtypekeys Java 28](#_Toc365533672)

[6.3 Violationtypekeys C# 28](#_Toc365533673)

[7. Subsystem specification 29](#_Toc365533674)

[7.1 Provided sevices of the validate component 29](#_Toc365533675)

[7.1.1 Provided methods 29](#_Toc365533676)

[7.1.2 Provided Data Transfer Objects 30](#_Toc365533677)

[7.2 Domain model 32](#_Toc365533678)

[7.3 Starting up/resetting the component 33](#_Toc365533679)

[7.4 Creating/generating ruletypes and violationtypes objects 34](#_Toc365533680)

[7.4.1 Violationtypes 34](#_Toc365533681)

[7.4.2 Generating violationtypes with dynamic classloading 34](#_Toc365533682)

[7.4.3 Ruletypes 35](#_Toc365533683)

[7.4.4 Generating ruletypes with dynamic classloading 35](#_Toc365533684)

[7.5 Adding new ‘types’ 37](#_Toc365533685)

[7.5.1 Adding a new ruletype 37](#_Toc365533686)

[7.5.2 Adding new violationtypes 38](#_Toc365533687)

[7.5.3 Adding a new programming language 39](#_Toc365533688)

[8. Testing 41](#_Toc365533689)

[8.1 ImportExportTest 41](#_Toc365533690)

[8.2 ValidateTest 41](#_Toc365533691)

[8.3 Testing the ruletypes 42](#_Toc365533692)

[9. Future work 43](#_Toc365533693)

[9.1 Ideas for improvement 43](#_Toc365533694)

[9.2 Ideas for extension 43](#_Toc365533695)

# Introduction

This document is written for developers and is meant to give them understanding how the validate component is built for the HUSACCT-tool.

HUSACCT is an abbreviation for Hogeschool Utrecht Software Architecture Compliance Checking Tool and is a tool to check if a software product is built according to the initially defined architecture. The tool is divided into five components:

* Control, the component that merges the different components and provides a user interface (UI) for this
* Analyse, analyzes the source code of the following programming languages:
  + Java
  + C#
* Define, defines the architecture and mapping of the analyzed files to a (logical) module.
* Validate, validates the defined architecture based on the analyzed source code.
* Graphics, is in charge of the graphical representation of the defined architecture and the scanned source code.

This document is focused on the validate component.

The following subjects will be treated in this document:

Use cases of the validate component consist of the following information, depending on the use case:

* goal description per use case;
* details per use case;
* references to packages/classes implementing the use case;
* sequence diagram to show the mechanism of the use case;
* and how to extend the use case with new functionality.

In this document there will be no overview of the different types of rules the tool supports. Please read the HUSACCT manual, to find this information.

## Terminology

Not all use cases will be covered in this document, but sometimes speciific terms will be used. This terminology will be explained in this paragraph.

History points  
When HUSACCT is used periodically, there might be the need to compare violations from different time. One point of the past is a history point.

Rule types

A ruletype is a rule that contains the logic and information to perform an validation and can be applied on ore more modules (depending on the ruletype). (For more information about modules see the documentation of the define component)

Violation types  
A violationtype is the type of the violation that can occur during the validation process. It is possible to filter on violationtypes in the define process (for more information, look into the documentation of the define component).  
Each ruletype has a different set of violationtypes. These volationtypes depend on the programming language from the source code that is scanned by the analyse component.

As an example we’ll take the ‘Is Not Allowed To Use’ rule, in table 1 the different violation types are displayed, based on their respective programming languages.

|  |  |
| --- | --- |
| PROGRAMMING LANGUAGE | VIOLATIONTYPES |
| Java | Constructor  Invocation of a method |
| C# | Delegate  Invocation of a method |

Table : Is Not Allowed To Use Rule

Do note that the information shown in table 1 is just example data and is not accurate.

Violationtype key  
To identify violationtypes each violationtype has a unique key.

In table 2 an overview is given of the currently supported violationtype keys.

|  |  |
| --- | --- |
| KEY | VIOLATIONTYPE |
| InvocMethod | Invocation of a method |
| InvocConstructor | Invocation of a constructor |
| AccessPropertyOrField | Acces of a property or field |
| ExtendsConcrete | Extending a concrete class |
| ExtendsAbstract | Extending a abstract class |
| ExtendsLibrary | Extending a class from a library |
| Implements | Implementation of a interface |
| Declaration | Declaration |
| Annotation | Annotion of a class or attribute |
| Import | import (used and unused) |
| Exception | throw an exception of class |

Table

Ruletype key  
To identify ruletypes each ruletype has a unique key.

In table 3 an overview is given of the currently supported violationtype keys.

|  |  |
| --- | --- |
| KEY | RULETYPE |
| IsNotAllowedToUse | Is not allowed to use |
| IsAllowedToUse | Is allowed to use |
| IsOnlyAllowedToUse | Is only allowed to use |
| IsOnlyModuleAllowedToUse | Is the only module allowed to use |
| MustUse | Must use |
| IsNotAllowedToMakeBackCall | Is not allowed to use modules in higher layer (back call rule) |
| IsNotAllowedToMakeSkipCall | Is not allowed to use modules in a not directly lower layers (skip call rule) |
| InterfaceConvention | Interface convention |
| SuperClassInheritanceConvention | Superclass Inheritence convention |

Table

Exception rules  
Some ruletypes can have exceptions. Exception rules are most often counterparts of the used ruletype. For example in case of the ‘Is Not Allowed To Use’, the ‘Is Allowed To Use’ can be used as exception rule.

Violation

When the source code does not meet the defined architecture, violations will be returned, which will show you the problems in your architecture.

Configuration  
Parts of the graphical user interface (GUI) can be configured in the validate component:

* Severity: gives an indication what the severity is of the violation. Custom severities can be made. This means you can define the impact of a specific ruletype, and define its color, to be shown in the gui.
  + Default Severity: the default values of severities, defined by the tool itselve.
  + Custom Severity: is a severity that is created by the user in the Configuration UI of the valiate component.
* Active violationtypes per ruletype: which violationtypes should be enabled by default in the filter of the define component?
* Default and Allowed rules: rules can be configured per module. For each module there are preconfigured default and allowed rules. The allowed rules are static (based on a the module type), but the default rules can be configured, according to the users wishes.

# Functionality

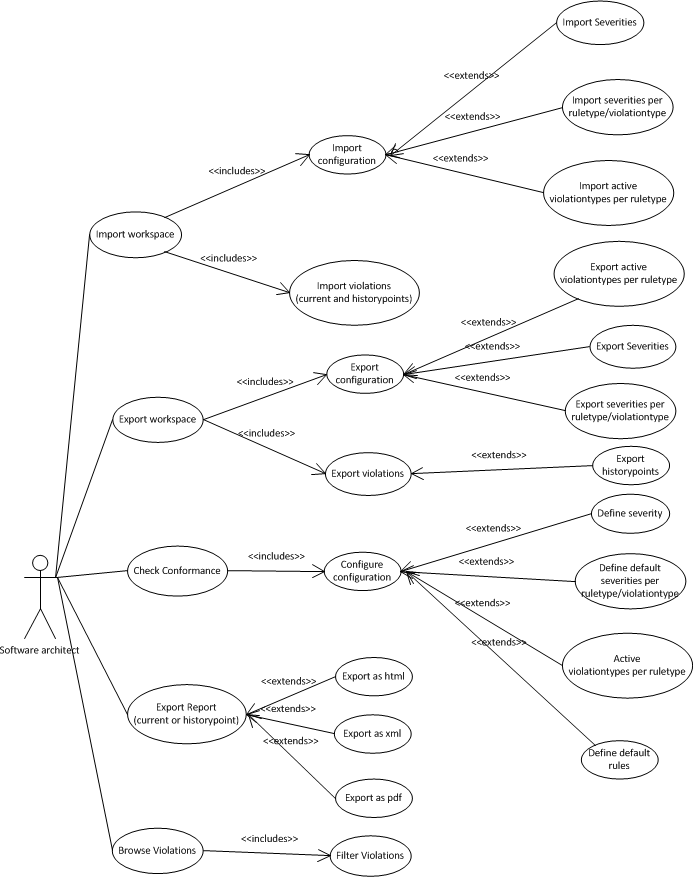
In this chapter all main functionalities are covered by using the use case diagram which has been made during the development process. This use case diagram gives an overview of the functionality that has been implemented in the project thus far.

The most significant/important use cases covered in this chapter are:

* Check Conformance;
* Configure configuration;
* Export Report;
* Export workspace;
* and Import workspace.

## Use case diagram

In figure 1 all the realized use cases are shown. As described above only the most significant use cases will be treated/explained in the next paragraphs.



Figure

## Check Conformance

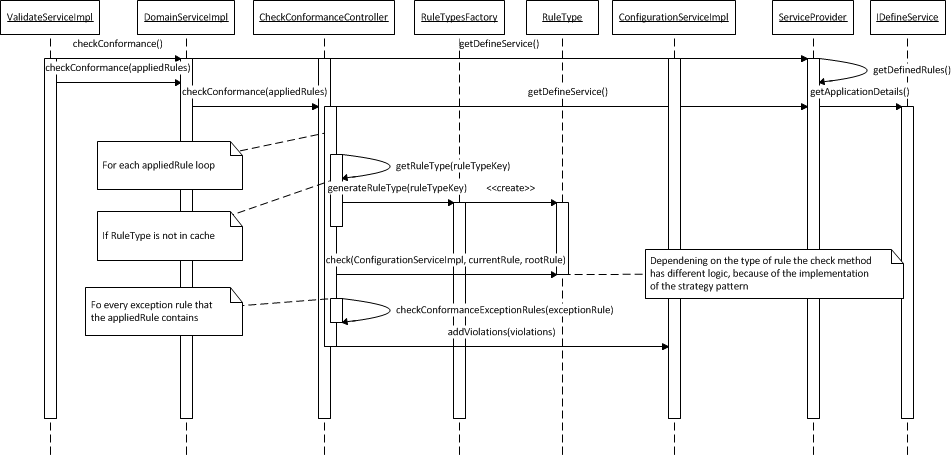
Check conformance is the most important use case in the validate component.

Check conformance validates the rules defined in the define component against the scanned dependencies from the analyse component.

### Textual description

|  |  |  |
| --- | --- | --- |
| NAME | CHECK CONFORMANCE | |
| Goal | check if there are violations against the defined rules in combination with the scanned dependencies. | |
| Description | The system retreives all defined rules. For each rule, the system checks if there is a violation. In case of a violation, the system will then check if there is an active exception rule, which – if the exception rule is passed – will then filter out the violation. | |
| Actor | Software architect, developer. | |
| Constraints | No false negatives are allowed.  No false positives are allowed.  All true violations must be reported.  A violation may only be reported once. | |
| Precondition | The actor has defined the architecture and the system has analysed the application. The analyse component has called the check conformance method from the validate component. | |
| Main Scenario | Actor | System |
|  |  | S1 The method check conformance is called |
|  |  | S2 The system checks for the applied rules and configuration where to make the violations. |
| Result | The current violations are updated | |

### Check conformance mechanism



Figure

In figure 2 (previous page) the mechanism of the check conformance functionality is shown. When the method checkConformance is called all defined rules will be fetched from the define component. With this data the service layer will be called and the layer then calls the method checkConformance(appliedRules) in class *husacct.validate.domain.check.CheckConformanceController.* This class is responsible for checking the conformance of the architecture using the defined rules from the define component.

Because every ruletype has different logic, the strategy pattern is used. When the method check(ConfigurationServiceImpl, currentRule, rootRule) is called the implementation of the check method will be executed of the RuleType instance class.

For more information on the getApplicationDetails() command from the IDefineService, please use the Define system documentation.

#### EXCEPTION RULES MECHANISM

After the method check is called a rule can also have exception rules, these rules are checked in the method checkConformanceExceptionRules(exceptionRule) (see figure 2). The parameter of this method is a RuleDTO. A RuleDTO contains a ‘from’ and a ‘to’ parameter, these parameters contains the logical modules (and these logical modules contains the mapping of the classes). If an exception is created the classes/packages of the exceptionrule must be filtered out the rootrule/mainrule, so they will not conflict with the exception rule. But because there is a possibility to define exception rules with a specific violationtype(s), a list of violationtypes will be created per class/package during filtering. An example of this scenario: ‘Is Not Allowed To Use’ rule is not allowed to use component c1, except for (‘Is Allwed To Use’ rule) class *a.b.C* only with a constructor call. All classes of component must be validated with the provided keys, only for class *a.b.C* violationtype must be filtered out in the mainrule/rootrule (‘Is Not Allowed To Use’ rule).

To make this exception mechanism work a filter method must be called. This will happen in the implementation of the rule, because dependening on the rule classes/packages must be filtered with on specific parameter (‘from’, ‘to’ or both). The following possibilities are provided.

* Filter only the ‘from’ parameter, the ‘from’ parameter of the rootrule/mainrule will be filtered against the ‘from’ parameter of the exceptionrule(s).
* Filter only the ‘to’ parameter, the ‘to parameter of the rootrule/mainrule will be filtered against the ‘to parameter of the exceptionrule(s).
* Filter both ‘from’ and ‘to’.

The classes that are responsible for this are placed in package *husacct.validate.domain.check.util*. The most significant classes are *husacct.validate.domain.check.util.CheckConformanceUtilClass* and *husacct.validate.domain.check.util.CheckConformanceUtilPackage*. The class *CheckConformanceUtilClass* can filter classes and maps these classes to the logical defined modules (from the define component). Because the path of a class can be exists multiple times in the hierarchical structure, the path of the class in the lowest hierarchical level must be mapped to the associated lowest logical module. (See for more information about the hierarchical level and how this is sent in the RuleDTOs see the documentation of define)

The class *CheckConformanceUtilackage* can filter packages and maps these classes to the logical defined modules (from the define component). The functionality of both class are almost identical, but because the class *CheckConformanceUtilClass* is most used this class will be explained.  
  
The class *husacct.validate.domain.check.util.CheckConformanceUtilClass* provides the following methods:

* filterClassesFrom(RuleDTO rule), only filters the ‘from’ parameter;
* filterClassesFrom(RuleDTO rule), only filters the ‘to’ parameter;
* filterClasses(RuleDTO rule), filter both ‘from’ and ‘to’ parameters.

The common methods that are used by *CheckConformanceUtilClass* and *CheckConformanceUtilPackage* are placed in the abstract class *husacct.validate.domain.check.util.CheckConformanceUtil*, the method removeExceptionPaths in class *CheckConformanceUtil* will filter the classes/packages based on the violationtypes. When all the violationtypes are filtered away, the rule will be removed from the classpaths that will be validated.

The filter methods will return a *husacct.validate.domain.validation.internal\_transfer\_objects.Mappings* class that contains a ‘from’ and ‘to’ attribute. These attributes contains a list of *husacct.validate.domain.validation.iternal\_tranfer\_objects.Mapping* objects and contains the same information as the RuleDTO, but the violationtypes/classes are filtered and a mapping with a logical module is made.

### Callback to the code viewer

In the Violation overview UI, a button is implemented to view the source code in a fancy code viewer.

When a violation is selected from the list, the following variables will be set; sourceClassPath and sourceLine, sourceSeverity. Then, when the “Show violation source” button is selected, a call will be made to open the code viewer using the following function:

ServiceProvider**.**getInstance**().**getControlService**().**displayErrorInFile**(**sourceClassPath**,** sourceLine**,** sourceSeverity**);**

The implementation of the code viewer is in the control package.

## Configure configuration

The configuration of the system contains the following information/possibilities, that is contained in different repositories in package *husacct.validate.*:

* Define Severity: creating a custom severity or change the name of colour of an existing severity. Contains a list of all the severities in this repository.
* Active violationtypes per ruletype: possibility to change which violationtypes should be enabled by default in the filter of the define component. Contains a list that stores the information of violationtypes which should be enabled by default in this repository.
* Severity per ruletype/violationtype: the possibility to give a violation- or ruletype a different severity than the default severity. Contains a list of all keys and its associated severities.
* History points: a repository where all the history points of violations are stored in the memory.
* Current violations: a list of all current violations, when the method checkConformance is executed the results are stored in this repository.
* Default rules: configure which rules should be added by default, when adding a module in the define interface.

The service layer that provides methods to modify or get the data is the class *husacct.validate.domain.configuration.ConfigurationServiceImpl*. All other classes of package *husacct.validate.domain.configuration* have visibility default, so only through the service class *ConfigurationServiceImpl* methods of these classes can be reached. *ConfigurationServiceImpl* class is only instantiated when *husacct.validate.ValidateServiceImpl* is. After this, the reference to the *ConfigurationServiceImpl*-objectis passed in different methods. Because the validate component makes use of only one instance of the *ConfigurationServiceImpl,* changes in the configuration are directly visible in the tool. This decision was made to pass the reference of the object as opposed to implementing a singleton.

### Configuration data repositories SeverityPerTypeRepository

The way in which data is stored in the repository *husacct.validate.domain.configuration.SeverityPerTypeRepository* might cause some confusion. The data is saved in an attribute of type HashMap<String, HashMap<String, Severity>>.

ruletype key or violationtype key

Programming language

HashMap<String, HashMap<String, Severity>>

*Above is explained what data is saved in each part of the map.*

## Export report

When checkConformance is called the detected violations will be saved in *husacct.validate.domain.configuration.ViolationRepository* after invoking the method addViolations of *husacct.validate.domain.configutation.ConfigurationServiceImpl*. When a history point is created, the violations in the *ViolationRepository* will be copied to *husacct.validate.domain.configuration.ViolationHistoryRepository*. A report can be generated from either the current violations or a saved history point. However, the functionality of creating a report based on a history point is currently not possible through the GUI.



Figure

In figure 3 the flow of execution is shown of creating a report of the current violations with the method exportViolations(file, extensionType). The flow of execution of creating a report of a history point is shown with the method exportViolations(file, extensiontype, date). Because the UI is created and maintained by the control component, this part is modeled in the sequence diagram as ‘ExportViolationsUI’.

**Classes covered:**

* IValidateService (and ValidateServiceImpl)
* IReportService (and ReportServiceImpl)
* ReportFactory
* ReportWriter (Abstract Class)
* ExtensionTypes
* FileNotAccessibleException
* UnknownStorageTypeException
* ReportException
* Report
* StatisticsImage.

The reports are created in the validate component. That’s why the ValidateServiceImpl has the method exportViolations. There are two different exportViolations, they both accept two parameters: a *java.io.File* which defines the storage location of the report and a *String* defining the extension type. Above the two parameters mentioned, the second method also accepts a *java.util.Calendar* parameter, which refers to a specific history point.

The *ValidateServiceImpl* will pass the methods to the *ReportServiceImpl* class. First the permissions are check, to make sure the given destination is writable. If the destination is inaccesible, a *FileNotAccessibleException* exception is thrown. The method will delegate the parameters to the *ReportFactory*. If a calendar parameter is used, the violation history will obtained from the *TaskServiceImpl*. The violations and severities will then be obtained from the violation history point and passed on to the *ReportFactory*. In other cases, where no calendar object is used, the severities will be obtained from the *TaskServiceImpl* with the method “getAllViolations()” and “getAllSeverities()” for the latest violations and severties.

The *ReportFactory* has one goal, to instantiate a *ReportWriter* object and call the method “createReport()” on the instantiation. The report will be created based on a *Report* object. The only parameters that are missing for the *Report* are the application name and application version. These are obtained from the *DefineService*. Next an if statement will check the extension types to determine the implementation of the *ReportWriter*. In case the implementation cannot be found an exception will be thrown by the system. If the implementation is found the createReport method will be called and the report will be created. The kind of report is defined by the ReportWriter implementation. If any error occurs during the export, the system will throw a *ReportException*.

The factory makes use of the *ExtensionTypes* class, which contains the *ExtensionType* enumeration. The if statement checks the given extension type parameters against the values in the *ExtensionType* enum. The *ExtensionTypes* class provides the method getExtensionTypes returing an array of all available extension types.

### Adding a new Reporttype

To add a new reportype the following steps need to be taken, in this subparagraph we will take a new excel reporttype as an example. Most classes will remain unaffected by this change; the classes that must change are the husacct.validate.task.extensiontypes.*ExtensionTypes* enumeration, husacct.validate.report.*ExportReportFactory* and a new ‘ReportWriter’.

The first step is to add an extra constant to the *ExtensionTypes* enumeration called EXCEL. This enumeration needs to have the extension as parameters in the constructor, in this case it will be ‘xslx’. The result: *PDF*("pdf"),

*HTML*("html"),

*XML*("xml"),

*EXCEL*("xlsx");

Next the *ExportReportFactory* needs to be changed. The if-else statement needs to be extended to also include a check whether the extensiontype was a excel type. If this is the case, the ‘ReportWriter’ will be an ‘ExcelReportWriter’. Of course the ‘ExcelReportWriter’ doesn’t exist yet so the next step is to create a new ‘ExcelReportWriter’ class that will contain the logic to convert the data to excel.

This part is of course entirely based on what kind of report will be going to be created; all the different reportwriter that are located in *husacct.validate.task.reportwriter* none of them are the same. They do however have in common that they all show the violations ,and the graphic reports even show a chart of the severities. Besides that they all show statistics about the violations that have occurred.

The new ‘ReportWriter’must extend *husacct.validate.task.report.writer.ReportWriter* and can make use of the violations and severities which are available through the report attribute of the superclass (*ReportWriter*). There is a path available to the statistics image in the Report. This path is simply the directory to where the file was going to be stored plus the files name. The husacct.validate.task.report.writer.*PDFReportWriter* for example makes use of it by putting it in its *pdf* and the deleting the image so that no unnecessary files will be left on the computer. The *husacct.validate.domain.validation.report* . The *ReportWriter* provides the following attributes that can be used by a new ‘ReportWriter’:

* path: the path of the folder where the file must be stored;
* fileName: the name of the new file;
* and the extensionType: the extensiontype of the file.

## Export workspace

Before closing HUSACCT, one might want to to save his configuration, history points and current violations. This is possible by exporting the current workspace (or clicking ‘Save and Exit’ while exiting the application). The mechanism of exporting a workspace is maintained by the control component, because the interface *husacct.validate.IValidateService* extends interface *husacct.common.savechain.ISaveable* a savechain is started by the control component. With the method exportWorkspaceData() the process for exporting a workspace will be triggered. For more information about the savechain see the documentation of the control component.

Figure

Explanation of figure 4; because the flow of *ISaveable* is controlled by the control component the sequence diagram will start where the sequence diagram of the control component ends. The class *husacct.validate.task.export.ExportController* is responsible for the exporting process. First the *ExportController* asks *husacct.validate.domain.configuration.ConfigurationServiceImpl* for all required configuration information. This information will be delegated to exportclasses that have the knowledge/logic to transform the given datatypes/attributes to XML. All the XML objects are merged in the *ExportController* and will be returned to the control component.

## Import workspace

In the previous paragraph is explained how data of a workspace can be exported. This data can obviously also be imported.

The mechanism of importing a workspace is maintained by the control component, because the interface *husacct.validate.IValidateService* extends interface *husacct.common.savechain.ISaveable* a savechain is started by the control component. With the method loadWorkspacedata(element) in *husacct.validate.ValidateServiceImpl* the importing process is triggered. For more information about the savechain see the documentation of the control component.



Figure

Explanation of figure 5; because the flow of *ISaveable* is controlled by the control component the sequence diagram will start where the sequence diagram of the control component ends. The class *husacct.validate.task.fetch.ImportController* is responsible for the importing process. The tag that is exported is extracted from the XML by the control component. This XML is exactly the same as the result of the XML that was returned during the exporting of the workspace. Tags will be identified and extracted and delegated to the class with the knowledge to extract this information into objects. These objects will be returned to the *ImportController*. The *ImportController* takes care of placing the objects in the right repository, this will be done with the methods provided by the *ConfigurationServiceImpl*.

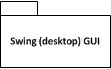
# Decisions and justifications

The decisions and justifications that were made during the development process.

|  |  |
| --- | --- |
| DECISION | JUSTIFICATION |
| To change the language in the application we will use ResourceBundels in Java. | The User Interface must support the English language, but other languages can be set by the user. |
| To identify the different ruletypes/violatypes in another language we will use keywords. Using keywords in combination with resourcebundles makes it easy to switch between languages. | It must be easy to add new ruletypes and violationtypes. |
| To make sure the right key is typed we will use enumerations | It must be easy to add new ruletypes and violationtypes. |
| To decide which exporter to use, we are going to implement the factory pattern. So new reporttypes can easily be added. | Other reporttypes must easily be added. |
| For check conformance all the different ruletypes must extend a method with an abstract method, so the strategy pattern can be applied. | It must be easy to add new ruletypes. |
| The configuration of which violationtypes can occur by which ruletypes is defined in the source code. This because this will hardly change | Configuration in xml or properties files are error-prone, because users can edit them easily. For things that will hardly change add these configurations in the source code. |
| To export the configuration we will take into account that in the future a database can be added, initially we will only focus on exporting in the xml file format | It must be easy to add new import- and export formats for exporting a workspace. |
| For performance reasons we created a Util class, which contains a few static methods, used throughout the domain. For example *husacct.validate.domain.check.util* | Validation must be performed as fast and accurate as possible. |
| For loading ruletypes and violationtypes dynamic classloading will be used | It must be easy to add new ruletypes and violationtypes. |
| To prevent that classloading is executed every time, the Lazy Load pattern will be applied | Apply caching were possible if the (generated) data is always the same. |
| For major performance reasons we decided to not use a singleton, but to pass a configuration service as a parameter in order to still keep one instance. | Avoid a Singleton where possible. |

# Software partioning

In figure 6 the software partitioning with architectural rules is shown. In figure 6 the packages can be seen as subcomponents. The mapping of physical classes/packages will be given to the defined subsystems in table 4.

****

No skipp-call is allowed except for class ConfigurationServiceImpl

Is not allowed to use the presentation layer except for a notify from an observer

Presentation

Task

Domain

Figure

Is not allowed to use Swing (desktop) GUI except for class GuiController

Other rules

* All the classes of package *husacct.validate.domain.configuration* have visibility package except for class *husacct.validate.domain.configuration.ConfigurationServiceImpl*.
* All the subpackages of package *husacct.validate.domain.validation.ruletype* must extend abstract class *husacct.validate.domain.validation.rule.RuleType*
* All enumeration that contain violationtypekeys constants of a language must implement *husacct.validate.domain.validation.violationtype.IViolationType*.
* All classes in package *husacct.validate.task.report.writer* must extend class *husacct.validate.task.writer.ReportWriter*, except for class *husacct.validate.task.writer.ReportWriter*. (because the class is in the same folder as in the target).
* Class *husacct.validate.domain.factory.RuleTypesFactory* is the only module allowed to use *husacct.validate.domain.factory.RuleTypesFactory*.

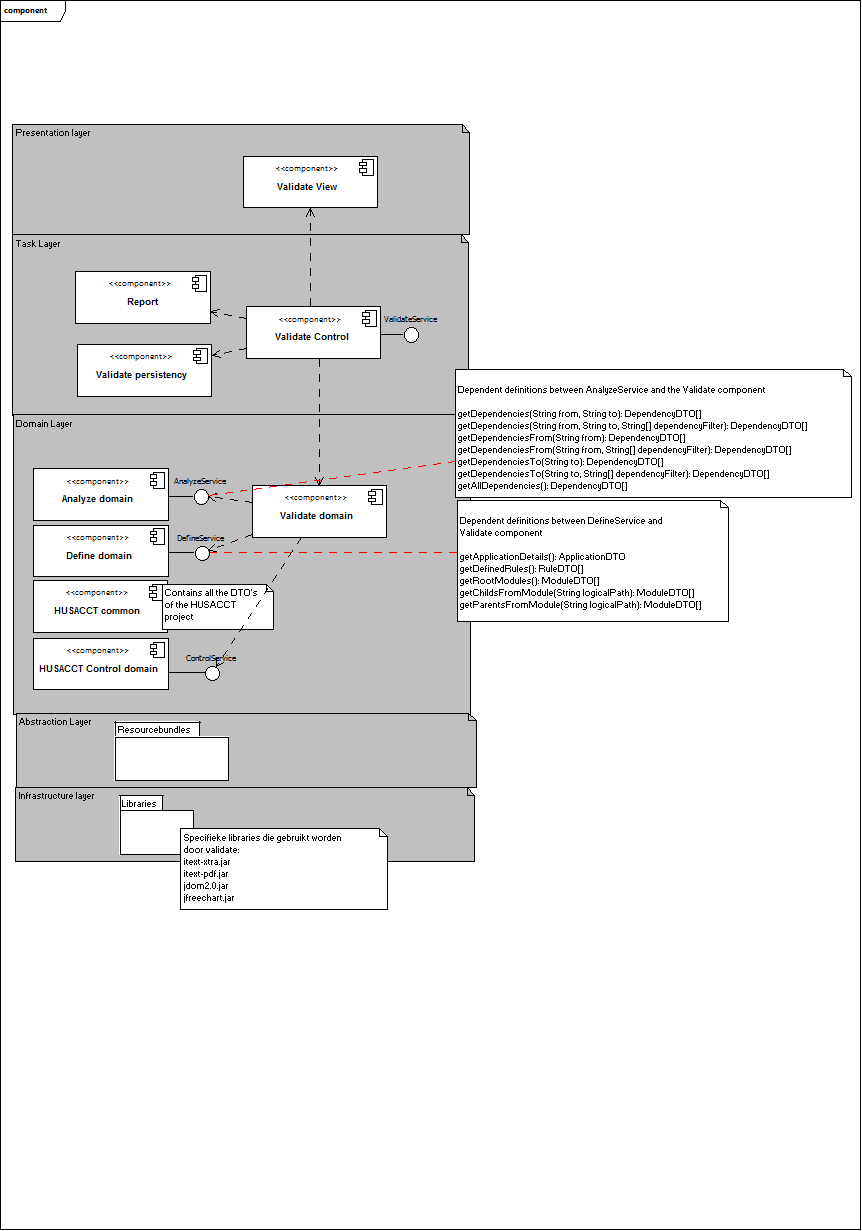
## Mapping of the physical classes to the software partitioning

|  |  |
| --- | --- |
| SUBSYSTEM | PACKAGE/CLASS |
| Swing (desktop) GUI | All underlying packages and classes of package: *husacct.validate.presentation* |
| Report | All underlying packages and classes of package: *husacct.validate.task.report* and the classes *husacct.validate.task.IReportService*  *husacct.validate.task.ReportServiceImpl* |
| Import XML | All underlying packages and classes of package: *husacct.validate.task.fetch* |
| Filter | Class *husacct.validate.task.filter.FilterController* |
| Export XML | All underlying packages and classes of package: *husacct.validate.task.export* |
| ValidateServiceImpl | Contains the classes:  *husacct.validate.IValidateService*  *husacct.validate.ValidateServiceImpl* |
| Exceptions | All the underlying packages and classes of package: *husacct.validate.domain.exception* |
| Configuration | All the underlying packages and classes of package: *husacct.validate.domain.configuration* |
| Assembler | All the underlying packages and classes of package:  *husacct.validate.domain.assembler* |
| Validation (domain/data classes) | All the underlying packages and classes of packages:  *husacct.validate.domain.validation* |
| Factory | All the underlying packages and classes of packages:  *husacct.validate.domain.factory* |
| Check Conformance | All the underlying packages and classes of packages:  *husacct.validate.domain.check* |

Table

The classes *husacct.validate.task.TaskServiceImpl* and *husacct.validate.domain.DomainServiceImpl* are placed in the root of the associated layer.

## Relations with other components

In figure 7 the relation with other components is shown.

Figure

# Dependency types

In HUSACCT there are 6 general dependency types, which will be explained in this chapter.

To futher explain the dependency types, they will be accompanied with examples.

## Call

The call dependency type is used when new instances are made of classes or when methods or constructors are called.

### Example

**public** **class** SocialNetwork {

**public** SocialNetwork(){

}

}

**public** **class** Account {

**public** Account(){

**new** SocialNetwork();

}

}

## Access

The access dependency is used for *Instance variables, Instance variable constants, Class variables, Class variable constants* and *Enumerations.*

## Inheritance

The inheritence dependency is given, when a class is extended or inherited by another (abstract) class or interface.

### Example

Extends abstract class

**public** **abstract** **class** ASocialMedia {

}

**public** **class** SchoolGroup **extends** AsocialMedia {

}

Implements interface

**public** **interface** IASocialMedia {

}

**public** **class** IdenticaList **implements** IASocialMedia {

}

## Declaration

One of the most common ways for Access dependencies to show up, are with variable instances and as parameters of a method. As a third type which is detected as Access dependency, is the return type.

The remaining types which fit the requirements for an Access dependency are *Exceptions, Type casts* and *Object references.*

### Example

Instance variable

**public** **class** SocialNetworkInfo {

**public** Timeline time;

}

Type cast

**public** **class** SocialNetwork {

**private** String type = "default";

**public** String getSocialNetworkType(){

**return** this.type;

}

}

**public** **class** Followers {

**private** Object socialnetwork;

**public** Followers(){

System.*out*.println((SocialNetwork) socialnetwork.getSocialNetworkType());

}

}

Exception

**public** **class** ASocialMediaException **extends** Exception{

**public** ASocialMediaException(String message){

**super**(message);

}

}

**public** **class** PublicMessage {

**public** PublicMessage(){

**try** {

getMessages();

} **catch** (ASocialMediaException e) {

e.printStackTrace();

}

}

**public** String[] getMessages() **throws** ASocialMediaException{

**throw** **new** ASocialMediaException("No Messages");

}

}

## Annotation

Annotations are detected also detected in the code, and will be given the similarly named dependency type.

### Example

**public** **@interface** SocialTagAnnotation {

**public** String[] tags();

}

@SocialTagAnnotation(tags = { "three","different","tags" })

**public** **class** Trends {

}

## Import

Import is used, when an import is detected in the source code.

### Example

**import** husacct.validate.domain.DomainServiceImpl;

# RuleViolation TypeKeys

## Rulekeys

|  |  |
| --- | --- |
| Key | Rule |
| IsNotAllowedToUse | Is not allowed to use |
| IsAllowedToUse | Is allowed to use |
| IsOnlyAllowedToUse | Is only allowed to use |
| IsOnlyModuleAllowedToUse | Is the only module allowed to use |
| MustUse | Must use |
| BackCall | Is not allowed to use modules in higher layer (back call rule) |
| SkipCall | Is not allowed to use modules in a not directly lower layers (skip call rule) |
| InterfaceConvention | Interface convention |
| SuperClassInheritenceConvention | Superclass Inheritence convention |
| CyclesBetweenModules | Cycles/loops between modules |

## Violationtypekeys Java

|  |  |
| --- | --- |
| Key | Violationtype |
| InvocMethod | Invocation of a method |
| InvocConstructor | Invocation of a constructor |
| AccessPropertyOrField | Access of a property or field |
| ExtendsConcrete | Extending an abstract class |
| ExtendsAbstract | Extending a concrete class |
| ExtendsLibrary | Extending a library |
| ExtendsInterface | Extending an interface |
| Implements | Implements |
| Declaration | Declaration |
| Annotation | Annotation |
| Import | Import |
| Exception | Exceptions |

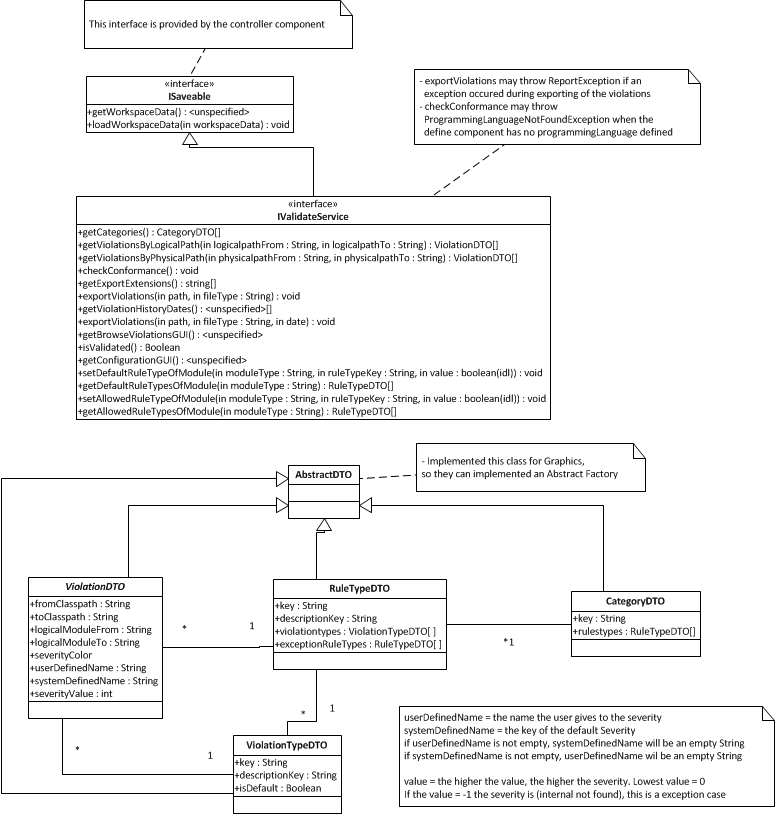
## Violationtypekeys C#

|  |  |
| --- | --- |
| Key | Violationtype |
| InvocMethod | Invocation of a method |
| InvocConstructor | Invocation of a constructor |
| AccessPropertyOrField | Access of a property or field |
| Extends | Extending a class |
| Implements | Implements |
| Declaration | Declaration |
| Annotation | Annotation |
| Import | Import |
| Exception | Exceptions |
| Delegate | Delegate |

# Subsystem specification

## Provided sevices of the validate component

To ensure that the components can communicate with eachother, each component will provide an interface. The Serviceprovider (see documentation of the control component) is a class that is responsible for the communication to other components through the provided interfaces of the specific component. Because the husacct.*Serviceprovider* is accessible in every component it is easy to call interfaces from another component.



Figure

### Provided methods

Figure 8 contains the interface *husacct.validate.IValidateService*, this interface uses the service provider to provide the methods in this interface for other services:

* getCategories(): returns all the categories of the rules, these categories contains the rules and these rules contain the possible violationtypes;
* getViolationByLogicalPath(logicalPathFrom, logicalPathTo): returns all the violations that cope with the given arguments. This method is provided for the graphics component to visualize the violations against the (logically) defined architecture;
* getViolationsByPhysicalPath(physicalPathFrom, physicalPathTo): return all the violations that cope with the given arguments. This method is provided for the graphics component to visualize the violations against the physical classes (with their dependencies);
* checkConformance(): starts the mechanism to validate the rules that are defined in the define component against the dependencies from the scanned source code from the analyse component;
* getExportExtensions(): return all extensions which are supported to export violations;
* exportViolations(path, fileType): saves the current violations to a report in the specified fileType (the possible fileTypes are provided by the method getExportExtensions).;
* getViolationHistoryDates(): return all the dates of the history points in the system ;
* exportViolations(path, fileType, date) saves the violations from a given historyPoint (possible dates are provided by the method getViolationHistoryDates) to a report in the specified fileType (possible fileTypes are provided by the method getExportExtensions);
* getBrowseViolationsGUI(): returns the screen that gives the user an overview of all the violations, with the possibility of filtering;
* isValidated(): returns true or false if the method checkConformance is called. This method is provided for the control component to keep track of the (work)flow of the application. See for more information about the workflow of the application the documentation of the control component;
* getConfigurationGUI(): returns the screen that gives the user the possibility to configure the validate component;
* getWorkspaceData(): returns a JDOM (library that is used) XML object that contains all the data that need to be saved in an xml file;
* loadWorkspaceData(element): loads the xml that was exported in the method getWorkspaceData().
* setAllowedRuleTypeOfModule(String moduleType, String ruleTypeKey, boolean value), sets the allowedRules based on a given moduleType and ruleTypeKey.
* getAllowedRuleTypesOfModule(String type), gets the allowed rules based on a given module type.
* setDefaultRuleTypeOfModule(String moduleType, String ruleTypeKey, boolean value), sets the default rules based on a given moduleType and ruleTypeKey.
* getDefaultRuleTypesOfModule(String type), gets the default rules based on a module type.

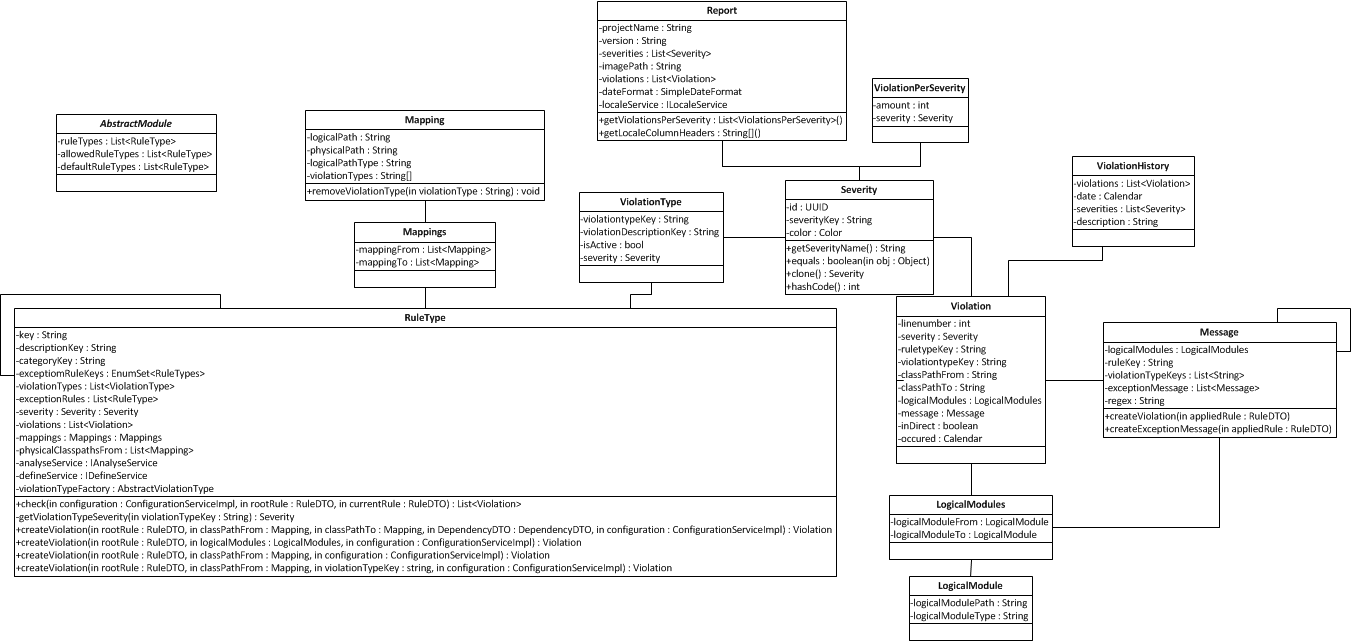
### Provided Data Transfer Objects

The information that is exchanged between the components are wrapped in Data Transfer Objects (DTO). These DTOs only contain the information of a model, but no (business) logic.

The implementing the Data Transfer Objects helps in the decoupling between components. The validate component uses the following DTOs to exchange data to other services:

* CategoryDTO: contains information about the category of a ruletype and also a list of RuleTypeDTOs that belong to this category;
* RuleTypeDTO: contains the information about a ruletype and also a list of ViolationTypeDTOs that can occur during validating the specified rule;
* ViolationTypeDTO: contains information about the violationtype and if the violationType should be enabled in the filter of the define component (for more information see the component of the define component);
* and the ViolationDTO contains all the information of a violation, including a reference to the RuleTypeDTO of the defined rule where the violation is created. Also an reference to ViolationTypeDTO of violationtype of the violation.

## Domain model

In figure 9 the (most important) domain classes are shown. The classes are located in subpackages of or in the root of *husacct.validate.domain.validation*.

Figure

## Starting up/resetting the component

Before the application is started the husacct.*ServiceProvider* (see system documentation of control) is responsible that only one instance of the ValidateService is instantiated. If the ValidateService is called some classes will be instantiated and passed as arguments to other classes to prevent that the classes are instantiated multiple times. This decision was made because otherwise there would be a lot of singletons in the validate component, which hurts the performance.

Resetting the ValidateService can be done by creating a new instance of the ValidateService.

Starting up (and thus resetting) will be illustrated in figure 10.



Figure

ValidateServiceImpl is instantiating other classes when it is created. In the entire validate component all classes that end with ‘ServiceImpl’ are only instantiated once when the ValidateServiceImpl is called. The ‘ServiceImpl’ are service layers for the corresponding layer where the class belongs. Service layers are implemented to get more control over the flow of the application and to create a boundry for the underlying classes in the specific (sub)component.

The sequence diagram in figure 4 only shows the classes that are called from the *ValidateServiceImpl* and not from the calls when a specific service is instantiated. These calls/methods/mechanism will be covered when the most significant use cases will be treated.

The services have to following responsibilities:

* *ConfigurationServiceImpl*: responsible for providing information/methods about the current configuration of the validate component, also provides methods to change the configuration data;
* *DomainServiceImpl*: responsible for providing access for the significant use cases (for example checkConformance) and for reusable information from the domain layer;
* *TaskServiceImpl*: responsible for providing methods that the presentation layer can invoke. This service is created to provide the possibility in the future to create an new (type of) GUI, for example a webbased GUI;
* *ReportServiceImpl*: responsible for providing methods for creating reports of the current violations or the history points that are created;
* *GuiController*: responsible for instantiating the desktop (Java Swing) GUI. This controller is created to instantiate the desktop GUI when there’s a call. This controller is implemented because sometimes a GUI is not used, for example in the Maven plugin (see documentation of the control component). The advantage of this behaviour, is that the GUI component will only be instantiated when the GUI is actually used, this means that HUSACCT takes up less resources in a graphical-less enviroment.

## Creating/generating ruletypes and violationtypes objects

One of the responsibilities of the validate component is to maintain the different ruletypes and provide a mechanism that is flexible so new ruletypes can easily be added. In the next subparagraph some examples will be given of some ruletypes in combination with violationtypes. Next there will be an explanation of how the mechanism works for violationtypes. At last there will be an explanation of how the mechanism works for ruletypes.

### Violationtypes

As described in the previous subparagraph violationtypes belong to one or more ruletypes and dependening on the programminglanguage of the scanned source code violationtypes can differ.

To get the violationtypes of a rule you must instantiate the class *husacct.validate.domain.factory.ViolationTypeFactory* and call the method getViolationTypeFactory (one of the two overloaded methods) this will return a *husacct.validate.domain.factory.AbstractViolationType*, which is an abstract implementation of a concrete class that contains the combinations of violationtypes which are possible per ruletype for a specific programming language. Every violationtype has a keyword, with this keyword a violationtype can be identified. All the information about the keys are in the enumarations in package *husacct.validate.domain.validation.violationtype*. For example we take the source of *husacct.validate.domain.validation.violationtype.JavaViolationTypes*: *IMPORT*("Import", ViolationCategories.*DEPENDENCY\_RECOGNITION*, DefaultSeverities.*LOW*) The name of the key is’ Import’, this violationtype belongs in category ‘Dependency Recognition’ and the Severity of the rule is defined as ‘Low’.

### Generating violationtypes with dynamic classloading

The class *husacct.validate.domain.factory.ruletype.ViolationtypeGenerator* is responsible for creating new ViolationType objects. Only the class *husacct.validate.domain.factory.violationtype.AbstractViolationType* is allowed to create a new instance of *ViolationtypeGenerator*.

In figure 11 the mechanism of the generation of a new violationtypes is shown.



Figure

The method getAllViolationTypes expects *husacct.validate.domain.validation.violationtype.IViolationType* as parameter. The enumerations that implement this interface are converted to a List<IViolationType> and are passed as arguments to the method getAllViolationTypes. Inside this method for each IViolationType the class-object of the *IViolationType* will be extracted. Then with reflection the information of the violationtype is extracted through the interface.

The method getAllViolationTypes returns an ‘internal transfer object’ of class *husacct.validate.domain.validation.internaltransferobjects. CategoryKeySeverityDTO* and contains the key of the violationtype and the key of category of the violationtype. The results of the method getAllViolationTypes are cached in *AbstractViolationType* so the violationtypes doesn’t have to be generated every time. The violationtypes are only generated when a new *AbstractViolationType* is instantiated.

### Ruletypes

One of the responsibilities of the validate component is to maintain the different ruletypes and provide a mechanism that is extendable for new ruletypes.

Therefore a mechanism is created with dynamic classloading. All the rules are located in subpackages of *husacct.validate.domain.validation.ruletype.* The name of the subpackages is the categorykey, with this keys, ruletypes are divided into different categories. The enumeration *husacct.validate.domain.validation.ruletype.RuleTypes* contains the information about the keys, for example *IS\_ONLY\_ALLOWED*("IsOnlyAllowedToUse", DefaultSeverities.*LOW*)Defines the keyword of the rule (which must be the name of the class without the word Rule at the end) and the default Severity of the rule. With the keyword every ruletype can be identified and it is possible to get an individual rule instead of a set of rules, this is possible with the method: generateRuleType in *RuleTypesFactory*.

### Generating ruletypes with dynamic classloading

The class *husacct.validate.domain.factory.ruletype.RuleTypesGenerator* is responsible for creating new *husacct.validate.domain.validation.ruletype.RuleType* objects with dynamic class loading. Only the class *husacct.validate.domain.factory.ruletype.RuleTypesFactory* is allowed to create an new instance of the *RuleTypesGenerator* class.

The class *RuleTypesFactory* alsogenerates the exception rules that belong to the specific ruletypes and adds this exception to the specifc ruletype. The exception keys are defined in the class of the rule itself. An example for this behaviour is found in: *husacct.validate.domain.validation.ruletype.legalityofdependency.IsNotAllowedToUseRule*

**public** **class** IsNotAllowedToUseRule **extends** RuleType {

**private** **final** **static** EnumSet<RuleTypes> *exceptionRules* = EnumSet.*of*(RuleTypes.*IS\_ALLOWED*);

the *RuleTypes* enumeration is used to make sure the key of the rule is always the same for the specific rule and is also used as unique identifier for a rule.

In figure 12 the mechanism of creating new ruletypes will be shown.



Figure

The instantiation of the *RuleTypesFactory* class, also triggers the instantation of the *RuleTypesGenerator* class. The class *RuleTypesGenerator* provides two methods generateAllRules that will create all the available ruletypes from the keys that are available in the *RuleTypes*  enumeration. The method generateRules expects an EnumSet with specific values from the *RuleTypes* enumeration.

First the method getRuleClasses is called, this methods adds an extra parameter to the method and sends it to the overloaded method getRuleClasses. This extra parameter is an array where all the possible paths of the packages are stored, based on its different ruletype locatations. The overloaded getRuleClasses loads the classes with the dynamic class loading mechanism. When this results of the loaded class-objects are returned the information is extracted through reflection (methods: getCategoryKey, getRuleKey and getDefaultSeverity). The methods generateRules and generateAllRules will return a HashMap with the ruletype key as key of the HashMap and a *husacct.validate.domain.validation.internaltransferobjects.CategoryKeyClassDTO* as value. *CategoryKeyClassDTO* contains the following information:

* category key of the rule
* class-object of the rule, so a new instance an easily be created with reflection in *RuleTypesFactory*.
* The default severity of the rule.

After these methods are completed the results will be saved in a caching mechanism in the *RuleTypesFactory* so the rules are only generated once.

In case a public method of *RuleTypesFactory* is called, the cache of *RuleTypesFactory* will be searched for the given RuleTypeKey. If the RuleTypeKey is found a new instance of *husacct.validate.domain.factory.AbstractViolationType* is created, which then retrieves the corresponding violationtypes of the rule.

## Adding new ‘types’

The following subparagraphs will describe how to add new ruletypes and violationtypes, and things to be aware of when adding a new programming language.

In the next paragraph will be shown how a new rule can be added. The example we’ll use is the ‘Cycle’ rule, which is of the category ‘dependency limitation’ and will detect cycles between classes. This rule will have 2 violationtypes: ‘ShortCycle’ and ‘LongCycle’. In the last subparagraph we will describe what needs to be changed when HUSACCT is going to support a new programming language, in this case it will be PHP.

### Adding a new ruletype

The first thing that needs to be done is to update the enumeration with all the keys. In class *husacct.validate.domain.validation.ruletype.RuleTypes* we will add the following line:

*CYCLE*("Cycle", DefaultSeverities.*LOW*), The key of the rule will be cycle and if a violation on this rule occurs the severity will be LOW by default.

If the ruletype ‘Cycle’ is a mainruletype (a rule that is not an exception rule en must be sent with to define when they call the current ruletypes) the enumeration that is just created must be added to the enumset of attribute mainRuleTypes in enumeration *RuleTypes*.

All rules are located in the package *husacct.validate.domain.validation.ruletype*. The name of the subpackage defines the keyword of the category of the rule. To add a new category a new package must be added, in this cases it will be: *husacct.validate.domain.validation.ruletype.dependencylimitation.* Because a new category is created this new package is not known in the class *husacct.validate.domain.factory.ruletype.RuleTypesGenerator.* The new path of this package must be added in the array of attribute ruleTypeLocations of class *RuleTypesGenerator*.

The name of the new rule will be ‘CycleRule’, to fit with the namingconvention: keyword + ‘Rule’. After the class *husacct.validate.domain.validation.ruletype.dependencylimitation.CycleRule* has been created the following steps need to be taken for this class:

* Class *husacct.validate.domain.validation.ruletype.RuleType* must be extended
  + Implement the derived constructor
  + Implement the derived check method
* Define which exceptionrules can occur with the ‘Cycle’ ruletype and pass this in the constructor to the superclass (the name of the parameter is ‘exceptionRules’)
  + If the ‘Is Allowed To Use’ rule is an exception type, create an EnumSet with the value of the *RuleTypes* enumeration.
  + If there are no exceptionrules possible, create an empty enumSet (EnumSet.*noneOf*(RuleTypes.**class**);)
* Implement logic for the check method so it can return a List<Violation> objects.

### Adding new violationtypes

In the previous subparagraph the ‘Cycle’ ruletype has been implemented, but this rule didn’t have any violationtypes. Violationtypes will be sent to the define component so the user can choose the filter on specific violationtypes during the checkConformance method.

The first step is to add an extra violationtype category. These categories aren’t used at the moment in the application, but for functionality in the future it might be necessary or helpful. Add the following lines to enumeration *husacct.validate.domain.validation.violationtype.ViolationTypeCategories*

*DEPENDENCY\_LIMITATION*("DependencyLimitation")

The second step that needs to be taken is to create the new violationtypes. Some enumerations in package *husacct.validate.domain.validation.violationtype* implement the interface *husacct.validate.domain.validation.violationtype.IviolationType*. These classes contain the information of the possible violationtypes for a specific programming language. In this case the new violationtypes need to be added for Java and C#. The following lines need to be added in *husacct.validate.domain.validation.violationtype.JavaViolationTypes* and *husacct.validate.domain.validation.violationtype.CSharpViolationTypes*:

*SHORT\_CYCLE*("ShortCycle", ViolationCategories.*DEPENDENCY\_LIMITATION*, DefaultSeverities.*HIGH*),

*LONG\_CYCLE*("LongCycle", ViolationCategories.*DEPENDENCY\_LIMITATION*, DefaultSeverities.*HIGH*),

The next step that needs to be taken is to identify the rule. In *husacct.validate.domain.factory.violation.AbstractViolationType* the following method needs to be added.

**protected** **boolean** isCycle(String ruleTypeKey){

**if** (ruleTypeKey.equals(RuleTypes.*CYCLE*)) {

**return** **true**;

}**else**{

**return** **false**;

}

}

For all available languages the method createViolationTypesByRule needs to be changed so the language can identify the ruletype. In this case the following else-if statement will be added in *husacct.validate.domain.factory.violationtype.JavaViolationTypeFactory*

**else** **if** (isCycle(ruleTypeKey)) {

**return** generateViolationTypes(ruleTypeKey, EnumSet.*of*(JavaViolationTypes.*SHORT\_CYCLE*, JavaViolationTypes.*LONG\_CYCLE*,));

}

This step should be repeated for every *husacct.validate.domain.factory.violationtype.****[programminglanguage name]****ViolationTypeFactory*.

### Adding a new programming language

When a new programming language must be supported some changes must be made in the validate component. In this subparagraph we will describe how to support PHP.

First a new enumeration must be created that will contain the information about the possible violationtypes supported for the new programming language. This enumeration will be placed in *husacct.validate.domain.validation.violationtype*. In the case of PHP the name of the enumeration will be *PhpViolationTypes* and will implement interface *husacct.validate.domain.validation.violationtype.IViolationType*. The structure of PhpViolationTypes should look like this

**public** **enum** PhpViolationTypes **implements** IViolationType{

*SHORT\_CYCLE*("ShortCycle", ViolationCategories.*DEPENDENCY\_LIMITATION*, DefaultSeverities.*HIGH*),

*LONG\_CYCLE*("LongCycle", ViolationCategories.*DEPENDENCY\_LIMITATION*, DefaultSeverities.*HIGH*);

//MORE VIOLATIONTYPES CAN BE ADDED HERE FOR PHP

**private** **final** String key;

**private** **final** DefaultSeverities defaultSeverity;

**private** **final** ViolationCategories violationCategory;

PhpViolationTypes(String value, ViolationCategories violationCategory, DefaultSeverities defaultSeverity){

**this**.key = value;

**this**.defaultSeverity = defaultSeverity;

**this**.violationCategory = violationCategory;

}

@Override

**public** String toString(){

**return** key;

}

@Override

**public** String getCategory() {

**return** violationCategory.toString();

}

@Override

**public** DefaultSeverities getDefaultSeverity() {

**return** defaultSeverity;

}

}

The next step is to create a factory class that can create violationtypes. This class will be created in package *husacct.validate.domain.factory.violationtype*, with (in this case) the name *PhpViolationTypeFactory* this class will extend *husacct.validate.domain.factory.AbstractViolationType*. When implemented the derived methods from *AbstractViolationType* the new class will look like this:

**public** **class** PhpViolationTypesFactory **extends** AbstractViolationType {

PhpViolationTypesFactory(ConfigurationServiceImpl configuration, String languageName) {

**super**(configuration, "PHP");

}

@Override

**public** List<ViolationType> createViolationTypesByRule(String ruleTypeKey) {

**if**(isCycle(ruleTypeKey)){

**return** generateViolationTypes(ruleTypeKey, EnumSet.*of*(JavaViolationTypes.*SHORT\_CYCLE*, JavaViolationTypes.*LONG\_CYCLE*,));

}

**return** Collections.*emptyList*();

}

@Override

List<IViolationType> createViolationTypesMetaData() {

**return** Arrays.*asList*(EnumSet.*allOf*(CSharpViolationTypes.**class**).toArray(**new** IViolationType[]{}));

}

}

In the constructor the superclass is called with the current configuration and the String “PHP”. This name of the programming language must exactly (case-sensitive) match the name of the programming language as it is defined in the analyse component. This is for configuration purposes (see paragraph 2.3).   
You can extend the method createViolationTypesByRule as described in subparagraph 5.5.2)

The last thing needs need to be changed is the class *husacct.validate.domain.factory.violationtype.ViolationTypeFactory,* so when the programming language PHP is scanned the right instance of *AbstractViolationType* is returned. Add the following else-if statement in method getViolationTypeFactory in class *ViolationTypeFactory*.

**else** **if** (language.toLowerCase().equals("php")) {

**return** **new** PhpViolationTypeFactory(configuration);

}

Now PHP-support is succesfully added in the validate component of the HUSACCT tool.

# Testing

Tests are created with JUnit 4 and are available in *husacctest.validate*. This chapter will explain two classes with tests: *husaccttest.validate.ImportExportTest* and *husaccttest.validate.ValidateTest.*

The class *husacct.validate.ValidateTestSuite* runs all the testclasses that are defined in this class.

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

ValidateTest.**class**,

ImportExportTest.**class**

})

Above code is from *ValidateTestSuite* and defines which tests of which classes needs to be executed. Above code will execute the tests of *ValidateTest* and *ImportExportTest*. If an new testclass must be added add this new testclass to above lines of code in *ValidateTestSuite*. For each test an new instance of *husacct.validate.ValidateServiceImpl* is created so for every test the information of the validate component will be resetted.

## ImportExportTest

The testclass *ImportExportTest* will test if the imported and exported workspace information of the validation component is correct. The methods to check if a value from the xml is available in the associated repository are shared by both test: TestImporting and TestExporting.

TestImporting

TestImporting will check if the xml-file located at *husaccttest.validate.testfile.xml* will be loaded correctly and if the imported data is correctly saved to the repositories. The XML-file will be converted to a JDOMObject, with this JDOMObject the key associated with a (configuration) repository will be extracted. Next up, all the values from this repository are obtained and tests will be started to check if the XML-value corresponds with the value(s) from the repository.

TestExporting

TestExporting will check if the data that is currently available in the repository is correctly exported. The key associated with a (configuration) repository is extracted from the returned JDOMObject, these values are then obtained and tests will be start to check if the XML-values corresponds with the values from the repository.

## ValidateTest

The testclass *ValidateTest* will test the methods that the class *husacct.validate.ValidateServiceImpl* and *husacct.validate.IValidateService* provides to other services.

getBrowseViolationsGUI and getConfigurationUI

This test will check if the screens are not equal to null. If the screen is an instance of *javax.swing.JInternalFrame* and if the visibility of the screen is set to false. (This is a convention that is made with the control component).

getExportExtensions

Check if the provided export extensions are equal to the keys that are expected to be given to other services when this method will be called by other components.

exportViolations

*cannot test exporting a report with JUnit. This needs to be done in the GUI of HUSACCT and the output need to be checked manually.*

getCategories

Checks if the categories are the same as the provided categories in which the ruletypes are dived into.

getRuleTypes

Checks if the ruletypes that are implemented are returned

getViolationTypesJavaLanguage, getViolationTypesCSharpLanguage and getViolationTypesNoLanguage

Checks the violationtypes that are associated with the rules, provided different testcases for different languages to check if this mechanism is working correctly. Also implemented a test when language is defined in the define component. (The ruletypes must be returned, but o violationtypes should be returned)

isValidatedBeforeValidation and isValidatedAfterValidation

Checks if the right boolean is returned for the control component to check when the method isValidated is called

getViolationByLogicalPath and getViolationsByPhysicalPath

*Can’t test this method because of the dependencies with analyse and define.(See explanation next paragraph)*

## Testing the ruletypes

Testing the ruletypes was done with the GUI of the application. Because of the dependencies with the define and analyse component.

Define: there must be rules defined

Analyse: source code must be scanned

Because the importing functionality for the defined rules was not working properly during the development process no JUnit test are written to test the implementation and results of the rules. At the moment it is possible to create JUnit tests to verify the results of the rules.

Tests could be implemented in the following way: define the architecture in the GUI and map the class/packages to modules. Export the architecture, save the file as a XML and store in the folder of: *husaccttest.validate (possibly in a subpackage)*. Scan the source code of an application and check if the result is the same as the defined architecture. Finally call checkConformance and verify the results with JUnit tests.

Although above example is risky, if something changes in the analyse or define component that is related to the methods that are used during the test, the test(s) will fail.

# Future work

In this chapter the future work that could be done in this component will be treated.

## Ideas for improvement

* Maybe make use of the IObservableService instead of own observable implemented in class *husacct.validate.domain.configuration.ConfigurationServiceImpl*, class *husacct.validate.presentation.BrowseViolations* is subscribed on this observable. Drawbacks of this idea the observer will also be notified when something changes in another component.
* Count rows in HTML-report when violations are filtered.
* Need for a better filtering mechanism that also provide the possibility to add for example multiple ‘Is Not Allowed To Use’-rule on the same module, the violations from these rules will be filtered if they are contradictory with another defined rule.
* Refactor classes which have the name Java in them. Some classes are called \*JAVA and some are called Java (so make the capital case consistent).

## Ideas for extension

* Metrics, for the history of violations of a specific timespan.
* Possibility to export filtered violations (from the GUI) to a report.
* Possibility to export multiple history points to a report.