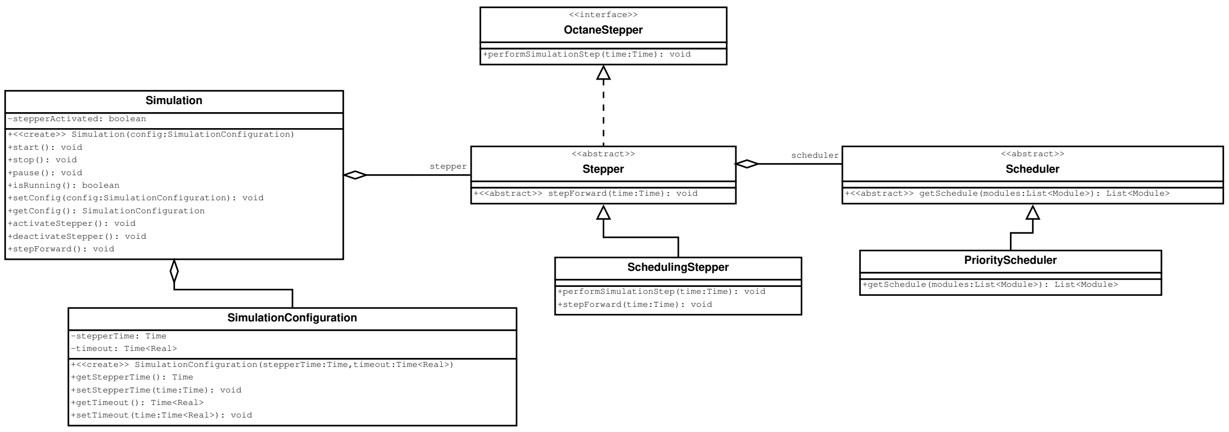
Design
Flowchart-Module for OCTANE

Tobias Bleymehl Tobias Bodmer Diana Burkart
Matthias Lüthy Sebastian Weber
Jonathan Schenkenberger


<<interface>>
OctaneStepper
+performSimulationStep(time:Time): void
Simulation
-stepperActivated: boolean
+<<create>> Simulation(config:SimulationConfiguration)
+start(): void
+stop(): void
+pause(): void
+isRunning(): boolean
+setConfig(config:SimulationConfiguration): void
+getConfig(): SimulationConfiguration
+activateStepper(): void
+deactivateStepper(): void
+stepForward(): void
stepper
<<abstract>>
Stepper
+<<abstract>> stepForward(time:Time): void
scheduler
<<abstract>>
Scheduler
+<<abstract>> getSchedule(modules:List<Module>): List<Module>
SchedulingStepper
+performSimulationStep(time:Time): void
+stepForward(time:Time): void
PriorityScheduler
+getSchedule(modules:List<Module>): List<Module>
SimulationConfiguration
-stepperTime: Time
-timeout: Time<Real>
+<<create>> SimulationConfiguration(stepperTime:Time,timeout:Time<Real>)
+getStepperTime(): Time
+setStepperTime(time:Time): void
+getTimeout(): Time<Real>
+setTimeout(time:Time<Real>): void

December 24, 2017

Contents

1 Introduction 3

2 Design 4

2.1 Global Design . . . 4

2.1.1 Package Diagram . . . 4

2.1.2 Detailed Global Design . . . 5

2.2 Starting the Program . . . 6

2.2.1 Starting without arguments . . . 6

2.2.2 Starting with an existing model instance and no general GUI . . . 6

2.2.3 Starting without an Graphical User Interface . . . 6

2.3 Thread management . . . 6

3 Controller 7

3.1 List of Commands . . . 7

3.2 Command Example . . . 7

3.3 Different Tabs for different Models . . . 8

3.4 Controller management: Multiple Tabs and CommandBuilders . . . 8

4 View 9

4.1 Graphical User Interface . . . 9

4.2 Flowchart . . . 9

4.3 GUI Facade . . . 11

4.3.1 class GUIFacade . . . 11

4.4 Warning Menu . . . 12

5 Model 13

5.1 Model Facade . . . 13

5.1.1 class ModelFacade . . . 14

5.2 Flowchart . . . 15

5.2.1 abstract class FlowChartElement . . . 16

5.2.2 abstract class Module . . . 16

5.2.3 class FlowChart . . . 17

5.2.4 Validation . . . 18

5.2.5 Pins and Connections . . . 23

5.3 Graphical Representation . . . 24

5.3.1 abstract class GraphicalRepresentation . . . 26

5.3.2 abstract class ModulePinGR . . . 27

5.3.3 class StdInputPinGR . . . 28

5.3.4 class StdOutputPinGR . . . 28

5.3.5 class RigidPinGR . . . 28

5.3.6 abstract class FlowChartElementGR . . . 28

5.3.7 abstract class ModuleGR . . . 29

5.3.8 abstract class ConnectionGR . . . 31

5.3.9 class StdArrowGR . . . 31

5.3.10 class RigidConnectionGR . . . 32

5.3.11 class ConnectionLineGR . . . 33

5.3.12 class FlowChartGR 34
5.3.13 class LabelGR 35
5.3.14 «enum» PinOrientation 35
5.3.15 Calculate Route detailed specification 36
5.4 Simulation 36
5.4.1 class Simulation 37
5.4.2 class SimulationConfiguration 38
5.4.3 interface OctaneStepper 39
5.4.4 abstract class Stepper 39
5.4.5 class SchedulingStepper 39
5.4.6 abstract class Scheduler 39
5.4.7 class PriorityScheduler 39
5.4.8 Scheduler detailed specification 40
5.4.9 Simulation Sequence 41
5.5 Store and Load 42
5.5.1 Formal of modules and dynamic loading 42
5.5.2 class StoreLoadFacade 43
5.6 Data types 44
5.6.1 abstract class Data 45
5.6.2 abstract class DataSignature 46
5.7 Units 47
5.7.1 class UnitD 47
5.8 Function module 48
5.8.1 class FunctionElement 48
5.8.2 Change Function Sequence 49
5.8.3 Validation override 49
5.9 Warning system 51
5.10 Error handling 52
5.10.1 Handling Exceptions 52

6 References to the requirement specification **53**

1 Introduction

This document shows the Software Design of our Flowchart-Module for the Open-Source simulation environment OCTANE. The design focuses on the correct completion of all criteria specified in the Product Specifications. It is important to notice that besides fulfilling all Must-Haves criteria in the Product Specifications, our design also integrates as many Nice-To-Have criteria as possible. This can lead to possible changes in the design during the Implementation-Phase. Besides the correct implementation of all criteria, expandability is one of the main goals our design achieves. Each part is designed carefully to enable future developers to change the behaviour of the Software according to their needs and allow future changes of the operation mode that cannot be foreseen. Therefore we are only supplying one possible solution to the complex problem of managing a Flowchart-Module. The design itself is quite complex which challenged us to present it as clearly as possible. Therefore we chose to explain the whole product design by splitting it into various pieces and explaining each piece by its own. We also added descriptions to the diagrams to make it easier to grasp the idea behind each design. Important sequences are supported by sequence diagrams which show their flow through the program.

2 Design

This section shows the overall structure of our product. It is supposed to provide the reader with an overview of the design. Each part of the design is then further explained in the corresponding sections.

2.1 Global Design

Since we develop a software with a reusable model we decided to use the Model-View-Controller (MVC) architectural pattern. This pattern enables us and other developers to create multiple views, models and controllers. It makes the program itself open to expansion since multiple views and models can be used and fitted to the users needs. We will now explain our overall design by presenting our package diagram and the more detailed correspondent to it.

2.1.1 Package Diagram

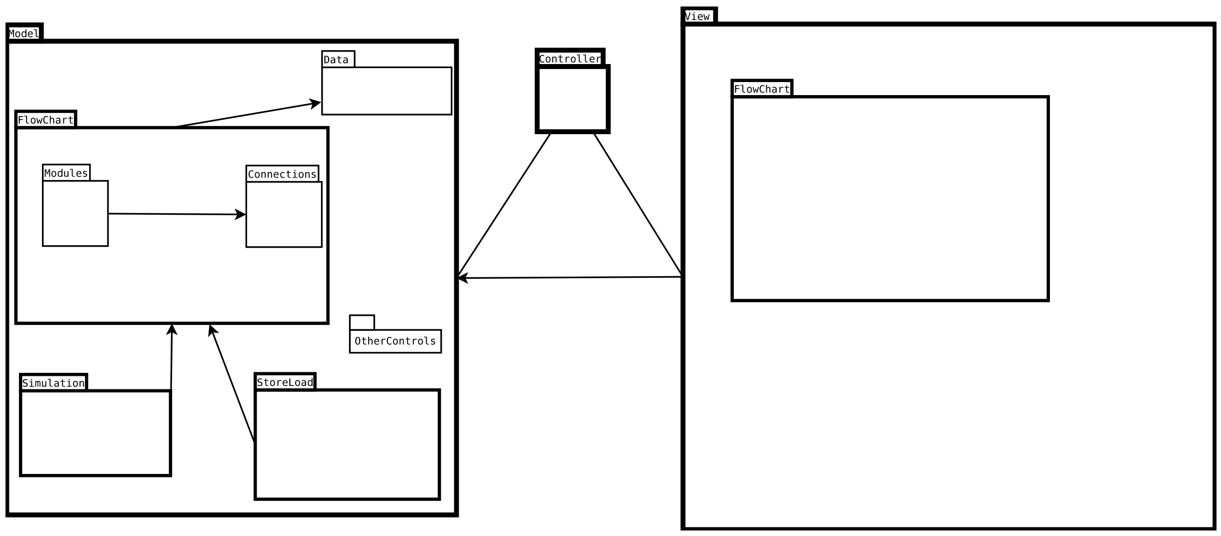


Figure 1: Package Diagram of the global design - This diagram shows the package structure of our global design following the MVC-architectural pattern.

Our global design and therefore also our package diagram (as seen in figure 1) follows the architectural pattern Model-View-Controller. This pattern splits our product into three connected parts. It separates the presentation from the internal data which makes the code more expandable and better capsuled. This is clearly visible by looking at the package diagram and the three summarizing packages View, Model and Controller. This design allows us to create multiple models and views without having to change the whole application. The encapsulation is further improved by the controller which manages the main communication between the parts. The diagram shows how our packages are placed in the MVC-architectural pattern and how they interact. Especially in the Model package there are many things to notice: The Simulation package is responsible for stepping through the flowchart and all related actions. The StoreLoad package manages how flowcharts and other data from the Data package is stored and loaded. At last, the Flowchart package manages the modules and connections in the matching flowchart and how they are created. All these packages interact heavily which is explained in detail later in this document.

2.1.2 Detailed Global Design

To further examine our global design we present the overall structure of our application in figure 2 combined with the most important classes it contains. One can still recognize the MVC architectural pattern. The details of each part of the global design are explained throughout this document. Here we just present the structure to give an overall idea how the classes are connected and how they interact. It is supposed to help to understand the functionality of each part of our application in the global context.

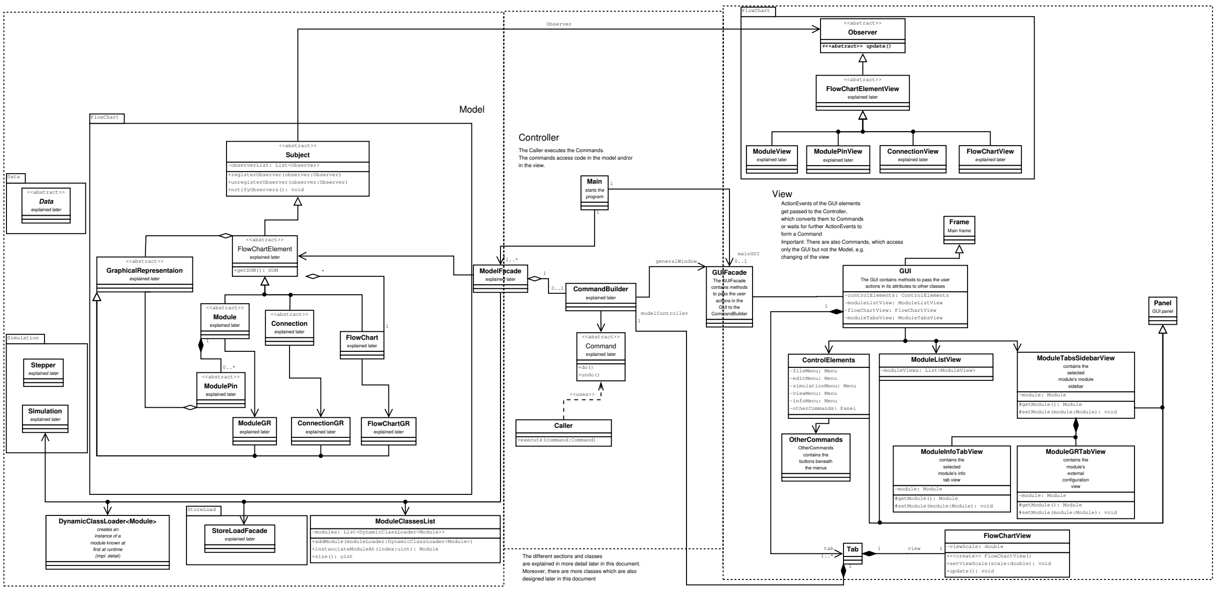


Figure 2: Class diagram of the global design - This diagram shows the structure of our overall design. One can see the most important classes our applications contains and how they are connected. Besides showing important classes, notice that the diagram still shows the integration of the MVC-architectural pattern.

2.2 Starting the Program

The Design we forged provides multiple ways to start the program. This is quite important since it strongly increases the possibilities of scenarios in which our software can be used.

2.2.1 Starting without arguments

The most simple way to start our program is to pass no arguments at all. Doing this will start an empty Flowchart-Program with the main Graphical User Interface (GUI) attached. In the interface the user can modify the flowchart and add and remove components. The user is also able to control the simulation of the flowchart from within the interface.

2.2.2 Starting with an existing model instance and no general GUI

A different possibility we provide to start the program is to start with a view of an existing model instance and no general GUI. This option is generally useful to other program developers, since it allows them to integrate our product into their plug-in based architecture.

2.2.3 Starting without an Graphical User Interface

The last option is to launch no GUI at all. This option simulates and loads models hidden from the end user. A common use case would be simulating a flowchart in an OCTANE simulation. This option allows the user to simulate various objects in his program from which the user calls our model, without having to start a GUI which the user may not need.

2.3 Thread management

It is always important to have a fluid interaction with the graphical representation of the program. To guarantee a flawless interaction, we are running the GUI on a different thread. There are some other important threads too. We have decided that each model will be running on a different thread to gain performance. When a simulation is being calculated on a model, there will be a Timeout-thread, killing the separate model-calculation process if it takes too long. It has to be added that the general simulation will be sequential. This reduces the complexity of our Simulation package drastically.

3 Controller

The controller interacts with the Model and the View to put the user input in operation. The user input is generated and passed by the View to the CommandBuilder. It then summarizes actions to commands which can be executed. Each command offers a method "do()" to be executed, and "undo()" to revert the command's effects. After the command building is finished, Caller.execute(command) is being called and the command is executed. The CommandBuilder has a list of the last recent 20 commands, so it is able to possibly revert them.

3.1 List of Commands

File menu	**Edit menu**	**Simulation menu**	**View menu**	**Info menu**	**FlowChart**
NewDocument	Undo	StartSimulation	ZoomIn	ShowInfo	AddModule
Open	Redo	PauseSimulation	ZoomOut		AddConnection
Save		StopSimulation	ZoomStd		ChangeModule
Save As		RunSimulationCycle	ToggleGrid		ChangeConnection
Import		AddModule	ShiftView		SelectModule
Export		ReloadModuleList	CenterView		SelectConnection
			ToggleModuleList		DeselectModule
					DeselectConnection
					DeleteModule
					DeleteConnection

This list shows a first overview of the commands. The commands mentioned provide the functionality to satisfy all must-have criteria and some of the nice-to-have criteria. We do not claim the list to be complete and reserve our right to further update the list if necessary. It shows which commands are connected to which menus in the GUI. All other control elements in the GUI are equivalent to the commands from the menus and are therefore not mentioned here.

3.2 Command Example

Here we present the flow through our application when a command is executed. It is explained in the sequence diagram in figure 3 which shows the sequence of the start simulation command. The user first presses the StartSimulationButton, from which the action gets passed to the CommandBuilder. The CommandBuilder creates a new StartSimulationCommand instance and tells the Caller to execute it. The Caller runs the "do()" method of the command instance, which calls "startSimulation()". The shown sequence diagram is simplified, because the command does more, e.g. set the edit state of the GUI. These extra steps are not important to show the overall flow of a command and would be rather distracting and confusing. The data changes in the model that result from a command will be passed to the View of the flowchart by observers and therefore not by a command.

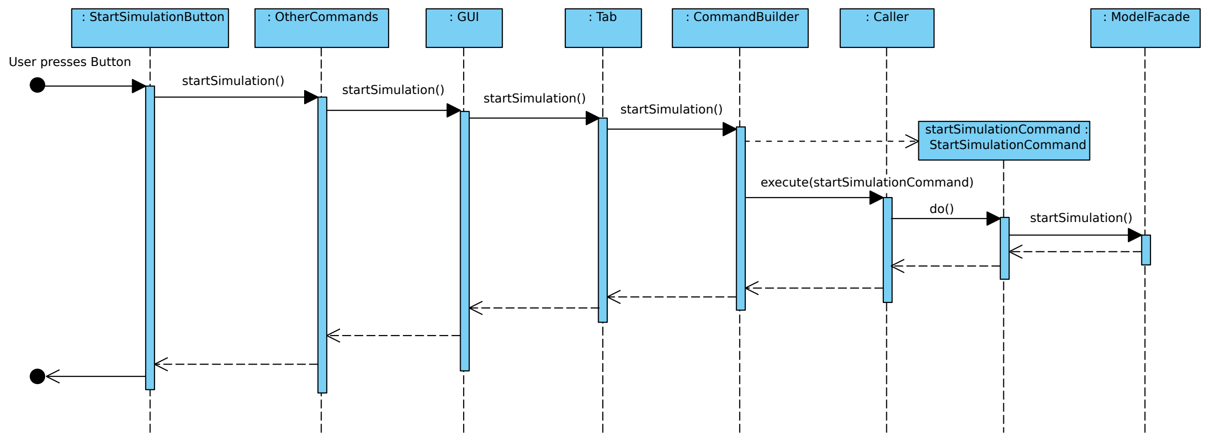


Figure 3: Start simulation command - This sequence diagram shows the flow through the class structure of an command explained at the example of the "Start Simulation Command".

3.3 Different Tabs for different Models

As mentioned before there can be multiple models opened in the application. This gets us to the question of what the interaction between these different CommandBuilders and the single GUIFacade could look like. Our solution encapsulates the different CommandBuilders in Tabs, which are held by the GUI class. This means that only CommandBuilders are going to use the GUIFacade because of simplicity, not the other way around.

3.4 Controller management: Multiple Tabs and CommandBuilders

Generally every instance of CommandBuilder has its own respective Model and each CommandBuilder is possessed by a Tab. This makes it possible to run multiple simulations at once. Furthermore there does not have to be a View at all because a View could be added to any invisible module. In case no View is added, the Model rests hidden in the background, waiting to be called e.g. by OCTANE. If a View is visible, the actions performed in the GUI are forwarded to the respective CommandBuilder. To manage all Models, the GUIFacade will indirectly hold all Tab instances with their CommandBuilder and thus their Model attached. Moreover, the Main class can also hold further invisible Models.

4 View

The view is responsible to interact with the user and put his/her input into action. Everything that is visible to the user is managed by the View. It describes the interface between the user and all data and actions. If the user wants to access data or wants to interact with a flowchart, the user tells it to the View by performing the correct actions in the View. The View then communicates those actions to the Controller which if applicable interacts with the Model. Overall, the View encapsulates the part of the application that a normal user would see from the outside. It is part of the MVC-architectural pattern and therefore open to expansion and change.

4.1 Graphical User Interface

The GUI contains the panels mentioned in the global design. The ControlElements class contains e.g. menus and buttons, which pass user actions to the GUI. The passed actions are processed by the CommandBuilder, which is explained in the Controller. (section 3)

4.2 Flowchart

Figure 4 shows the structure of a flowchart and its elements inside the View. The classes in the diagram define the way how elements and its labels are displayed. The overall structure of this part follows the observer architectural design pattern. Each class in the diagram implements the observer class and therefore observes a corresponding partner in the graphical representation part of the product (for further details see section 5.3). Since all classes here are only visual parts of the graphical representation they only offer the three methods update(), draw() and delete(). The functionality of these methods is quite easy. The method update() is called by the graphical representation counter parts if the data of an object changed and it has to be updated in the view. The method draw() defines how each element is drawn inside of the View. Method delete() deletes the object from the View.

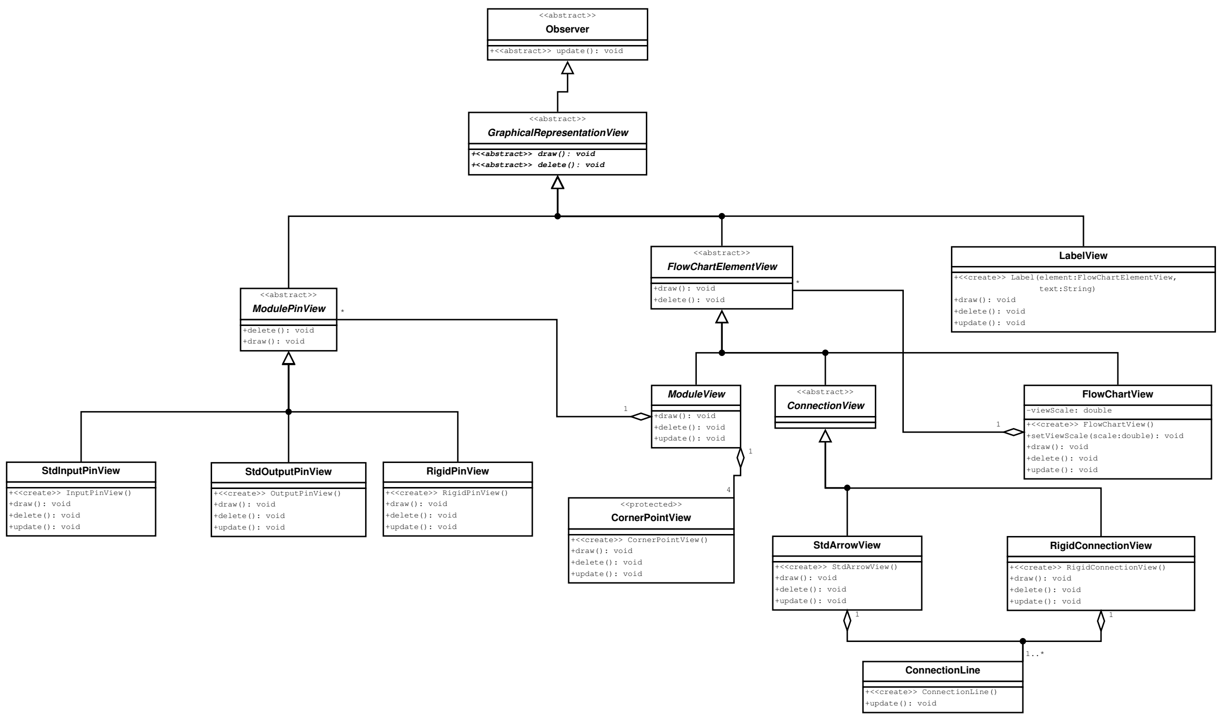
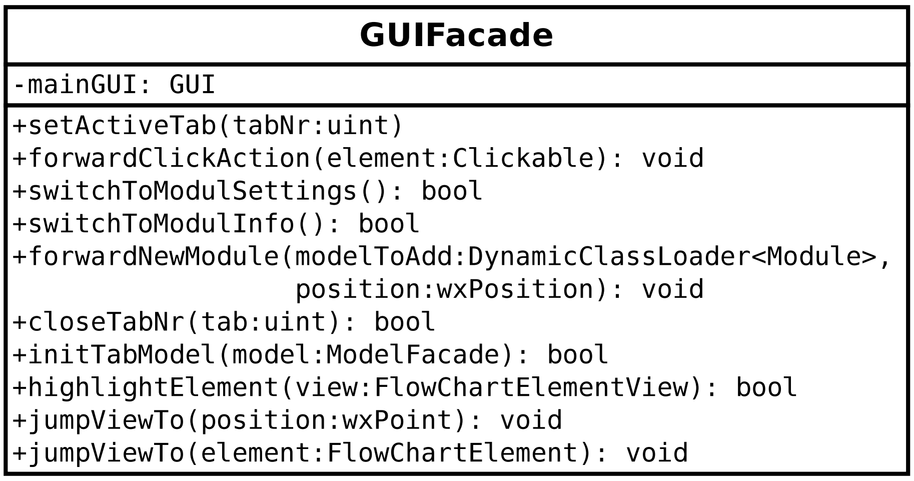


Figure 4: View of a flowchart - The class diagram shows how the view elements of a flowchart and its labels are internally structured and how they can interact.

One can see that the structure used to view the various elements is quite easy. Each element just defines how it is drawn and deleted from the flowchart and how it updates itself. This simple structure provided by the MVC pattern makes the View extremely simple and easy to expand. It encapsulated the data from how it is viewed.

4.3 GUI Facade

The GUIFacade is the main interaction point for Views, generally easing the communication between View elements and the respective CommandBuilder. Note that multiple methods are probably added and/or changed in the future.



```
GUIFacade
-mainGUI: GUI
+setActiveTab(tabNr:uint)
+forwardClickAction(element:Clickable): void
+switchToModulSettings(): bool
+switchToModulInfo(): bool
+forwardNewModule(modelToAdd:DynamicClassLoader<Module>,
                  position:wxPosition): void
+closeTabNr(tab:uint): bool
+initTabModel(model:ModelFacade): bool
+highlightElement(view:FlowChartElementView): bool
+jumpViewTo(position:wxPoint): void
+jumpViewTo(element:FlowChartElement): void
```

Figure 5: Facade of the GUI - The facade of the GUI defines how all parts can interact with the view.

4.3.1 class GUIFacade

`+setActiveTab(tabNr : uint) : void`
Lets the GUI switch to the desired tab.

`+forwardClickAction(element : Clickable) : void`
You can interact with many different elements and buttons in the GUI. To wrap click-events together, the element can be recognized using a Clickable interface. This method forwards the element to the CommandBuilder.

`+switchToModulSettings() : bool`
If a ModulSidebar is active, the tab gets changed to the settings part (if it was not there already). The method returns true, if a model was selected and false otherwise.

`+switchToModuleInfo() : bool`
If a ModulSidebar is active, the tab gets changed to the general info part. (if it was not there already). The method returns true, if a model was selected and false otherwise.

`+forwardNewModule(modelToAdd : DynamicClassLoader<Module>, position : wxPosition) : void`
If a new module gets dragged into a flowchart, this method is forwarding the details to the assigned CommandBuiler.

`+closeTabNr(tab : uint) : bool`
Closes the tab with the number tab. Returns true, if the operation was successful.

`+initTabModel(model : ModelFacade) : bool`
This method can be called to show a previously hidden Flowchart instance in the GUI as a new tab. It returns true, if the operation was successful.

`+highlightElement(view : FlowChartElementView) : bool`
This method is used, if something wants to move certain FlowChartElements in the foreground by highlighting them. The method returns true if the module was found and could be highlighted.

`+jumpViewTo(position : wxPoint) : void`
Jumps with the views center to the given point.

`+jumpViewTo(element : FlowChartElement) : void`
Jumps with the views center to the given FlowChartElement.

4.4 Warning Menu

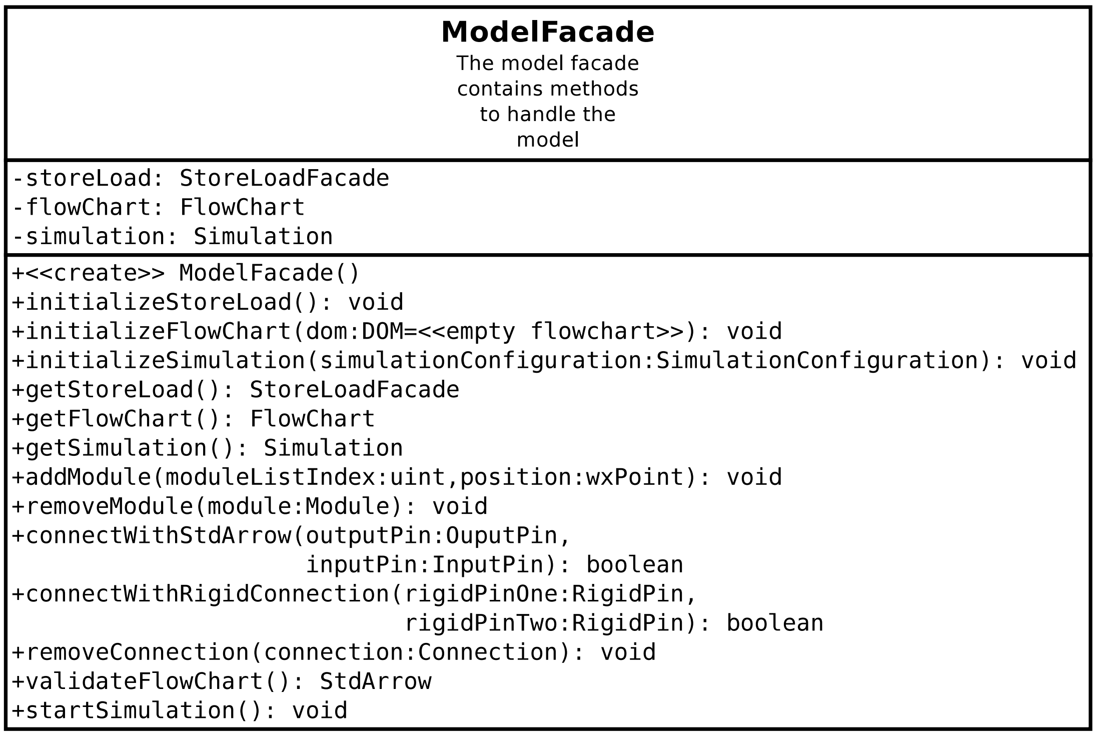
To give the user a better overview of what to do and to show potential problems, we implement a warning menu. The implementation is explained in the Model section 5.9. As of yet, you can find all current warnings for this Tab in the top-right of the upper menu bar. If you click on the respective icon, a view will appear which is holding all active warnings.

5 Model

The Model is responsible for the data and logic of the program. It is the most important part of the MVC pattern since everything is managed by the Model. Each element of the flowcharts only exists inside the Model and is simulated from there. The View only displays these elements and the data which exists inside the Model. Since it is the most important part it is obvious that it is also the most complex. In this section we explain our Model step by step with all its sub-packages and classes.

5.1 Model Facade

The ModelFacade allows other packages to interact with the Model. It can also be used from inside the Model package to ease access to other parts of the Model. It follows the facade software pattern which supports the idea of encapsulation. The facade provides protection of the interior of the Model and a clear interface to interact with it. The pattern also allows to make design decisions inside the package independently from the outside use of the package.



ModelFacade

The model facade contains methods to handle the model

```
-storeLoad: StoreLoadFacade
-flowChart: FlowChart
-simulation: Simulation
```

```
+<<create>> ModelFacade()
+initializeStoreLoad(): void
+initializeFlowChart(dom:DOM=<<empty flowchart>>): void
+initializeSimulation(simulationConfiguration:SimulationConfiguration): void
+getStoreLoad(): StoreLoadFacade
+getFlowChart(): FlowChart
+getSimulation(): Simulation
+addModule(moduleListIndex:uint,position:wxPoint): void
+removeModule(module:Module): void
+connectWithStdArrow(outputPin:OuputPin,
                     inputPin:InputPin): boolean
+connectWithRigidConnection(rigidPinOne:RigidPin,
                            rigidPinTwo:RigidPin): boolean
+removeConnection(connection:Connection): void
+validateFlowChart(): StdArrow
+startSimulation(): void
```

Figure 6: Facade of the Model - This facade of the Model defines how parts outside the Model can interact with the Model.

5.1.1 class ModelFacade

`+«create» ModelFacade()`
Creates a new ModelFacade. Nothing is initialized.

`+initialiazeStoreLoad() : void`
Initializes the StoreLoad package. This method should be called at up-start in order to be able to load necessary data.

`+initialiazeFlowChart(dom : DOM) : void`
Initializes the Flowchart package and builds a flowchart of the given DOM. If no DOM is passed to this method, an empty flowchart is created. This method should be called at up-start in order to have a main flowchart for this model.

`initialiazeSimulation(simulationConfiguration : SimulationConfiguration) : void`
Initializes the Simulation package with the given configuration. This method should be called at up-start in order to have a Simulation package for this model.

`+getStoreLoad() : StoreLoadFacade`
Returns the StoreLoadFacade for this model.

`+getFlowChart() : FlowChart`
Returns the main flowchart for this model.

`+getSimulation() : Simulation`
Returns a handle of the simulation for this model.

`+addModule(moduleListIndex : uint, position : wxPoint) : void`
Adds the module specified at the given moduleListIndex in the ModuleClassesList to this model's main flowchart at the specified position.

`+removeModule(module : Module) : void`
Removes the given Module.

`+connectWithStdArrow(outputPin : OutputPin, inputPin : InputPin) : boolean`
Connects the given OutputPin with the given InputPin using an StdArrow. Returns false if the connection cannot be created because the DataSignatures do not match. Only a fast check is done in this method.

`+connectWithRigidConnection(rigidPinOne : RigidPin, rigidPinTwo : RigidPin) : boolean`
Connects the two given rigid pins. Always returns true, because RigidPins are - like rigid connections - a feature which is in the software we develop only of graphical nature. However, other developers might add later a useful implementation of rigid connections in the program logic part of the model.

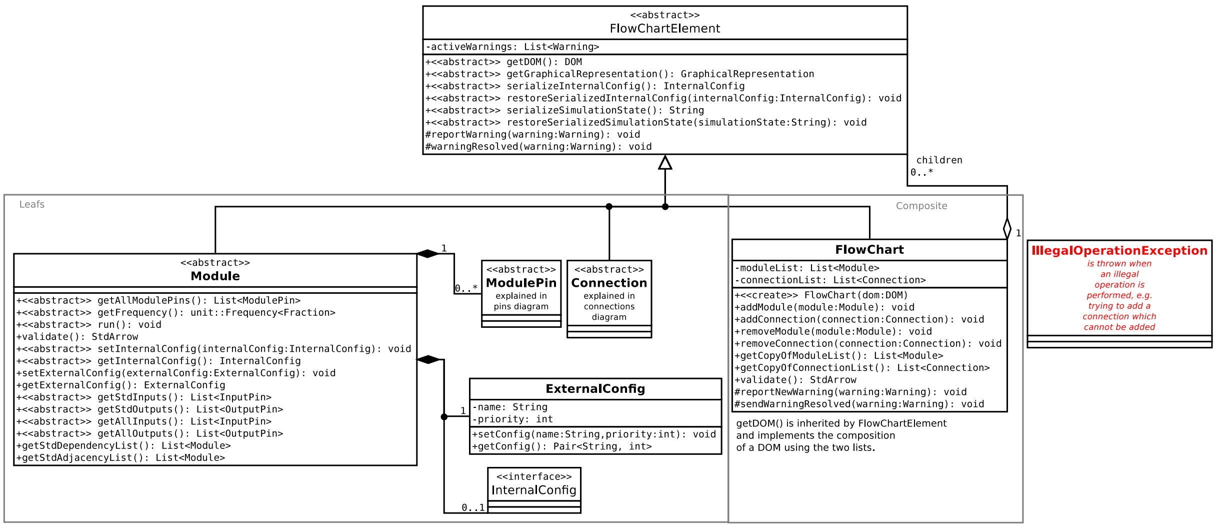
`+removeConnection(connection : Connection) : void`
Removes the given Connection.

`+validateFlowChart() : StdArrow`
Validates the main flowchart recursively. This method is called from inside Model by the Simulation package before starting the simulation. However, it can also be called from the outside to validate the flowchart without thereafter starting a simulation. Returns *null* if anything is correct, returns the first affected incorrect StdArrow found.

`+startSimulation() : void`
Starts the simulation. This method includes validating the flowchart beforehand, so caller does not have to validate it.

5.2 Flowchart

Figure 7 shows the overall structure of a flowchart and which elements it consists of. The structure is an implementation of the composite architectural software pattern which allows our flowchart elements to be quite variable. A flowchart can own any number of instances of FlowChartElement. Flowchart itself is a FlowChartElement which means that a flowchart is able to hold multiple flowcharts with various connections and modules each in the future. This makes this structure so adaptive which provides the possibility to encapsulate flowcharts inside a flowchart. Each class except ExternalConfig in figure 7 has a graphical representation as explained in section 5.3.



<<abstract>>
FlowChartElement
-activeWarnings: List<Warning>
+<<abstract>> getDOM(): DOM
+<<abstract>> getGraphicalRepresentation(): GraphicalRepresentation
+<<abstract>> serializeInternalConfig(): InternalConfig
+<<abstract>> restoreSerializedInternalConfig(internalConfig:InternalConfig): void
+<<abstract>> serializeSimulationState(): String
+<<abstract>> restoreSerializedSimulationState(simulationState:String): void
#reportWarning(warning:Warning): void
#warningResolved(warning:Warning): void
children
0..*
Leafs
Composite
<<abstract>>
Module
+<<abstract>> getAllModulePins(): List<ModulePin>
+<<abstract>> getFrequency(): unit::Frequency<Fraction>
+<<abstract>> run(): void
+validate(): StdArrow
+<<abstract>> setInternalConfig(internalConfig:InternalConfig): void
+<<abstract>> getInternalConfig(): InternalConfig
+setExternalConfig(externalConfig:ExternalConfig): void
+getExternalConfig(): ExternalConfig
+<<abstract>> getStdInputs(): List<InputPin>
+<<abstract>> getStdOutputs(): List<OutputPin>
+<<abstract>> getAllInputs(): List<InputPin>
+<<abstract>> getAllOutputs(): List<OutputPin>
+getStdDependencyList(): List<Module>
+getStdAdjacencyList(): List<Module>
1
0..*
<<abstract>>
ModulePin
explained in pins diagram
<<abstract>>
Connection
explained in connections diagram
ExternalConfig
-name: String
-priority: int
+setConfig(name:String,priority:int): void
+getConfig(): Pair<String, int>
1
<<interface>>
InternalConfig
0..1
FlowChart
-moduleList: List<Module>
-connectionList: List<Connection>
+<<create>> FlowChart(dom:DOM)
+addModule(module:Module): void
+addConnection(connection:Connection): void
+removeModule(module:Module): void
+removeConnection(connection:Connection): void
+getCopyOfModuleList(): List<Module>
+getCopyOfConnectionList(): List<Connection>
+validate(): StdArrow
#reportNewWarning(warning:Warning): void
#sendWarningResolved(warning:Warning): void
1
getDOM() is inherited by FlowChartElement and implements the composition of a DOM using the two lists.
IllegalOperationException
is thrown when an illegal operation is performed, e.g. trying to add a connection which cannot be added

Figure 7: Model of Flowchart elements (logic-part) - This class diagram shows the structure of the flowchart and its elements. It follows the composite architectural software pattern.

The diagram also shows the structure of each type of FlowChartElement quite nicely: Modules have multiple module pins attached to them and an external configuration as well as an internal configuration. Modules are instantiated by an empty constructor. Each class in the figure is further explained in the following sections.

5.2.1 abstract class FlowChartElement

`-activeWarnings : List<Warning>`
Contains a list of all warnings that have been reported.

`+«abstract» getDOM() : DOM`
Creates and returns the DOM of the current FlowchartElement.

`+«abstract» getGraphicalRepresentation() : GraphicalRepresentation`
Returns the GraphicalRepresentation of the current FlowchartElement.

`+«abstract» serializeInternalConfig() : InternalConfig`
Serializes this FlowChartElement. Returns the serialization as an InternalConfig.

`+«abstract» restoreSerializedInternalConfig(internalConfig : InternalConfig) : void`
Restores a serialized FlowChartElement.

`+«abstract» serializeSimulationState() : String`
Serializes the simulation state.

`+«abstract» restoreSerializedSimulationState(simulationState : String) : void`
Restores a serialized simulation state.

`#reportNewWarning(warning : Warning) : void`
This method is used to report a detected warning.

`#sendWarningResolved(warning : Warning) : void`
This method is used to report that a warning has been resolved.

5.2.2 abstract class Module

`+getAllModulePins() : List<Module>`
Returns all Pins of this module, including RigidPins and Pins added by future developers. At first in the list are all StdPins, thereafter all RigidPins, (thereafter other pins if any).

`+getFrequency() : Frequency<Fraction>`
Returns the module's frequency.

`+run() : void`
Runs the module. Is called in each simulation step in which this module would like to be called.

`+validate() : StdArrow`
Validates this module's Std-InputPins. Thereafter, it puts correct complete-does-care default data on all its Std-OutputPin. Returns a faulty StdArrow if any exists, otherwise returns NULL.

`+setInternalConfig(internalConfig : InternalConfig) : void`
Sets this module's internal configuration. This method should only be called by the module's respective ConfigurationView.

`+getInternalConfig() : InternalConfig`
Returns this modules internal configuration. This method should only be called by the module's respective ConfigurationView.

`+setExternalConfig(externalConfig : ExternalConfig) : void`
Sets this module's external configuration.

`+getExternalConfig() : ExternalConfiguration`
Returns this module's external configuration.

`+getStdInputs() : List<InputPin>`
Returns all Std-Inputs of this module.

`+getStdOutputs() : List<OutputPin>`
Returns all Std-Outputs of this module.

`+getAllInputs() : List<InputPin>`
Returns all inputs of this module.

`+getAllOutputs() : List<OutputPin>`
Returns all outputs of this module.

`+getStdInputs() : List<InputPin>`
Returns all Std-Inputs of this module.

`+getStdDependencyList() : List<Module>`
Returns all modules which are directly connected with a StdArrow to one of this module's Std-InputPins.

`+getStdAdjacencyList() : List<Module>`
Returns all modules which are directly connected with an StdArrow to one of this module's Std-OutputPins.

5.2.3 class FlowChart

`-moduleList : List<Module>`
Contains a list of modules being part of this flowchart.

`-connectionList : List<Connection>`
Contains a list of connections being part of this flowchart.

`+«create» Flowchart(dom : DOM)`
Creates a new flowchart with a given DOM. If no DOM is passed to this method, an empty flowchart is created.

`+addModule(module : Module) : void`
Adds a module to the current flowchart, say adds the given module to the moduleList.

`+addConnection(connection : Connection) : void`
Adds a connection to the current flowchart, say adds the given connection to the connectionList.

`+removeModule(module : Module) : void`
Removes a module from the current flowchart, say deletes the given module from the moduleList.

`+removeConnection(connection : Connection) : void`
Removes a connection from the current flowchart, say deletes the given connection from the connectionList.

`+getCopyOfModuleList() : List<Module>`
Returns a copy of the moduleList of the flowchart as a list of modules.

`+validate() : StdArrow`
Validates the main flowchart recursively. Returns *null* if anything is correct, returns the first affected incorrect StdArrow found.

`#reportNewWarning(warning : Warning) : void`
This method is used to report a detected warning.

`#sendWarningResolved(warning : Warning) : void`
This method is used to report that a warning has been resolved.

5.2.4 Validation

Flowchart validation is an important aspect of our work. It enables the user to create valid flowcharts and helps him to avoid major mistakes in the process of building the flowchart. This speeds up the development time of a flowchart and helps us internally to run the flowchart.

Figure 8 and 9 show a detailed specification how flowcharts are validated and figure 10 how modules are validated.

```
 1 public StdArrow* validate () {
 2   Module *moduleWithLowestName = moduleList.get(0);
 3   if(moduleWithLowestName == NULL) return NULL; //there are no modules
 4 
 5   Module *module = moduleWithLowestName;
 6   Set<Module*> visitedModules, roots;
 7   for(; visitedModules.size() < moduleList.size();
 8       module = set_complement(visitedModules, asSet(moduleList)).getFirst())
 9       //module is element of moduleList but not of visitedModules
10   {
11     Pair< Set<Module*>, Set<Module*> > visitedModulesAndFoundRoots =
12       BFS_Dependency(module);
13     visitedModules.addAll(visitedModulesAndFoundRoots.get(0));
14     roots.addAll(visitedModulesAndfoundRoots.get(1));
15   }
16 
17   //Now we have found all roots, we can now begin validating from the roots on
18   Set< List<Module*> > modulesInCorrectOrder; //for every root an own list
19   for(Module* module : roots)
20   {
21     Pair< Set<Module*>, List<Module*> > visitedModulesAndSortedModuleList =
22       BFS_Adjacency(module);
23     modulesInCorrectOrder.add(visitedModulesAndSortedModuleList.get(1));
24   }
25 
26   //the for-loops must be like this, because the lists must be validated "parallely"
27   for(int i = 0; i < maximumOfListSizes(modulesInCorrectOrder); i++) {
28     for(Set<Module*> list : modulesInCorrectOrder) {
29       if(i < list.size()) {
30         StdArrow affectedArrow = list.get(i).validate();
31         if(affectedArrow != NULL) return affectedArrow;
32       }
33     }
34   }
35 
36 
37   return NULL;
38 }
```

Figure 8: validate() in FlowChart (pseudocode)

Description of code 8:
From line 0 to 15 the roots of the different module dependencies are being found, as well as all visited modules generally. Afterwards the correct succession in relation to their dependencies is being calculated. The modules will be stored in the list, where its root is located. This happens up to line 24. At line 27 it is finally time to check each module in the calculated succession. This is achieved by iterating through each layer separately of the respective sorted root lists. In each iteration every module contained in this layer will be validated.

```
#define DEPENDENCY true
#define ADJACENCY false
//returns at elem0 the set of visited modules and at elem1 the set of roots
private Pair< Set<Module*>, Set<Module*> > BFS_Dependency(Module* root) {
  Set<Module*> roots; //let roots be on the heap,
            //so that it is possible to add elements in another method
  Pair< Set<Module*>, List<Module*> > bfsResult = BFS(module, DEPENDENCY, &roots);
  Pair< Set<Module*>, Set<Module*> > ret;
  ret.set(0, bfsResult.get(0));
  ret.set(1, roots);
  return ret;
}

//returns at elem0 the set of visited modules and at elem1 the list of modules
//(list because the order must be correct)
private Pair< Set<Module*>, List<Module*> > BFS_Adjacency(Module* root) {
  return BFS(module, ADJACENCY, NULL);
}

//returns at elem0 the set of visited modules and at elem1 the list of modules sorted by level
//(lowest index means low level (near root)), secondary by name
private Pair< Set<Module*>, List<Module*> >
            BFS(Module* root, bool reverse, Set*<Module*> highestLevelMarker) {
  //BFS and setting highest level marker
  Map<Module*, Module*> parentMap; //maps key=child to value=parent
  Map<Module*, int> levelMap; //maps key=module to value=level
  Set<Module*> q;
  q.add(root);
  parentMap.put(root, root);
  levelMap.put(root, 0);
  for(int l = 0; !q.isEmpty(); l++) {
    Set<Module*> qLocal;
    for(Module* module : q) {
      //scan module
      int adjFoundCount = 0;
      List<Module*> adjList;
      if(reverse) adjList = module.getStdDependencyList();
      else adjList = module.getStdAdjacencyList();
      for(Module* adj : adjList) {
        if(!parentMap.contains(adj)) { //unexplored
          qLocal.add(adj);
          levelMap.put(adj, l + 1);
          parentMap.put(adj, module); //module is now parent of adj
          adjFoundCount++;
        }
      }
      if(adjFoundCount == 0) highestLevelMarker.add(module);
    }
    q = qLocal;
  }
  //generating the return value
  int highestLevel = 0;
  Set<Module*> visited = parentMap.keySet();
  SortedSet<Module*> setOfModulesSorted; //sorter: primary: level, secondary: name
  for(Module* module : levelMap.keySet()) {
    int level = levelMap.get(module);
    setOfModulesSorted.insert(module); //sorted insert
    if(level > highestLevel) highestLevel = level;
  }
  List<Module*> listOfModulesSorted = setOfModulesSorted.asList();
  return Pair< Set<Module*>, List<Module*> >(visited,listOfModulesSorted);
}
```

Figure 9: Helper methods for validate() in FlowChart (pseudocode)

```
//in Module abstract class (standart implementation)

//notice: for a better understanding, cast exception throwing/handling is left out here
//notice: ValidatingExceptions will contain also which module is affected and other information
public StdArrow* validate() {
  List<StdPin> allInputPins = getStdInputs(); //casts not shown here
  List<OutputPin> allOutputPins = getStdOutputs();

  //checking input datatypes
  for(StdPin* pin : allInputPins) {
    Data *data = pin.getDefaultData();
    if(data == NULL)
        throw new ValidationException("This module has not specified default data!");

    DataSignature* signPtr = pin.getBeforeCheckDataSignature();
    if(signPtr == NULL)
        throw new ValidationException("This module has not specified data signature!");

    DataSignature pinDataSignature = &signPtr;
    if(!isCheckingRecursive(pinDataSignature))
        throw new ValidationException("This module has not overriden validate, 'though it had to!");

    StdArrow* arrow = (StdArrow*)(pin.getConnections().get(0));
    Data *dataPtr = arrow.getData();
    if(dataPtr == NULL) {
      //we are a module in a cycle(selected as root). However, we still need to check other pins
      continue;
    }

    Data data = &dataPtr;
    if(!data.hasSameType(pinDataSignature)) return arrow;
  }

  //putting default data on the outgoing connection
  for(OuputPin* pin : allOutputPins) {
    for(Connection* connection : getConnections()) {
      StdArrow* arrow = (StdArrow*)connection;
      Data *data = ((StdPin*)(pin.getModulePinImp())).getDefaultData();
      if(data == NULL)
        throw new ValidationException("This module has not specified default data!");

      arrow.putData(data);
    }
  }

  return NULL;
}

private bool isCheckingRecursive(DataSignature signature) {
  //base case
  if(signature.size() == 0 || !signature.isChecking()) {
    return signature.isChecking();
  }

  //recursive case
  for(int i = 0;i < signature.size();i++) {
    bool checking = signature.getComponent(i).isChecking() &&
         isCheckingRecursive(signature.getComponent(i));
    if(!checking) return false;
  }
  return true;
}
```

Figure 10: validate() in Module (pseudocode)

Description of code 9:
This pseudocode segment contains helper methods for the code 8. BFS_Dependency is executing BFS to generally find the set of roots and all found modules on its way. BFS_Adjacency adjusts the parameters and passes them into BFS.

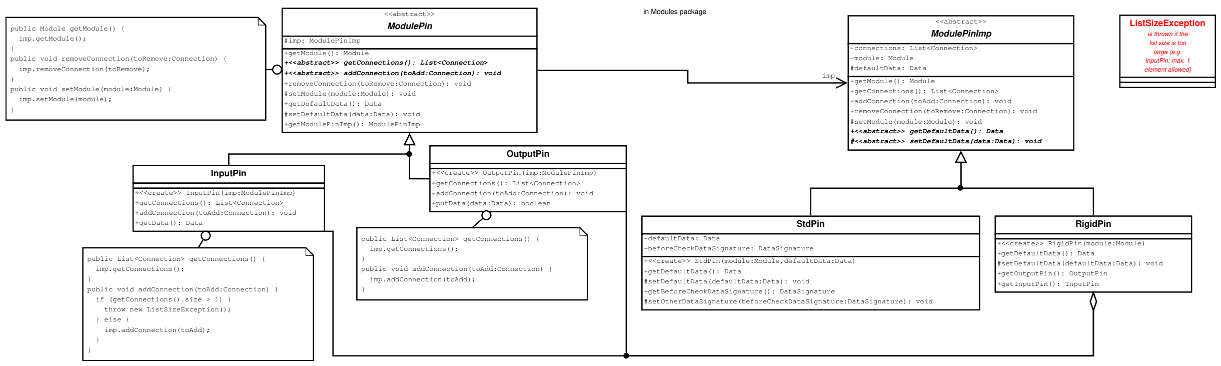
BFS is the main calculation function, realizing a Breadth-First-Search. The search is directly returning a sorted list of modules by their level in comparison to the passed root module.

Description of code 10:
The code shows the standard implementation of the validate() method inside Module. The verification proceeds as follows: At first there are basic checks for each input pin. It is checked, if every pin has its standard input set as well as certifying that each pin has a definition of its accepted DataSignature. If don't-cares are built in, the module has to override this method. At the end the method is getting the Data which lays on the input at the moment. Finally these inputs are compared to the DataSignatures which they should represent. If everything works correctly until now, the correct output is calculated and all output pins are sending their results.

During the code one helper function is called:
"isCheckingRecursive(signature : DataSignature)" looks at every internal data signature if it is also checking. If that is not the case, the DataSignature has don't-cares inside which do not have to be checked necessarily.

5.2.5 Pins and Connections

The diagrams in figure 11 and 12 show a part of the flowchart package in the Model. Figure 11 shows the pins residing in the Modules sub-package and figure 12 shows the connections residing in the Connections sub-package. Once a module is instantiates, it creates its necessary pins. For each standard pin, the module can also determine if it wants to have the data checked recursively at connection time. If it does not want it to be checked recursively at connection time, the compatibility of data types is only checked at validation time before starting the simulation. In order to create a standard pin the module creates an StdPin instance and uses this instance to create either an InputPin or an OutputPin. Thereafter, the module adds the created pin to the respective list which the module holds. In order to create a rigid pin the module creates a RigidPin and adds it to the respective list of rigid pins. Internally rigid pins have an input and an output pin.



in Modules package
<<abstract>>
ModulePin
#imp: ModulePinImp
+getModule(): Module
+<<abstract>> getConnections(): List<Connection>
+<<abstract>> addConnection(toAdd:Connection): void
+removeConnection(toRemove:Connection): void
#setModule(module:Module): void
+getDefaultData(): Data
#setDefaultData(data:Data): void
+getModulePinImp(): ModulePinImp
<<abstract>>
ModulePinImp
-connections: List<Connection>
-module: Module
#defaultData: Data
+getModule(): Module
+getConnections(): List<Connection>
+addConnection(toAdd:Connection): void
+removeConnection(toRemove:Connection): void
#setModule(module:Module): void
+<<abstract>> getDefaultData(): Data
#<<abstract>> setDefaultData(data:Data): void
imp
ListSizeException
is thrown if the list size is too large (e.g. InputPin: max. 1 element allowed)
InputPin
+<<create>> InputPin(imp:ModulePinImp)
+getConnections(): List<Connection>
+addConnection(toAdd:Connection): void
+getData(): Data
OutputPin
+<<create>> OutputPin(imp:ModulePinImp)
+getConnections(): List<Connection>
+addConnection(toAdd:Connection): void
+putData(data:Data): boolean
StdPin
-defaultData: Data
-beforeCheckDataSignature: DataSignature
+<<create>> StdPin(module:Module,defaultData:Data)
+getDefaultData(): Data
#setDefaultData(defaultData:Data): void
+getBeforeCheckDataSignature(): DataSignature
#setOtherDataSignature(beforeCheckDataSignature:DataSignature): void
RigidPin
+<<create>> RigidPin(module:Module)
+getDefaultData(): Data
#setDefaultData(defaultData:Data): void
+getOutputPin(): OutputPin
+getInputPin(): InputPin

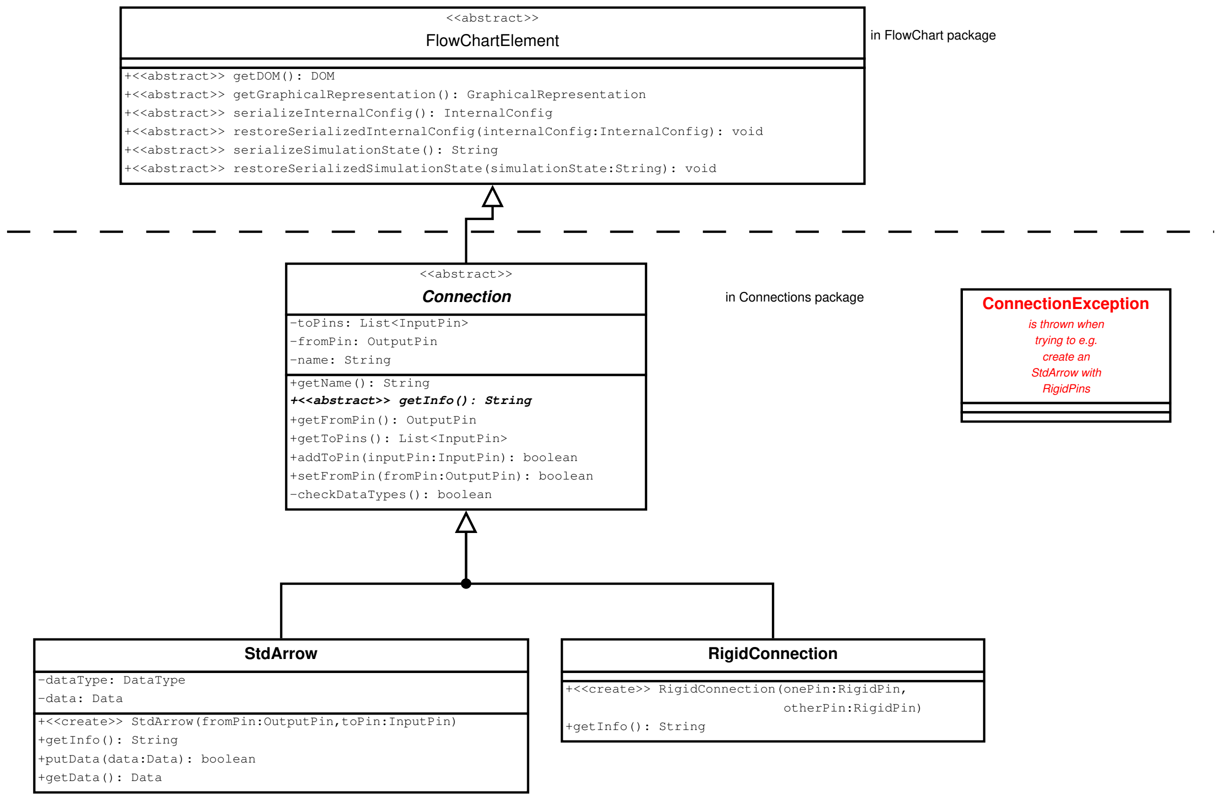
```
public Module getModule() {
  imp.getModule();
}
public void removeConnection(toRemove:Connection) {
  imp.removeConnection(toRemove);
}
public void setModule(module:Module) {
  imp.setModule(module);
}
```

```
public List<Connection> getConnections() {
  imp.getConnections();
}
public void addConnection(toAdd:Connection) {
  if (getConnections().size > 1) {
    throw new ListSizeException();
  } else {
    imp.addConnection(toAdd);
  }
}
```

```
public List<Connection> getConnections() {
  imp.getConnections();
}
public void addConnection(toAdd:Connection) {
  imp.addConnection(toAdd);
}
```

Figure 11: Model of pins - This class diagram shows the logic of pins inside the model.

Figure 11 shows the structure of the pin classes quite well. One can see that it strongly follows the bridge architectural software pattern making the pins easy to change in functionality. Each pin has its own implementation which defines the action of certain methods as can be seen in the detailed specifications (pseudocode) inside figure 11. This structure allows the pins to be very expandable and easy to change. This provides vast possibilities to change the pin's functionality according to the user's needs.



<<abstract>>
FlowChartElement
+<<abstract>> getDOM(): DOM
+<<abstract>> getGraphicalRepresentation(): GraphicalRepresentation
+<<abstract>> serializeInternalConfig(): InternalConfig
+<<abstract>> restoreSerializedInternalConfig(internalConfig:InternalConfig): void
+<<abstract>> serializeSimulationState(): String
+<<abstract>> restoreSerializedSimulationState(simulationState:String): void
in FlowChart package
<<abstract>>
Connection
-toPins: List<InputPin>
-fromPin: OutputPin
-name: String
+getName(): String
+<<abstract>> getInfo(): String
+getFromPin(): OutputPin
+getToPins(): List<InputPin>
+addToPin(inputPin:InputPin): boolean
+setFromPin(fromPin:OutputPin): boolean
-checkDataTypes(): boolean
in Connections package
ConnectionException
is thrown when trying to e.g. create an StdArrow with RigidPins
StdArrow
-dataType: DataType
-data: Data
+<<create>> StdArrow(fromPin:OutputPin,toPin:InputPin)
+getInfo(): String
+putData(data:Data): boolean
+getData(): Data
RigidConnection
+<<create>> RigidConnection(onePin:RigidPin, otherPin:RigidPin)
+getInfo(): String

Figure 12: Model of connections - The class diagram shows the connections class and its subclasses from the connections package.

The diagram shows the two types of connections we offer: StdArrow and RigidConnection. The StdArrow is a standard connection to send data from modules to other modules. The rigid connection is a connection which at the moments is only visually since it does not transfer any data. Overall the connections as well as the pins are made to be very easy to change and expanded. Their functionality can be changed completely to behave as the user wants them to do.

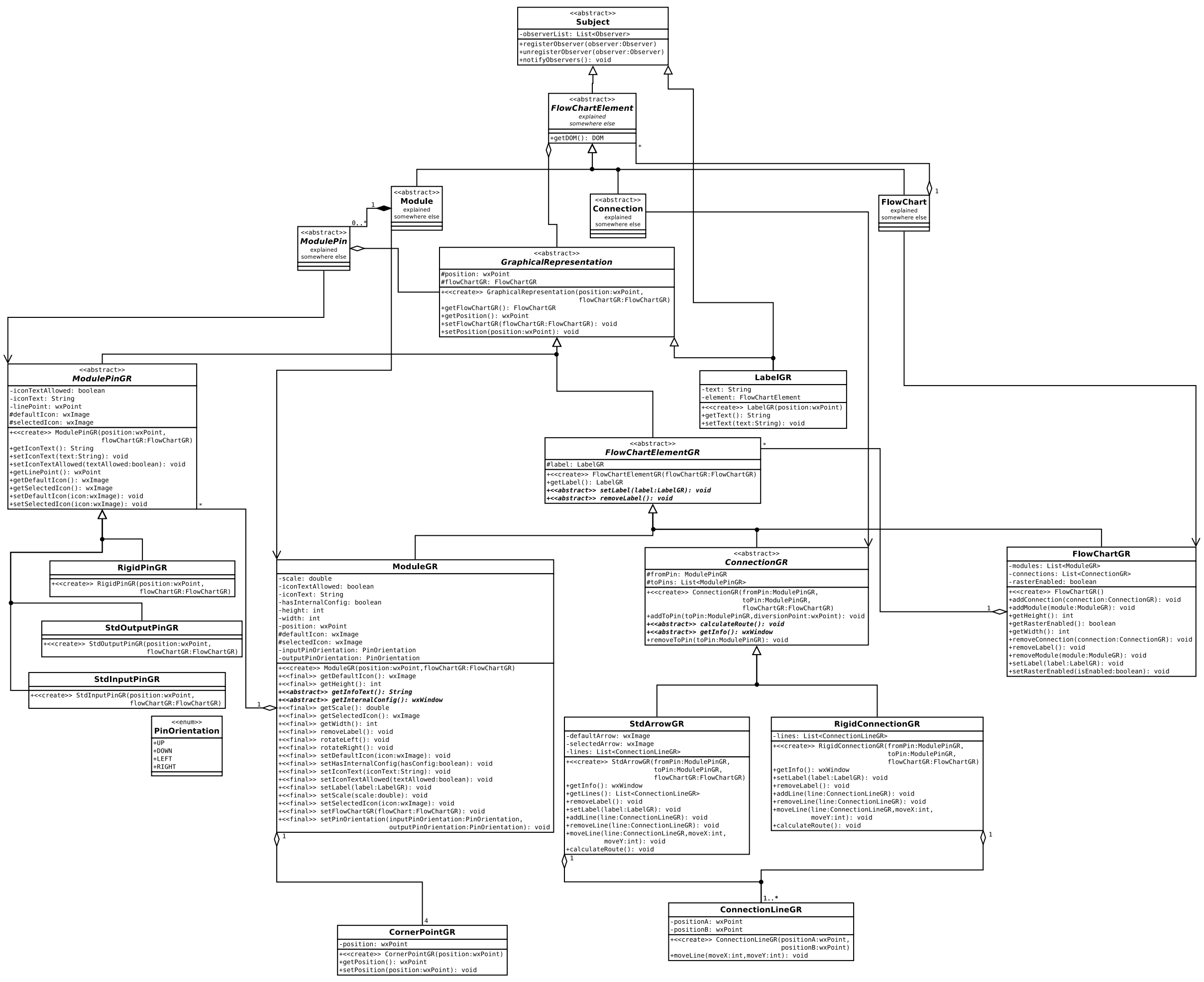
5.3 Graphical Representation

The flowchart's graphical representation is adjusted to the Model-View-Controller architectural software pattern. In the Model each element of the flowchart (FlowChartElement) which can be seen in the chart has a matching graphical representation (GraphicalRepresentation). In our case these are Module, Connection and FlowChart. In GraphicalRepresentation there is also a Label available which is not a FlowChartElement itself, but enables us to add labels to FlowChartElements that can also be viewed with the matching element.

For the module pins we offer the classes StdInputGR, StdOutputGR and RigidPinGR as graphical representations of inputs, outputs and pins for rigid connections. In general, each module pin can set its own icon which can be viewed and a text which is presented inside the icon.

Modules generally have four corners which are described by the class CornerPointGR. With each corner point it is possible to change the size of the module: when the user clicks on a corner point and moves the point to another position, CornerPointGR updates its own position and the scaling of the module. In this step the module checks that the other three corner points match to the new scaling as well. Modules can be scaled and turned in clockwise and anti-clockwise direction.

Similar to the module pins, each module can have an icon and a text which is displayed inside the icon. Modules have settings concerning their view but also configurations concerning their behaviour which are shown in the ModuleTabsSidebar. Additionally modules can have an internal configuration, which is presented in a separate window provided by the module itself because we do not know which configurations a module would like to provide. Modules have to posses a FlowChartGR. Each module can also have a label.



<<abstract>> Subject
-observerList: List<Observer>
+registerObserver(observer:Observer)
+unregisterObserver(observer:Observer)
+notifyObservers(): void
<<abstract>> FlowChartElement explained somewhere else
+getDOM(): DOM
*
<<abstract>> Module explained somewhere else
1
0..*
<<abstract>> ModulePin explained somewhere else
<<abstract>> Connection explained somewhere else
FlowChart explained somewhere else
1
<<abstract>> GraphicalRepresentation
#position: wxPoint
#flowChartGR: FlowChartGR
+<<create>> GraphicalRepresentation(position:wxPoint, flowChartGR:FlowChartGR)
+getFlowChartGR(): FlowChartGR
+getPosition(): wxPoint
+setFlowChartGR(flowChartGR:FlowChartGR): void
+setPosition(position:wxPoint): void
<<abstract>> ModulePinGR
-iconTextAllowed: boolean
-iconText: String
-linePoint: wxPoint
#defaultIcon: wxImage
#selectedIcon: wxImage
+<<create>> ModulePinGR(position:wxPoint, flowChartGR:FlowChartGR)
+getIconText(): String
+setIconText(text:String): void
+setIconTextAllowed(textAllowed:boolean): void
+getLinePoint(): wxPoint
+getDefaultIcon(): wxImage
+getSelectedIcon(): wxImage
+setDefaultIcon(icon:wxImage): void
+setSelectedIcon(icon:wxImage): void
*
LabelGR
-text: String
-element: FlowChartElement
+<<create>> LabelGR(position:wxPoint)
+getText(): String
+setText(text:String): void
<<abstract>> FlowChartElementGR
#label: LabelGR
+<<create>> FlowChartElementGR(flowChartGR:FlowChartGR)
+getLabel(): LabelGR
+<<abstract>> setLabel(label:LabelGR): void
+<<abstract>> removeLabel(): void
*
RigidPinGR
+<<create>> RigidPinGR(position:wxPoint, flowChartGR:FlowChartGR)
StdOutputPinGR
+<<create>> StdOutputPinGR(position:wxPoint, flowChartGR:FlowChartGR)
StdInputPinGR
+<<create>> StdInputPinGR(position:wxPoint, flowChartGR:FlowChartGR)
<<enum>> PinOrientation
+UP
+DOWN
+LEFT
+RIGHT
ModuleGR
-scale: double
-iconTextAllowed: boolean
-iconText: String
-hasInternalConfig: boolean
-height: int
-width: int
-position: wxPoint
#defaultIcon: wxImage
#selectedIcon: wxImage
-inputPinOrientation: PinOrientation
-outputPinOrientation: PinOrientation
+<<create>> ModuleGR(position:wxPoint,flowChartGR:FlowChartGR)
+<<final>> getDefaultIcon(): wxImage
+<<final>> getHeight(): int
+<<abstract>> getInfoText(): String
+<<abstract>> getInternalConfig(): wxWindow
+<<final>> getScale(): double
+<<final>> getSelectedIcon(): wxImage
+<<final>> getWidth(): int
+<<final>> removeLabel(): void
+<<final>> rotateLeft(): void
+<<final>> rotateRight(): void
+<<final>> setDefaultIcon(icon:wxImage): void
+<<final>> setHasInternalConfig(hasConfig:boolean): void
+<<final>> setIconText(iconText:String): void
+<<final>> setIconTextAllowed(textAllowed:boolean): void
+<<final>> setLabel(label:LabelGR): void
+<<final>> setScale(scale:double): void
+<<final>> setSelectedIcon(icon:wxImage): void
+<<final>> setFlowChartGR(flowChart:FlowChartGR): void
+<<final>> setPinOrientation(inputPinOrientation:PinOrientation, outputPinOrientation:PinOrientation): void
1
1
<<abstract>> ConnectionGR
#fromPin: ModulePinGR
#toPins: List<ModulePinGR>
+<<create>> ConnectionGR(fromPin:ModulePinGR, toPin:ModulePinGR, flowChartGR:FlowChartGR)
+addToPin(toPin:ModulePinGR,diversionPoint:wxPoint): void
+<<abstract>> calculateRoute(): void
+<<abstract>> getInfo(): wxWindow
+removeToPin(toPin:ModulePinGR): void
FlowChartGR
-modules: List<ModuleGR>
-connections: List<ConnectionGR>
-rasterEnabled: boolean
+<<create>> FlowChartGR()
+addConnection(connection:ConnectionGR): void
+addModule(module:ModuleGR): void
+getHeight(): int
+getRasterEnabled(): boolean
+getWidth(): int
+removeConnection(connection:ConnectionGR): void
+removeLabel(): void
+removeModule(module:ModuleGR): void
+setLabel(label:LabelGR): void
+setRasterEnabled(isEnabled:boolean): void
1
StdArrowGR
-defaultArrow: wxImage
-selectedArrow: wxImage
-lines: List<ConnectionLineGR>
+<<create>> StdArrowGR(fromPin:ModulePinGR, toPin:ModulePinGR, flowChartGR:FlowChartGR)
+getInfo(): wxWindow
+getLines(): List<ConnectionLineGR>
+removeLabel(): void
+setLabel(label:LabelGR): void
+addLine(line:ConnectionLineGR): void
+removeLine(line:ConnectionLineGR): void
+moveLine(line:ConnectionLineGR,moveX:int, moveY:int): void
+calculateRoute(): void
1
RigidConnectionGR
-lines: List<ConnectionLineGR>
+<<create>> RigidConnectionGR(fromPin:ModulePinGR, toPin:ModulePinGR, flowChartGR:FlowChartGR)
+getInfo(): wxWindow
+setLabel(label:LabelGR): void
+removeLabel(): void
+addLine(line:ConnectionLineGR): void
+removeLine(line:ConnectionLineGR): void
+moveLine(line:ConnectionLineGR,moveX:int, moveY:int): void
+calculateRoute(): void
1
4
CornerPointGR
-position: wxPoint
+<<create>> CornerPointGR(position:wxPoint)
+getPosition(): wxPoint
+setPosition(position:wxPoint): void
1..*
ConnectionLineGR
-positionA: wxPoint
-positionB: wxPoint
+<<create>> ConnectionLineGR(positionA:wxPoint, positionB:wxPoint)
+moveLine(moveX:int,moveY:int): void

Figure 13: Model of Flowchart elements and labels (graphical representation part) - The figure shows the graphical representation of each Flowchart element. Also it shows the application of the observer architectural software design pattern.

Connections always have one output pin (from pin) and possibly more than one input pin (to pin) assigned to them. These are added and removed one by one. The path a connection takes is calculated automatically by a rather simple algorithm. Furthermore each connection offers an information window with specific informations according to its connection type. Each connection can also have a label. The connection type arrow (StdArrowGR) contains one or more connection lines (ConnectionLineGR) and an arrow which are forged to a connection. Rigid connections only consist of one or more connection lines. It is possible to change the route of any connection.

A flowchart (FlowChartGR) can obtain any number of flowchart elements (FlowChartElementGR). Vice versa each flowchart element belongs to only one flowchart. Modules and connections as well as a label can be added and removed from a flowchart. Additionally the overall height and width of a flowchart can be returned and the grid in the background can be viewed and hidden.

The design in this part of the application follows the principle of the observer architectural software pattern. All described elements are implemented subjects of this pattern. The matching observers are in the View part of the application. A matching observer in the View has the suffix "-View" instead of "-GR". All implemented observers provide the methods update(), draw() and delete() to update, draw and delete the elements from the View. Additionally the View of the whole flowchart (FlowChartView) can be scaled.

5.3.1 abstract class GraphicalRepresentation

The GraphicalRepresentation class is the super class of all graphical representations of flowchart elements, module pins and labels. Each element has a position as well as a representation/icon.

`#position : wxPoint`
The elements position in the GUI. In general this refers to the upper left corner of the element.

`#flowChartGR : FlowChartGR`
Flowchart to which the element belongs.

`+«create» GraphicalRepresentation(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for GraphicalRepresentation. Parameters given are the position of the element and the flowchart it is assigned to.

`+getFlowChartGR() : FlowChartGR`
Getter for the Flowchart attribute.

`+getPosition() : wxPoint`
Getter for the elements position.

`+setFlowChartGR(flowChartGR : FlowChartGR) : void`
Setter for the Flowchart attribute.

`+setPosition(position : wxPoint) : void`
Setter for the elements position.

5.3.2 abstract class ModulePinGR

Graphical Representation of a module pin.

`-iconTextAllowed : boolean`
Attribute for deciding if it is possible to write a text on top of the icon.

`-iconText : String`
Text which is written on top of the icon.

`-linePoint : wxPoint`
Point where connection is connected to.

`#defaultIcon : wxImage`
Default pin icon which is displayed as long as the pin is not selected.

`#selectedIcon : wxImage`
Graphical representation of the pin as long as it is selected.

`+«create» ModulePinGR(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for ModulePinGR. Parameters given are the position of the pin and the flowchart it is assigned to.

`+getIconText() : String`
Getter for the icons text.

`+setIconText(text : String) : void`
Setter for the icons text.

`+setIconTextAllowed(textAllowed : boolean) : void`
Setter for the attribute textAllowed.

`+getLinePoint() : wxPoint`
Getter for the linePoint.

`+getDefaultIcon() : wxImage`
Getter for the defaultIcon.

`+getSelectedIcon() : wxImage`
Getter for the selectedIcon.

`+setDefaultIcon(icon : wxImage) : void`
Setter for the defaultIcon.

`+setSelectedIcon(icon : wxImage) : void`
Setter for the selectedIcon.

5.3.3 class StdInputPinGR

`+«create» StdInputPinGR(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for StdInputPinGR. Creates a new standard input pin. Parameters given are the position of the pin and the flowchart it is assigned to.

5.3.4 class StdOutputPinGR

`+«create» StdOutputPinGR(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for StdOutputPinGR. Creates a new standard output pin. Parameters given are the position of the pin and the flowchart it is assigned to.

5.3.5 class RigidPinGR

`+«create» RigidPinGR(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for RigidPinGR. Creates a new rigid pin. Parameters given are the position of the pin and the flowchart it is assigned to.

5.3.6 abstract class FlowChartElementGR

Abstract super class for all flowchart elements.

`#label : LabelGR`
Label which is assigned to an element.

`+«create» FlowChartElementGR(flowChartGR : FlowChartGR)`
Constructor method for FlowChartElementGR. The parameter given is the flowchart the element is assigned to.

`+getLabel() : LabelGR`
Getter for the element's label.

`+«abstract» setLabel(label : LabelGR) : void`
Setter for the element's label.

`+«abstract» removeLabel() : void`
Abstract method to remove the label from the element.

5.3.7 abstract class ModuleGR

`-scale : double`
Scaling of the module.

`-iconTextAllowed : boolean`
Boolean telling if a text can be written on top of the modules icon.

`-iconText : String`
Text which is displayed inside the icon.

`-hasInternalConfig : boolean`
Boolean telling if the module owns an internal configuration which can be accessed be the user.

`-height : int`
Height of the module.

`-width : int`
Width of the module.

`-position : wxPoint`
Position of the module.

`-priority : int`
Priority of the module.

`#defaultIcon : wxImage`
Graphical representation of the module while it is not selected.

`#selectedIcon : wxImage`
Graphical representation of the module while it is selected.

`+«create» ModuleGR(position : wxPoint, flowChartGR : FlowChartGR)`
Constructor method for ModuleGR. Parameters given is the position of the module and the flowchart it is assigned to.

`+«final» getDefaultIcon() : wxImage`
Getter for the defaultIcon.

`+«final» getHeight() : int`
Getter for the height of the module.

`+«abstract» getInfoText() : String`
Getter for the information text which can be setter individually by each module.

`+«abstract» getInternalConfig() : wxWindow`
Returns a new configuration window with the internal module configuration possibilities.

`+«final» getPriority() : int`
Getter for the modules priority.

`+«final» getScale() : double`
Getter for the modules scale.

`+«final» getSelectedIcon() : wxImage`
Getter for the selectedIcon.

`+«final» getWidth() : int`
Getter for the width of the module.

`+«final» removeLabel() : void`
Method to remove a module's label.

`+«final» rotateLeft() : void`
Method to rotate the module 90 degrees anti-clockwise.

`+«final» rotateRight() : void`
Method to rotate the module 90 degrees clockwise.

`+«final» setDefaultIcon(icon : wxImage) : void`
Setter for the defaultIcon.

`+«final» setHasInternalConfig(hasConfig : boolean) : void`
Setter of hasInternalConfig.

`+«final» setIconText(iconText : String) : void`
Setter for the iconText.

`+«final» setIconTextAllowed(textAllowed : boolean) : void`
Setter for iconTextAllowed.

`+«final» setLabel(label : LabelGR) : void`
Setter for the label. This method assigned a label to the method.

`+«final» setPriority(priority : int) : void`
Setter for the module's priority.

`+«final» setScale(scale : double) : void`
Setter for the module's scale.

`+«final» setSelectedIcon(icon : wxImage) : void`
Setter for the selectedIcon.

`+«final» setFlowChartGR(flowChart : FlowChartGR) : void`
Setter for the flowchart which the module is assigned to.

5.3.8 abstract class ConnectionGR

Abstract super class for all kind of connections between modules.

`#toPins : List<ModulePinGR>`
Pins the connection is connected to. Targets in case of directed connections.

`#fromPin : <ModulePinGR>`
Pins the connection is connected to. Sources in case of directed connections.

`+«create» ConnectionGR(fromPin : ModulePinGR, toPin : ModulePinGR, flowChartGR : FlowChartGR)`
Constructor method for ConnectionGR. Parameters given are the two pins the connection is connected to and the flowchart it is assigned to.

`+addToPin(toPin : ModulePinGR, diversionPoint : wxPoint) : void`
Method to add a new target pin.

`+«abstract» calculateRoute() : void`
Abstract method to calculate the route of an connection.

`+«abstract» getInfo() : wxWindow`
Abstract method to return an information window which contains individually configurated information for each connection type.

`+removeToPin(toPin : ModulePinGR) : void`
Method to delete the connection to the target pin. In case of branched connections is only deleted the part between the intersection and the target pin. In other cases it deletes the whole connection.

5.3.9 class StdArrowGR

Subclass of ConnectionGR defining standard directed connections in form of arrows.

`-defaultArrow : wxImage`
Default representation of the arrowhead as long as the connection is not selected.

`-selectedArrow : wxImage`
Representation of the arrowhead while the connection is selected.

`-lines : List<ConnectionLineGR>`
Connection lines which are contained by the connection.

`+«create» StdArrowGR(fromPin : ModulePinGR, toPin : ModulePinGR, flowChartGR : FlowChartGR)`
Constructor method for StdArrowGR. It creates an arrow between the source pin and the target pin defined by the passed parameters. The flowchart passed is the flowchart the arrow is assigned to.

`+getInfo() : wxWindow`
Method to return an information window showing informations of the arrow. In our case this is the flowing data and its type.

`+getLines() : List<ConnectionLineGR>`
Getter for the individual connection lines the arrow consists of.

`+removeLabel() : void`
Method to remove the label the arrow is assigned.

`+setLabel(label : LabelGR) : void`
Setter for the arrows label.

`+addLine(line : ConnectionLineGR) : void`
Method to add single connection lines to the arrow. This creates new segments in the arrows path.

`+removeLine(line : ConnectionLineGR) : void`
Method to remove single connection lines of the arrow.

`+moveLine(line : ConnectionLineGR, moveX : int, moveY : int) : void`
Method to move the passed connection line by moveX in the x-direction and moveY in the y-direction.

`+calculateRoute() : void`
Method to calculate the route of an arrow.

5.3.10 class RigidConnectionGR

Subclass of ConnectionGR which describes rigid connections which do not pass any data.

`-lines : List<ConnectionLineGR>`
Connection lines which are contained by the connection.

`+«create» RigidConnectionGR(fromPin : ModulePinGR, toPin : ModulePinGR, flowChartGR : FlowChartGR)`
Constructor method for RigidConnectionGR. Parameters passed are the two pins the connection is drawn between and the flowchart the rigid connection is assigned to.

`+getInfo() : wxWindow`
Method to return an information window given information about the connection.

`+setLabel(label : LabelGR) : void`
Setter for the label of the connection.

`+removeLabel() : void`
Method to remove the label from the connection.

`+addLine(line : ConnectionLineGR) : void`
Method to add single connection lines to the rigid connection. This creates new segments in the connections path.

`+removeLine(line : ConnectionLineGR) : void`
Method to remove single connection lines of the rigid connection.

`+moveLine(line : ConnectionLineGR, moveX : int, moveY : int) : void`
Method to move the passed connection line by moveX in the x-direction and moveY in the y-direction.

`+calculateRoute() : void`
Method to calculate the route of a rigid connection.

5.3.11 class ConnectionLineGR

Graphical representation of a straight line between two points.

`-positionA : wxPoint`
Point the line is connected to.

`-positionB : wxPoint`
Point the line is connected to.

`+«create» ConnectionLineGR(positionA : wxPoint, positionB : wxPoint)`
Constructor method for ConnectionLineGR. It creates a straight line between the two points passed.

`+moveLine(moveX : int, moveY : int) : void`
Method to change the lines position by the specified amounts in x- and y-direction.

5.3.12 class FlowChartGR

Flowchart graphical representation which consists of any number of connections, modules and flowcharts.

`-modules : List<ModuleGR>`
List of the modules inside the flowchart.

`-connections : List<ConnectionGR>`
List of the connections inside the flowchart.

`-rasterEnabled : boolean`
Boolean if the grid is activated or not.

`+«create» FlowChartGR()`
Constructor method for FlowChartGR. Creates a new flowchart.

`+addConnection(connection : ConnectionGR) : void`
Method to add a connection to the flowchart.

`+addModule(module : ModuleGR) : void`
Method to add a module to the flowchart.

`+getHeight() : int`
Getter for the height of the flowchart. Is calculated in perspective to the flowchart elements, not the grid.

`+getRasterEnabled() : boolean`
Getter for rasterEnabled.

`+getWidth() : int`
Getter for the width of the flowchart. Is calculated in perspective to the flowchart elements, not the grid.

`+removeConnection(connection : ConnectionGR) : void`
Method to remove the specified connection from the flowchart.

`+removeLabel() : void`
Method to remove the label from the flowchart.

`+removeModule(module : ModuleGR) : void`
Method to remove the specified module from the flowchart.

`+setLabel(label : LabelGR) : void`
Setter for the label of the flowchart.

`+setRasterEnabled(isEnabled : boolean) : void`
Setter for rasterEnabled.

5.3.13 class LabelGR

Graphical representation of a label of flowcharts and flowchart elements.

`-text : String`
Text which is shown inside the label.

`-element : FlowChartElement`
Flowchart element the label is assigned to.

`+«create» LabelGR(position : wxPoint)`
Constructor method to create a label at the given position.

`+getText() : String`
Getter for the labels text.

`+setText(text : String) : void`
Setter for the labels text.

5.3.14 «enum» PinOrientation

Orientation of the pins connected to a module.

`+UP`
Pin on the upper side of the module.

`+DOWN`
Pin on the lower side of the module.

`+LEFT`
Pin on the left side of the module.

`+RIGHT`
Pin on the right side of the module.

5.3.15 Calculate Route detailed specification

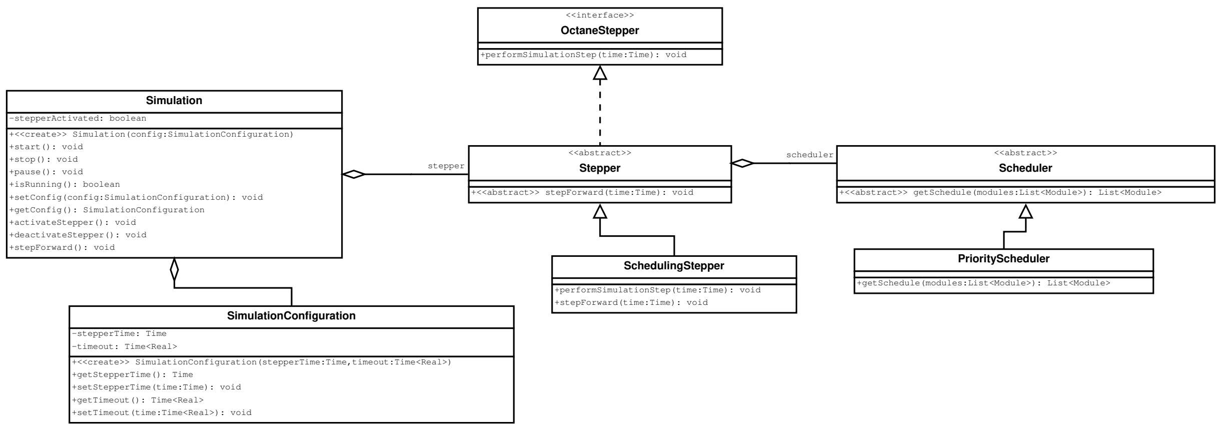
```
public void calculateRoute() {
  wxPoint fromPos = fromPin.getLinePoint();
  wxPoint[] toPos = new Array[toPin.size()];
  for (int i=0, i < toPin.size(), i++) {
    toPos[i] = toPin[i].getLinePoint();
  }
  for (int c = 0, c < toPin.size(), c++) {
    xDiff = toPos[c].x - fromPos.x;
    yDiff = toPos[c].y - fromPos.y;
    if (yDiff == 0 || xDiff == 0) {
      lines.add(new ConnectionLineGR(fromPos, toPos[1]));
    } else {
    int corner = new wxPoint(fromPos.x, from.y + yDiff)
    lines.add(new ConnectionLineGR(fromPos, corner));
    lines.add(new ConnectionLineGR(corner, toPos[c]));
    }
  }
}
```

Figure 14: calculateRoute() in RigidConnectionGR (pseudocode)

The algorithms described by figure 14 is a rather trivial way to calculate the route between two points. First the algorithm calculates the position of each point that is connected. This is one point for the source of the connection and possibly many for the targets of the connection. The algorithm then checks if two points are horizontally or vertically on the same line and connects them with one line if so. If this is not the case the two points can be connected by two lines. First the intersection of a vertical line from the source point and a horizontal line from the goal point is calculated. This intersection is connected to the two points. The connection now runs between the two points and this intersection/corner. All angles are 90 degrees. This algorithm is only a simple way to connect two points. Things to consider while creating a better algorithm for this problem would be the overlap of other elements and various visual aspects. This algorithm is the absolute minimum we will offer which provides all functionalities we need. A nice-to-have feature we would like to create is a routing algorithm which makes the flowchart visually attractive to the user.

5.4 Simulation

The simulation controls the flow of data inside flowcharts and determines when modules are executed. Therefore, the Simulation package is split into three parts: First, there is the Simulation class which defines the facade of the package. The Simulation class has a SimulationConfiguration which provides settings for the simulation to run. The other two main parts are the Stepper class and the Scheduler class. The Stepper class is the main actor of the package. It decides how modules are called and what happens during a step. In our case, the default stepper is the SchedulingStepper which calls all modules in the order of the schedule created by the scheduler. Therefore, the Scheduler class is the organizer of the package. By creating a schedule, it defines when the modules which want to step are executed. Our PriorityScheduler calculates the schedule by looking at each module's priority and therefore trying to give the modules with the highest priority the newest data. Which means that the order our Scheduler returns the modules is the optimal way to execute the modules so that the most important module (highest priority) gets the best data in each step. Also, it is important that the Stepper class implements OCTANE's OctaneStepper which enables other programs and applications to access our stepper from the outside and perform simulations without using other parts of our software.



<<interface>>
OctaneStepper
+performSimulationStep(time:Time): void
Simulation
-stepperActivated: boolean
+<<create>> Simulation(config:SimulationConfiguration)
+start(): void
+stop(): void
+pause(): void
+isRunning(): boolean
+setConfig(config:SimulationConfiguration): void
+getConfig(): SimulationConfiguration
+activateStepper(): void
+deactivateStepper(): void
+stepForward(): void
stepper
<<abstract>>
Stepper
+<<abstract>> stepForward(time:Time): void
scheduler
<<abstract>>
Scheduler
+<<abstract>> getSchedule(modules:List<Module>): List<Module>
SchedulingStepper
+performSimulationStep(time:Time): void
+stepForward(time:Time): void
PriorityScheduler
+getSchedule(modules:List<Module>): List<Module>
SimulationConfiguration
-stepperTime: Time
-timeout: Time<Real>
+<<create>> SimulationConfiguration(stepperTime:Time,timeout:Time<Real>)
+getStepperTime(): Time
+setStepperTime(time:Time): void
+getTimeout(): Time<Real>
+setTimeout(time:Time<Real>): void

Figure 15: Class diagram Simulation package - The diagram shows the classes of the simulation package. The structure of the diagrams follows the bridge architectural software pattern.

The structure of the package itself is also an important point to emphasise: It follows the principle of the bridge architectural pattern. This means that the Stepper class and the Scheduler class are only abstractions. The Simulation class holds an implementation of the Stepper class which performs stepper methods. What these methods do fully depends on the implementation of the stepper. In our case the implementation of stepper is a SchedulingStepper which steps according to a scheduler (which is also only an abstraction). It is obvious that this structure is extremely open to expansion. If a developer in the future wants the stepper to step differently he/she just writes an implementation of the Stepper class and passes it to the Simulation. The new stepper implementation now performs totally different actions. The same applies for the Scheduler.

5.4.1 class Simulation

`-stepperActivated : boolean`
Contains the information whether the simulation is active or not. By default this attribute is set to false. When the activateStepper() method is called, the attribute is set to true. When the deactivateStepper() method is called, the attribute is set to false.

`+«create» Simulation(config : SimulationConfiguration)`
Creates a new simulation with a specific SimulationConfiguration.

`+start() : void`
Starts the simulation of the current flowchart and the stepper is being activated.

`+stop() : void`
Stops the simulation of the current flowchart. The data of the modules are set to the default data values and the stepper gets deactivated.

`+pause() : void`
Pauses the simulation of the current flowchart. The data of the modules do not change, they will not be processed further, i.e. the stepper is not told to step any more.

`+isRunning() : boolean`
Returns the state of the activation of the stepper.

`+setConfig(config : SimulationConfiguration) : void`
Sets a new configuration for the simulation.

`+getConfig() : SimulationConfiguration`
Returns the current SimulationConfiguration of the simulation.

`+activateStepper() : void`
Activates the stepper of the simulation.

`+deactivateStepper() : void`
Deactivates the stepper of the simulation.

`+stepForward() : void`
Perform one step forward with the stepper of the simulation and the stepperTime given in SimulationConfiguration.

5.4.2 class SimulationConfiguration

`-stepperTime : Time`
Contains the time the stepper steps performing one step at a time.

`-timeout : Time<Real>`
Contains the maximal time modules are allowed to run in a step.

`+«create» SimulationConfiguration(stepperTime : Time, timeout : Time<Real>)`
Creates a new SimulationConfiguration with a stepperTime and a timeout.

`+getStepperTime() : Time`
Returns the current stepperTime.

`+setStepperTime(time : Time) : void`
Sets a new stepperTime.

`+getTimeout() : Time<Real>`
Returns the current timeout for a module. This timeout is the same for all modules.

`+setTimeout(time : Time<Real>) : void`
Sets a new timeout for all modules. The timeout is there only to prevent modules from running forever. It should not be set to low.

5.4.3 interface OctaneStepper

`+performSimulationStep(time : Time) : void`
Performs one simulation step in the current simulation with a given time.

5.4.4 abstract class Stepper

`+«abstract» stepForward(time : Time): void`
Performs one step forward with a given time.

5.4.5 class SchedulingStepper

`+performSimulationStep(time : Time) : void`
Performs one simulation step in the current simulation with a given time. Executes the modules given from the scheduler in the given order.

`+stepForward(time : Time) : void`
Performs one step forward with a given time. Executes the modules given from the scheduler in the given order.

5.4.6 abstract class Scheduler

`+getSchedule(modules : List<Module>) : List<Module>`
Returns a list of modules containing the given modules in the order of execution.

5.4.7 class PriorityScheduler

`+getSchedule(modules : List<Module>) : List<Module>`
Returns a list of modules containing the given modules in the order of execution. The order of execution is determined by the priority.

5.4.8 Scheduler detailed specification

The selection of modules to run by our scheduler is quite complex. Therefore we implemented a method in pseudocode to explain it. The function of the method is to sort the modules that want to run in a specific step so that the modules with the highest priority get the newest data. The algorithm is a modified Depth-First-Search which runs through every module and searches its dependencies and calls them to provide new data. The first step the algorithm does it to sort the list of modules it gets by priority. Then it does a recursion on each module in the list starting with the highest priority.

```
//notice: List<Module*> list get sorted by priority(highest priority first) with: sort(list);
//notice: modules contains all modules which would like to run in this step

public List<Module*> getStepSchedule(List<Module*> modules) {
  List<Module> schedule;
  sort(modules);

  while(Module temp : modules) {
    if(!marked(temp)) { //checks whether the module is marked
      List<Module> dependenciesOrdered = getOrder(modules, temp);   // implicitly a BFS is done here
      schedule.addAll(dependencies); //adding all the dependencies in correct order to the list
    }
  }

  return schedule;
}


private List<Module*> getOrder(List<Module*> modules, Module* root)
{
  //base case
  if(!modules.contains(root) || marked(root)) {
    return List<Module*>; //return empty list
  }

  //recursive case
  List<Module*> ordered;

  ordered.add(root);

  //marks the module (let the module remain marked, also after this helper method returns)
  mark(root);

  List<Module*> deps = module.getDependencies();
  sort(dep);
  reverse(dep);
  while (depModule : deps)
  {
    List<Module*> addModules = getOrder(modules, depModule);
    ordered.spliceFront(addModules); // add recursively found list to the front of the ordered list
  }

  return ordered;
}
```

Figure 16: PriorityScheduler getStepSchedule (pseudocode)

The recursive method “getOrder()” first checks if the module it is given wants to be run in this step or if it was already added to the schedule. If this is the case an empty list is returned. In the other case the method creates a list and adds the module. Then it gets the modules dependencies by calling itself on each dependency and adding them to the front of the list. This is also the reason why we have to invert the list to ensure that dependencies that are more important are run first.

5.4.9 Simulation Sequence

The simulation is externally started with the method “start()”. The Simulation class then determines the times it is supposed to run by calling its SimulationConfiguration with “getPlayTime()”. Afterwards it tells its stepper to “stepForward” the determined time. The stepper then steps as long until the time has run up. Each step it performs follows the same principle: First the stepper calls the scheduler with “getSchedule()” to create a schedule of the modules that want to be run in the next step. The scheduler creates the schedule with the help of a Depth-First-Search which determines the order in which the modules will be run (this is further explained in section 5.4.8). After the schedule is created and passed to the stepper, the stepper calls its method “run()” which runs the modules in the schedule in the order specified. This process stops after the time specified has run up or an fault occurred.

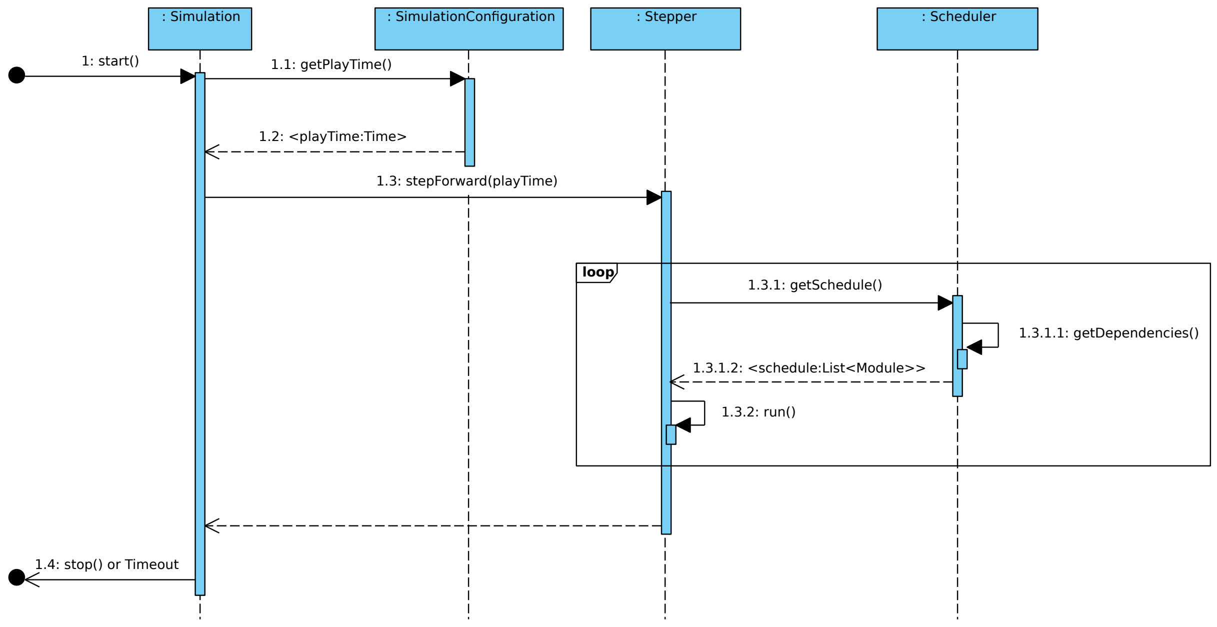
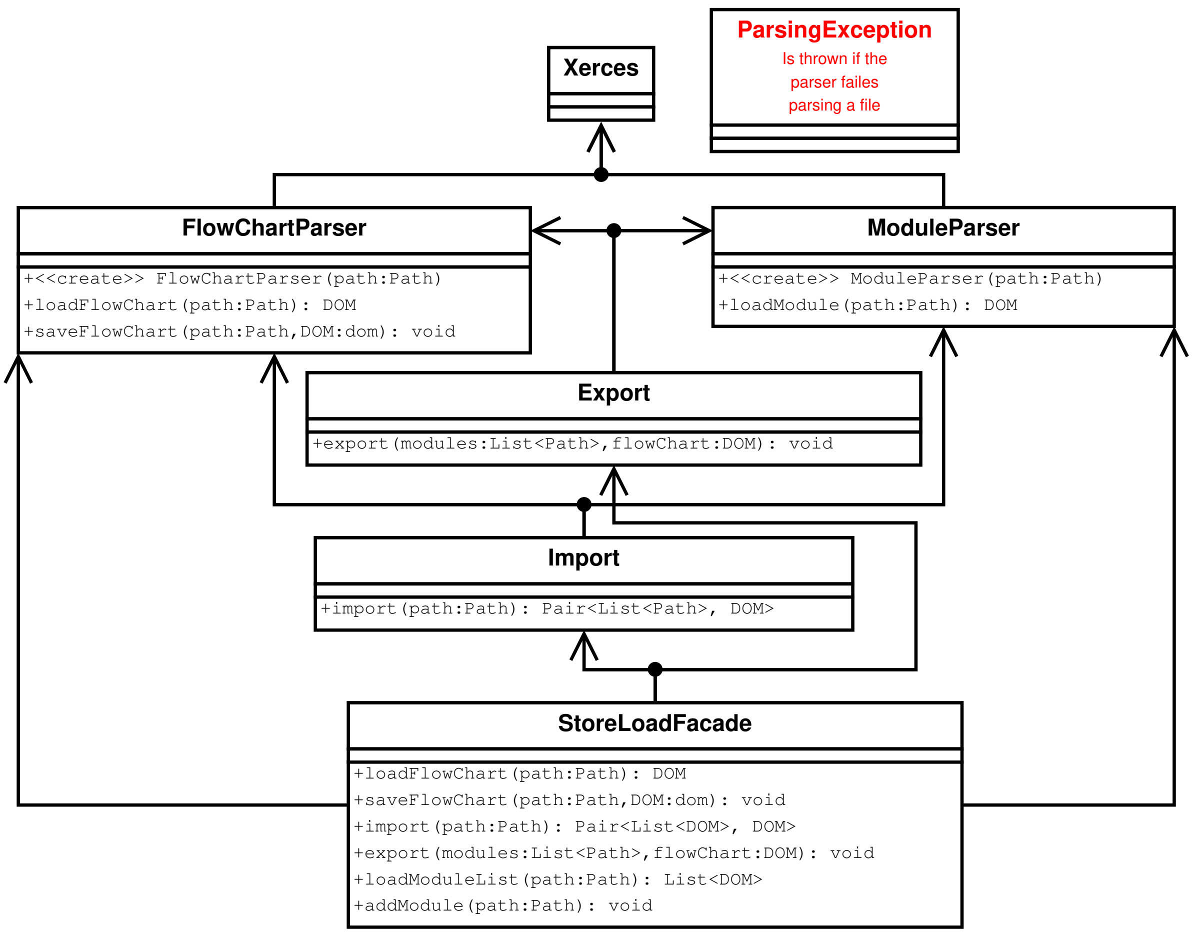


Figure 17: Simulation Start Sequence - This sequence diagram shows the flow of an simulation step.

It is important to mention that this sequence belongs to our implementation of the Stepper and Scheduler class. Also the call of the scheduler does not have to be in every step. It can be possible to tell another scheduler to calculate multiple schedules beforehand to improve runtime or to implement further scheduling features.

5.5 Store and Load



Xerces
ParsingException
Is thrown if the parser failes parsing a file
FlowChartParser
+<<create>> FlowChartParser(path:Path)
+loadFlowChart(path:Path): DOM
+saveFlowChart(path:Path,DOM:dom): void
ModuleParser
+<<create>> ModuleParser(path:Path)
+loadModule(path:Path): DOM
Export
+export(modules:List<Path>,flowChart:DOM): void
Import
+import(path:Path): Pair<List<Path>, DOM>
StoreLoadFacade
+loadFlowChart(path:Path): DOM
+saveFlowChart(path:Path,DOM:dom): void
+import(path:Path): Pair<List<DOM>, DOM>
+export(modules:List<Path>,flowChart:DOM): void
+loadModuleList(path:Path): List<DOM>
+addModule(path:Path): void

Figure 18: Store and Load

Storing and loading of modules and flowcharts is processed in this package. The offered functions can be accessed through the StoreLoadFacade and its methods. Parsing of XML-Documents to DOMs (DocumentObjectModel) is done by the parsing library Xerces, which supports Format-Files to check the correctness of the documents. Storing and loading of modules and flowcharts are offered by different parser classes. Import and export are combinations of module- and flowchart operations followed by a compression. The ParsingException is used to show incorrect XML-Documents. The ModuleList containing all different modules can be loaded and a module can be added to this list by copying it to the right folder and reloading the module list.

5.5.1 Formal of modules and dynamic loading

The modules are stored in a folder containing XML-Documents with all data required for the module. This includes a path to the code (the class inheriting our abstract classes, e.g. Module), which should be in the same folder. The code will be loaded by dynamic linking at runtime (if implementation of this feature succeeds, if not code will be linked beforehand).

5.5.2 class StoreLoadFacade

The StoreLoadFacade summarizes the functions of the package and offers them to the program.

`+loadFlowChart(path : Path) : DOM`
Loads a given XML-Document and tries to parse it with a FlowChartParser.

`+saveFlowChart(path : Path, dom : DOM) : void`
Saves the DOM of a flowchart at the given position with a FlowChartParser.

`+import(path : Path) : Pair<List<Path>, DOM>`
Decompresses a compressed folder containing modules and a flowchart. The modules will be copied in the module directory of the program and the flowchart will be loaded in the View.

`+export(modules : List<Path>, flowChart : DOM) : DOM`
Saves the current flowchart with additional modules in a folder, which gets compressed.

`+loadModuleList(path : Path) : List<DOM>`
Loads the modules from the given path with a ModuleParser.

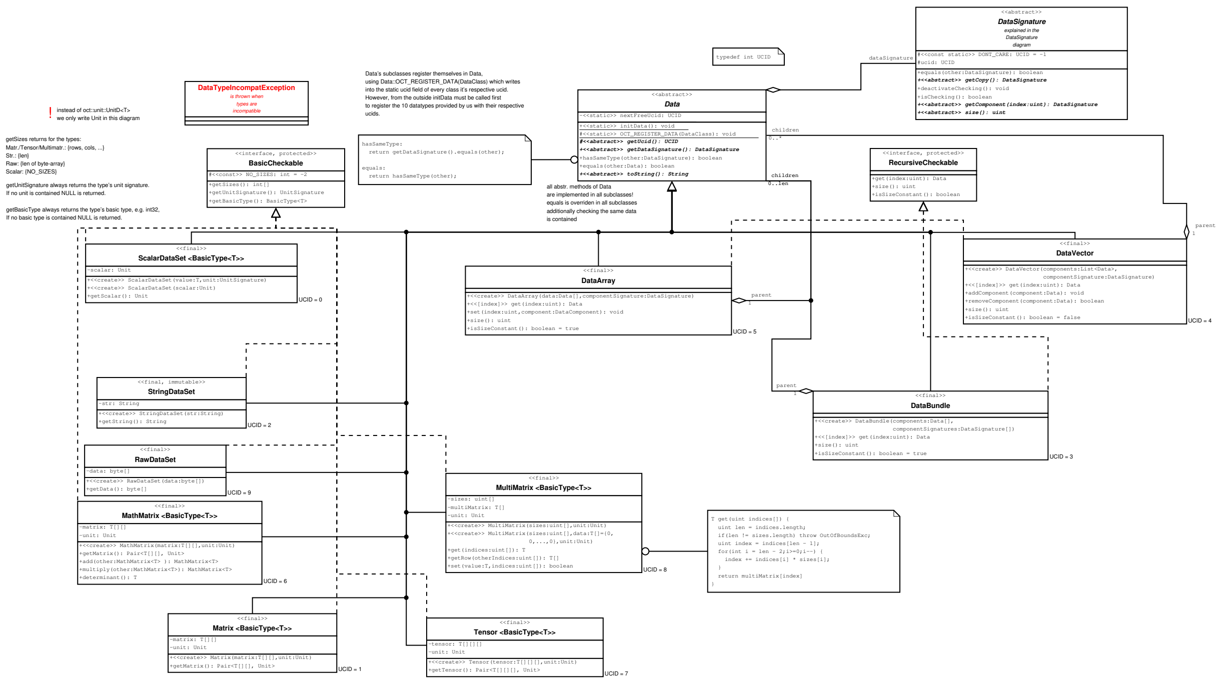
`+addModule(path : Path) : void`
Copies the given module in the module folder.

5.6 Data types

The Data package which is shown in figure 19 contains all possible data types which can be send via connections and processed by modules. Each type can wrap data, so that sending, receiving and processing data is easy. Every data type is identified with an unique class ID (UCID) and a DataSignature. However, the world only knows the DataSignature whereas the UCID is only visible within the Data package. The data signature is further explained in figure 20.

Fundamentally, there are two data types:

- Basic Types which can have sizes (e.g. rows and columns), a BasicType and a Unit
- Recursive Types which are a recursive container for other Data (other types)

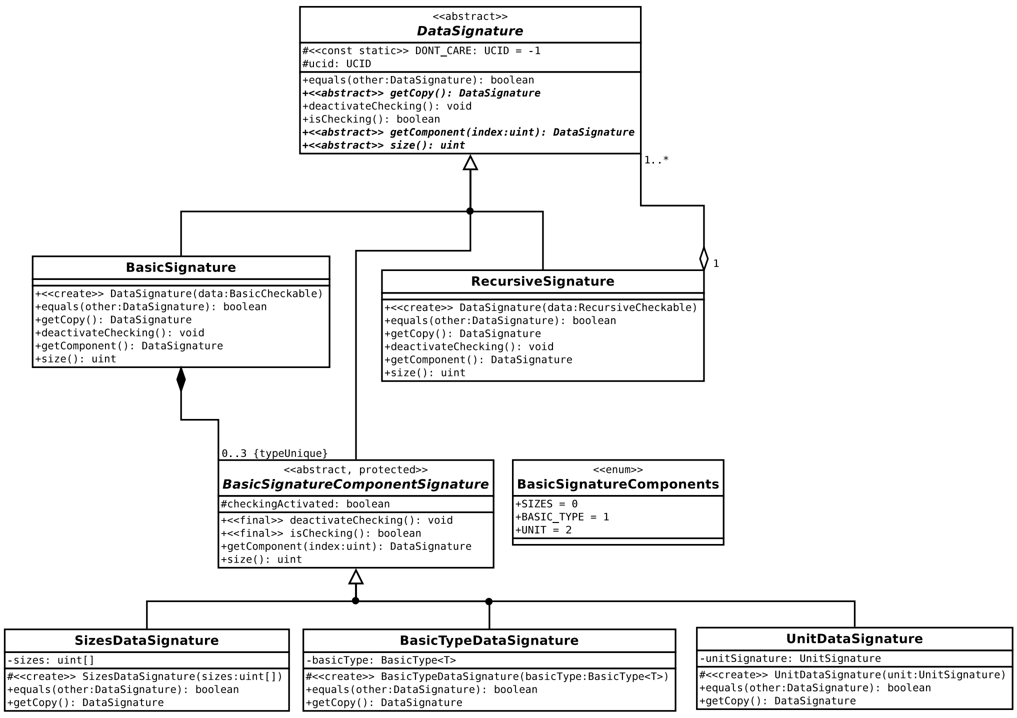


<<abstract>>
DataSignature
explained in the DataSignature diagram
#<<const static>> DONT_CARE: UCID = -1
#ucid: UCID
+equals(other:DataSignature): boolean
+<<abstract>> getCopy(): DataSignature
+deactivateChecking(): void
+isChecking(): boolean
+<<abstract>> getComponent(index:uint): DataSignature
+<<abstract>> size(): uint
typedef int UCID
dataSignature
Data's subclasses register themselves in Data, using Data::OCT_REGISTER_DATA(DataClass) which writes into the static ucid field of every class it's respective ucid. However, from the outside initData must be called first to register the 10 datatypes provided by us with their respective ucids.
DataTypeIncompatException
is thrown when types are incompatible
! instead of oct::unit::UnitD<T> we only write Unit in this diagram
getSizes returns for the types:
Matr./Tensor/Multimatr.: {rows, cols, ...}
Str.: {len}
Raw: {len of byte-array}
Scalar: {NO_SIZES}
getUnitSignature always returns the type's unit signature. If no unit is contained NULL is returned.
getBasicType always returns the type's basic type, e.g. int32, If no basic type is contained NULL is returned.
<<interface, protected>>
BasicCheckable
#<<const>> NO_SIZES: int = -2
+getSizes(): int[]
+getUnitSignature(): UnitSignature
+getBasicType(): BasicType<T>
<<abstract>>
Data
-<<static>> nextFreeUcid: UCID
+<<static>> initData(): void
#<<static>> OCT_REGISTER_DATA(DataClass): void
#<<abstract>> getUcid(): UCID
+<<abstract>> getDataSignature(): DataSignature
+hasSameType(other:DataSignature): boolean
+equals(other:Data): boolean
+<<abstract>> toString(): String
all abstr. methods of Data are implemented in all subclasses! equals is overriden in all subclasses additionally checking the same data is contained
children
0..*
children
0..len
<<interface, protected>>
RecursiveCheckable
+get(index:uint): Data
+size(): uint
+isSizeConstant(): boolean
parent
1
<<final>>
DataVector
+<<create>> DataVector(components:List<Data>, componentSignature:DataSignature)
+<<[index]>> get(index:uint): Data
+addComponent(component:Data): void
+removeComponent(component:Data): boolean
+size(): uint
+isSizeConstant(): boolean = false
UCID = 4
<<final>>
ScalarDataSet <BasicType<T>>
-scalar: Unit
+<<create>> ScalarDataSet(value:T,unit:UnitSignature)
+<<create>> ScalarDataSet(scalar:Unit)
+getScalar(): Unit
UCID = 0
<<final>>
DataArray
+<<create>> DataArray(data:Data[],componentSignature:DataSignature)
+<<[index]>> get(index:uint): Data
+set(index:uint,component:DataComponent): void
+size(): uint
+isSizeConstant(): boolean = true
UCID = 5
parent
1
<<final, immutable>>
StringDataSet
-str: String
+<<create>> StringDataSet(str:String)
+getString(): String
UCID = 2
parent
1
<<final>>
DataBundle
+<<create>> DataBundle(components:Data[], componentSignatures:DataSignature[])
+<<[index]>> get(index:uint): Data
+size(): uint
+isSizeConstant(): boolean = true
UCID = 3
<<final>>
RawDataSet
-data: byte[]
+<<create>> RawDataSet(data:byte[])
+getData(): byte[]
UCID = 9
<<final>>
MultiMatrix <BasicType<T>>
-sizes: uint[]
-multiMatrix: T[]
-unit: Unit
+<<create>> MultiMatrix(sizes:uint[],unit:Unit)
+<<create>> MultiMatrix(sizes:uint[],data:T[]={0, 0,...,0},unit:Unit)
+get(indices:uint[]): T
+getRow(otherIndices:uint[]): T[]
+set(value:T,indices:uint[]): boolean
UCID = 8
<<final>>
MathMatrix <BasicType<T>>
-matrix: T[][]
-unit: Unit
+<<create>> MathMatrix(matrix:T[][],unit:Unit)
+getMatrix(): Pair<T[][], Unit>
+add(other:MathMatrix<T>): MathMatrix<T>
+multiply(other:MathMatrix<T>): MathMatrix<T>
+determinant(): T
UCID = 6
<<final>>
Matrix <BasicType<T>>
-matrix: T[][]
-unit: Unit
+<<create>> Matrix(matrix:T[][],unit:Unit)
+getMatrix(): Pair<T[][], Unit>
UCID = 1
<<final>>
Tensor <BasicType<T>>
-tensor: T[][][]
-unit: Unit
+<<create>> Tensor(tensor:T[][][],unit:Unit)
+getTensor(): Pair<T[][][], Unit>
UCID = 7

```
hasSameType:
  return getDataSignature().equals(other);

equals:
  return hasSameType(other);
```

```
T get(uint indices[]) {
  uint len = indices.length;
  if(len != sizes.length) throw OutOfBoundsExc;
  uint index = indices[len - 1];
  for(int i = len - 2;i>=0;i--) {
    index += indices[i] * sizes[i];
  }
  return multiMatrix[index]
}
```

Figure 19: Data package - This class diagram the Data package containing all classes having to do with Data.



<<abstract>>
DataSignature
#<<const static>> DONT_CARE: UCID = -1
#ucid: UCID
+equals(other:DataSignature): boolean
+<<abstract>> getCopy(): DataSignature
+deactivateChecking(): void
+isChecking(): boolean
+<<abstract>> getComponent(index:uint): DataSignature
+<<abstract>> size(): uint
1..*
1
BasicSignature
+<<create>> DataSignature(data:BasicCheckable)
+equals(other:DataSignature): boolean
+getCopy(): DataSignature
+deactivateChecking(): void
+getComponent(): DataSignature
+size(): uint
RecursiveSignature
+<<create>> DataSignature(data:RecursiveCheckable)
+equals(other:DataSignature): boolean
+getCopy(): DataSignature
+deactivateChecking(): void
+getComponent(): DataSignature
+size(): uint
0..3 {typeUnique}
<<abstract, protected>>
BasicSignatureComponentSignature
#checkingActivated: boolean
+<<final>> deactivateChecking(): void
+<<final>> isChecking(): boolean
+getComponent(index:uint): DataSignature
+size(): uint
<<enum>>
BasicSignatureComponents
+SIZES = 0
+BASIC_TYPE = 1
+UNIT = 2
SizesDataSignature
-sizes: uint[]
#<<create>> SizesDataSignature(sizes:uint[])
+equals(other:DataSignature): boolean
+getCopy(): DataSignature
BasicTypeDataSignature
-basicType: BasicType<T>
#<<create>> BasicTypeDataSignature(basicType:BasicType<T>)
+equals(other:DataSignature): boolean
+getCopy(): DataSignature
UnitDataSignature
-unitSignature: UnitSignature
#<<create>> UnitDataSignature(unit:UnitSignature)
+equals(other:DataSignature): boolean
+getCopy(): DataSignature

Figure 20: Data signature - This class diagram further describes the data signature part of the Data package explaining what a data signature is and what types of signatures exist.

5.6.1 abstract class Data

`+«static» initData() : void`
Initializes the Data package by registering all the subclasses of Data provided in this package with their respective UCIDs.

`+hasSameType(other : DataSignature) : boolean`
Checks whether this Data instance's DataSignature is equal to the specified other DataSignature.

`+equals(other : Data) : boolean`
Checks whether this Data instance is exactly equal to the other, i.e. exact type equality and all contained data is equal. However, if one ore more checkers are deactivated in either this instance's or the other's DataSignature false is returned, even if the specified other data and this instance are equal. So this method should only be used with fully specified types.

`+«abstract» toString() : String`
Returns a String representation of the data type.

5.6.2 abstract class DataSignature

`+equals(other : DataSignature) : boolean`
Checks whether this DataSignature is to some extend equal to the other one. The check is performed recursively until the whole implicit type tree is checked. However, the tree can be cut at almost any point by deactivating checking. Recursive types which are way too interlaced may lead to a Stack overflow, in this case an Exception is thrown saying "type to interlaced".

`+«abstract» getCopy() : DataSignature`
Returns an exact copy of this data signature.

`+deactivateChecking() : void`
Cuts the tree here, i.e. this signature and all its components cannot be checked ever again.

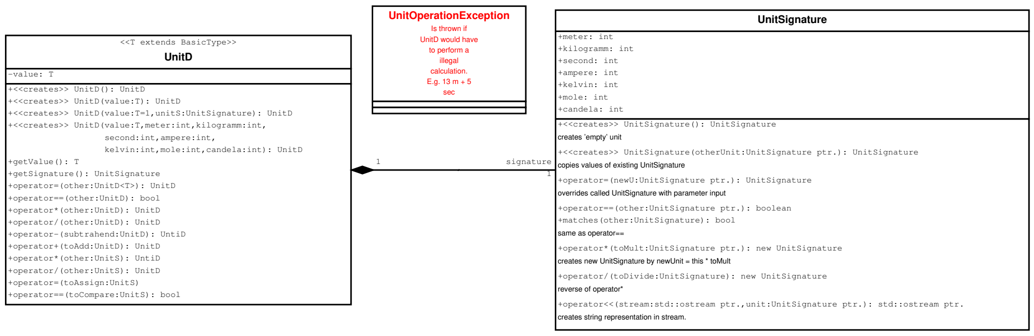
`+isChecking() : boolean`
Returns whether this signature is a checking signature or if it is deactivated.

`+«abstract» getComponent(index : uint) : DataSignature`
Returns the component of this DataSignature at the specified index. If the element does not exist (index too large or other error) an IllegalArgumentExeception is thrown.

`+«abstract» size() : uint`
Returns the size of this DataSignature, i.e. the (maximum index of its components)+1.

5.7 Units

In our simulation-heavy program there has to be some way to certify that values' units match. In the real world, this is achieved by using the SI-unit system. To provide the same functionality in our simulation environment, the basic types of the SI-system are modelled. The basic model called "UnitSignature" is provided by OCTANE. To simplify its use cases we will implement a new class using the UnitSignature functionality: UnitD. The advantage of the UnitD class is, that it also stores the value where the UnitSignature would be attached to. So if you want to multiply two UnitD instances, it can automatically calculate the resulting UnitSignature and also process the multiplication of the two values given. UnitD extends a generic type because the value can be of any BasicType provided.



<<T extends BasicType>>
UnitD
-value: T
+<<creates>> UnitD(): UnitD
+<<creates>> UnitD(value:T): UnitD
+<<creates>> UnitD(value:T=1,unitS:UnitSignature): UnitD
+<<creates>> UnitD(value:T,meter:int,kilogramm:int, second:int,ampere:int, kelvin:int,mole:int,candela:int): UnitD
+getValue(): T
+getSignature(): UnitSignature
+operator=(other:UnitD<T>): UnitD
+operator==(other:UnitD): bool
+operator*(other:UnitD): UnitD
+operator/(other:UnitD): UnitD
+operator-(subtrahend:UnitD): UntiD
+operator+(toAdd:UnitD): UnitD
+operator*(other:UnitS): UntiD
+operator/(other:UnitS): UnitD
+operator=(toAssign:UnitS)
+operator==(toCompare:UnitS): bool
UnitOperationException
Is thrown if UnitD would have to perform a illegal calculation. E.g. 13 m + 5 sec
1
signature
1
UnitSignature
+meter: int
+kilogramm: int
+second: int
+ampere: int
+kelvin: int
+mole: int
+candela: int
+<<creates>> UnitSignature(): UnitSignature
creates 'empty' unit
+<<creates>> UnitSignature(otherUnit:UnitSignature ptr.): UnitSignature
copies values of existing UnitSignature
+operator=(newU:UnitSignature ptr.): UnitSignature
overrides called UnitSignature with parameter input
+operator==(other:UnitSignature ptr.): boolean
+matches(other:UnitSignature): bool
same as operator==
+operator*(toMult:UnitSignature ptr.): new UnitSignature
creates new UnitSignature by newUnit = this * toMult
+operator/(toDivide:UnitSignature): new UnitSignature
reverse of operator*
+operator<<(stream:std::ostream ptr.,unit:UnitSignature ptr.): std::ostream ptr.
creates string representation in stream.

Figure 21: Unit package UnitD class - This class diagram shows the UnitD and the UnitSignature class.

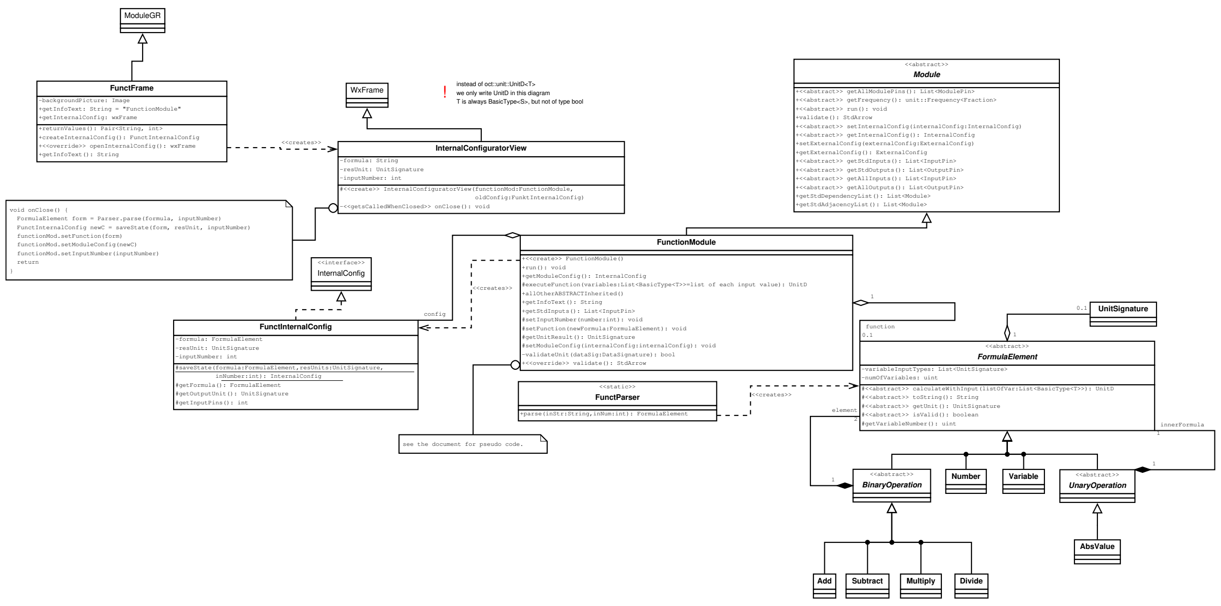
5.7.1 class UnitD

Generally this class overrides common operators, so that general operations can be used intuitively within a class.

`+getValue() : T`
Returns the value of the UnitD instance.

`+getSignature() : UnitSignature`
Returns the inner UnitSignature.

5.8 Function module


ModuleGR
FunctFrame
WxFrame
instead of oct::unit::UnitD<T> we only write UnitD in this diagram T is always BasicType<S>, but not of type bool
InternalConfiguratorView
<<abstract>> Module
FunctionModule
<<interface>> InternalConfig
FunctInternalConfig
<<static>> FunctParser
see the document for pseudo code.
UnitSignature
<<abstract>> FormulaElement
<<abstract>> BinaryOperation
Number
Variable
<<abstract>> UnaryOperation
AbsValue
Add
Subtract
Multiply
Divide

```
void onClose() {
  FormulaElement form = Parser.parse(formula, inputNumber)
  FunctInternalConfig newC = saveState(form, resUnit, inputNumber)
  functionMod.setFunction(form)
  functionMod.setModuleConfig(newC)
  functionMod.setInputNumber(inputNumber)
  return
}
```

Figure 22: Function Module - This class diagram shows the class structure of the included function module.

This module is not special by any means, it just implements the standard abstract Module as well as the ModuleGR. To function properly, the module has to implement a special parser, which converts Stings into functions with physical values. A wxFrame derived class implementation is also provided to be able to show the "configure module" window.

5.8.1 class FunctionElement

The function to be calculated is stored in a single FunctionElement instance. To be able to store complicated functions and also calculate them correctly, a composite design pattern is used. The advantages are that any legal formula in the real world (limited to our calculation operators) is also legal and displayable with this model. When the function is being calculated, each FunctionElement is calling the calculateWithInput(..) on its contained FunctionElement. This way the recursive structure is resolved from the leaf elements (being Numbers or Variables) to the top. On each stage the results of the lower levels are used to calculate the input to the current stage. E.g. Multiply takes the results of its two FunctionElements and multiplies their values and units together.

An interesting fact to note is, that there are no bracket elements needed, because the structure of the "FunctionElement-Tree" replaces the need to find the correct execution order. However, of course this means more work for the Parser.

Instantiation
This class can be created manually (for debugging mainly) or the whole structure of a formula can be created by the Parser class which is given a String.

5.8.2 Change Function Sequence

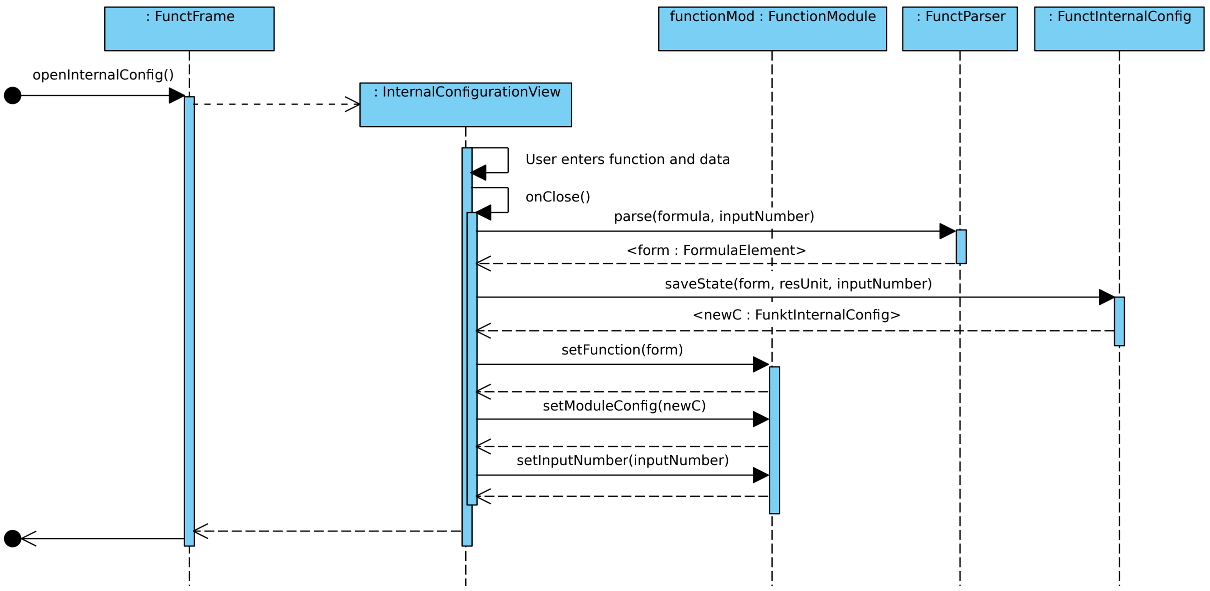

: FunctFrame
functionMod : FunctionModule
: FunctParser
: FunctInternalConfig
openInternalConfig()
: InternalConfigurationView
User enters function and data
onClose()
parse(formula, inputNumber)
<form : FormulaElement>
saveState(form, resUnit, inputNumber)
<newC : FunktInternalConfig>
setFunction(form)
setModuleConfig(newC)
setInputNumber(inputNumber)

Figure 23: Change Function Sequence - The sequence diagram shows the flow of actions of the user changing the function of the function module.

This section shows how a function is changed by the user and how the instances interact during that process. First the InternalConfiguration is opened “openInternalConfig()“ which creates an InternalConfigurationView which the user can edit. The user then enters the function he/she wants to be calculated by the module in the InternalConfigurationView. Then the user saves and closes the window which automatically calls the “onClose()“ function performing the following steps: First the FunctParser is called which parses the given information to return a FormulaElement describing the new formula entered. The FunctInternalConfig then creates a new FunctInternalConfig out of the new data. Afterwards the InternalConfigurationView calls three setters of the FunctionModule class to set the new function, configuration and input number.

5.8.3 Validation override

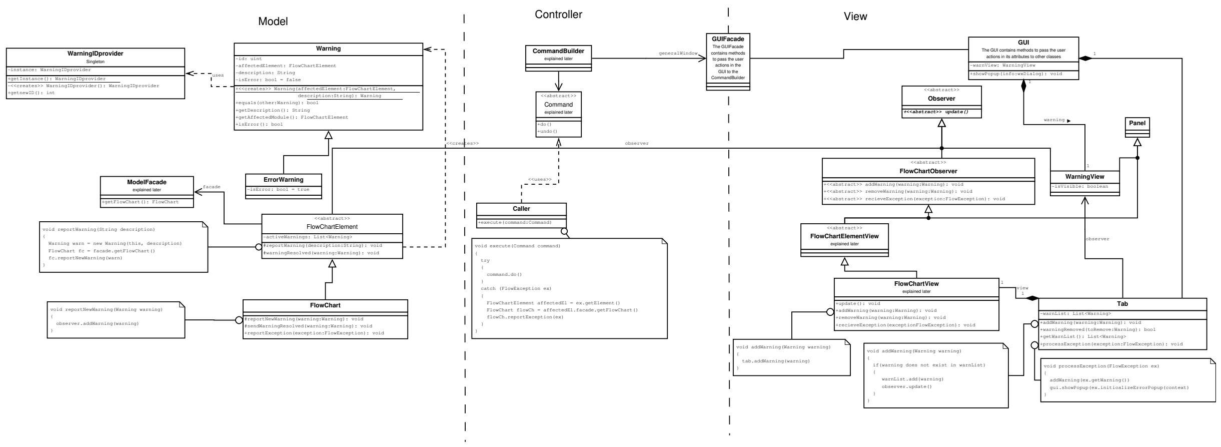
The override is necessary because the function module does not care, what the Units are of the inputs given. It does care however, how many inputs there are and whether the DataSignatures indicate that it is a scalar only. Equally important is that the given Units don’t contain don’t-cares any more. Because the function output would be inconsistent otherwise.
Exactly these requirements are being checked in code 24.

```
@Override
public StdArrow validate()
{
  List<InputPin> pins = getStdInputs();
  List<UnitD<BasicType<T>>> inputVars;

  for (ModulePin p : pins)
  {
    // this method checks, that any input is set to don't care, or if the dataType is not a Scalar.
    if(!validateUnits(p.getData().getDataSignature()))
      throw ValidationException("the module before has set invalid inputs.");
      //(units not acceptable)

    inputVars.add(((ScalarDataSet) p.getData()).getScalar())
  }// cast fault --> ValidationException


  if( function.getVariableNumber() != inputVars.size() )
    throw new ValidationException("expected #varNumbers of variables, but got #(inputVars)");

  List<BasicType<T>> varValues;
  List<UnitSignature> varSign;

  bool sameT = ...; //sameT is true exacty if: same basic type, e.g. only int32_t

  for(UnitD<BasicType<T>> var : inputVars) {
    if(!sameT) { //not all inputs same basic type --> calculating with double
      varValues.add((double) var.getValue());
    } else { //all inputs same basic type --> calculating with the given basic type
      varValues.add(var.getValue());
    }
    varSign.add(var.getSignature());
  }

    if(function.setExpectedUnits(varSign) == -1)
    throw new ValidationException("the module before has set invalid data.");
    //(units not acceptable) (e.g. contains dont-cares)

  UnitD res;
  try {
    UnitD res = function.calculateWithInput(varValues);
  } catch(UnitException exc) {
    //incompatible units --> ValidationException
    throw new ValidationException("the function-module could not validate because: "
            + exc.getCause());
  }

  out.getModulePinImp().setDefaultData(0 * res); //setting (0 * correct unit) as default data
  //putting the calculated data on the outgoing connection

  OutputPin out = (OutputPin) getStdOutputs().first();
  ScalarDataSet* data = new ScalarDataSet(res);
  out.putData(data);
  return NULL;
}
```

Figure 24: validate() in FunctionModule (pseudocode)

5.9 Warning system

It is often very hard to understand the processes underneath the GUI which are hidden from the user as it is in this application. There is always the need to prevent unexpected behaviour as well as overseeing some unset parameters. To prevent these things in our application we are implementing a warning system which will be described in this section. The warning system is integrated in the Model and can be interacted with from inside the main GUI.


Model
Controller
View
WarningIDprovider
Singleton
-instance: WarningIDprovider
+getInstance(): WarningIDprovider
-<<creates>> WarningIDprovider(): WarningIDprovider
+getnewID(): int
uses
Warning
-id: uint
-affectedElement: FlowChartElement
-description: String
-isError: bool = false
+<<creates>> Warning(affectedElement:FlowChartElement, description:String): Warning
+equals(other:Warning): bool
+getDescription(): String
+getAffectedModule(): FlowChartElement
+isError(): bool
<<creates>>
ErrorWarning
-isError: bool = true
ModelFacade
explained later
+getFlowChart(): FlowChart
facade
<<abstract>>
FlowChartElement
-activeWarnings: List<Warning>
#reportWarning(description:String): void
#warningResolved(warning:Warning): void
FlowChart
#reportNewWarning(warning:Warning): void
#sendWarningResolved(warning:Warning): void
+reportException(exception:FlowException): void
CommandBuilder
explained later
generalWindow
<<abstract>>
Command
explained later
+do()
+undo()
<<uses>>
Caller
+execute(command:Command)
observer
GUIFacade
The GUIFacade contains methods to pass the user actions in the GUI to the CommandBuilder
GUI
The GUI contains methods to pass the user actions in its attributes to other classes
-warnView: WarningView
+showPopup(info:wxDialog): void
1
<<abstract>>
Observer
+<<abstract>> update()
warning
Panel
<<abstract>>
FlowChartObserver
+<<abstract>> addWarning(warning:Warning): void
+<<abstract>> removeWarning(warning:Warning): void
+<<abstract>> recieveException(exception:FlowException): void
WarningView
-isVisible: boolean
observer
<<abstract>>
FlowChartElementView
explained later
FlowChartView
explained later
+update(): void
+addWarning(warning:Warning): void
+removeWarning(warning:Warning): void
+recieveException(exceptionFlowException): void
view
1
Tab
-warnList: List<Warning>
+addWarning(warning:Warning): void
+warningRemoved(toRemove:Warning): bool
+getWarnList(): List<Warning>
+processException(exception:FlowException): void

```
void reportWarning(String description)
{
  Warning warn = new Warning(this, description)
  FlowChart fc = facade.getFlowChart()
  fc.reportNewWarning(warn)
}
```

```
void reportNewWarning(Warning warning)
{
  observer.addWarning(warning)
}
```

```
void execute(Command command)
{
  try
  {
    command.do()
  }
  catch (FlowException ex)
  {
    FlowChartElement affectedEl = ex.getElement()
    FlowChart flowCh = affectedEl.facade.getFlowChart()
    flowCh.reportException(ex)
  }
}
```

```
void addWarning(Warning warning)
{
  tab.addWarning(warning)
}
```

```
void addWarning(Warning warning)
{
  if(warning does not exist in warnList)
  {
    warnList.add(warning)
    observer.update()
  }
}
```

```
void processException(FlowException ex)
{
  addWarning(ex.getWarning())
  gui.showPopup(ex.initioalizeErrorPopup(context)
}
```

Figure 25: Warning System - This class diagram shows an overview of all related and involved classes of the warning system

The warnings are stored as instances of Warning. They are stored in each FlowChartElement individually and are additionally summarized in the Tab class. Consistency is provided by passing the warnings (adding and removing) through the FlowChartView to Tab. WarningView extends from the Observer class, allowing it to get notified by Tab when a warning gets removed or added. The user can see those warnings by clicking on the “warnings”-button in the main GUI. After clicking on it, a new panel will appear and overlap the current FlowChartView. Only the active tab is allowed to display its warnings there. You can also click on each warning which results in closing of WarningView and jumping to the addressed warning in the FlowChartView itself.

To summarize: This functionality works on any FlowChartElement object natively. So implementing the behaviour into your own modules is simply achieved by calling the reportWarning(...) and warningResolved(...) methods. This can be used to shout out that a frequency is set too low, or maybe the output is not connected yet.

5.10 Error handling

In our software, the error handling is done via exceptions. Various exceptions can be thrown in the Model. However, exceptions should not be very common. Nevertheless, they all extend FlowException and are caught in Caller. If there is no Caller then there is probably no GUI at all. Meaning that the exceptions are thrown directly to the calling program. In figure 26 you can find an overview of all exceptions, that can be thrown.

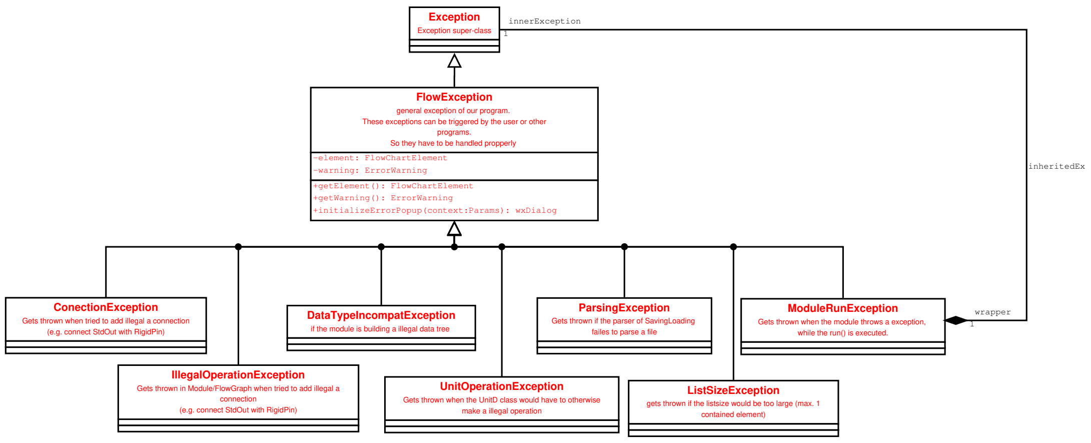

Exception
Exception super-class
innerException
1
inheritedEx
FlowException
general exception of our program.
These exceptions can be triggered by the user or other programs.
So they have to be handled propperly
-element: FlowChartElement
-warning: ErrorWarning
+getElement(): FlowChartElement
+getWarning(): ErrorWarning
+initializeErrorPopup(context:Params): wxDialog
ConectionException
Gets thrown when tried to add illegal a connection (e.g. connect StdOut with RigidPin)
DataTypeIncompatException
if the module is building a illegal data tree
ParsingException
Gets thrown if the parser of SavingLoading failes to parse a file
ModuleRunException
Gets thrown when the module throws a exception, while the run() is executed.
wrapper
1
IllegalOperationException
Gets thrown in Module/FlowGraph when tried to add illegal a connection (e.g. connect StdOut with RigidPin)
UnitOperationException
Gets thrown when the UnitD class would have to otherwise make a illegal operation
ListSizeException
gets thrown if the listsize would be too large (max. 1 contained element)

Figure 26: Exception Summary - The class diagram above shows all exceptions that can be thrown.

5.10.1 Handling Exceptions

The general principle is to only catch the exceptions that we cannot avoid, or recognize too late. When the exception is caught in Caller, he forwards the FlowException all the way to the respective Tab in a very similar fashion as in the warnings section described (see section 5.9, thus not further explained). In Tab the GUI is called to display a warning message for the user. An additional special Warning is placed as well.

6 References to the requirement specification

The following table contains all functional requirements from the requirement specification and a link to the position in this document where the function is implemented. Some of the wish features are not referenced to a section in this document. Maybe we will not implement those wish functionalities or only if there is some time left at the end of the implementation phase.

functional requirements	**implementation**
Module instances and modules	
F10 Inserting module instances in the flow graph	ModelFacade
F11 Selecting module instances in the flow graph	GUI Facade
F12 Deselecting module instances in the flow graph	GUI Facade
F13 Removing module instances in the flow graem	ModelFacade
F14 Moving module instances in the flow graph	Graphical Representation
F15 Turning module instances in the flow graph	Graphical Representation
F16 Scaling module instances in the flow graph	Graphical Representation
F17a Adjust inputs and outputs	Graphical Representation, FlowChart
F18a Providing a function module	Function Module
(W)F60 Providing a derivation module	-
(W)F90 Encapsulate a flow chart in a module	Graphical Representation
(W)F65 Providing a plotter module	-
Connections	
F20 Connecting module instances	ModelFacade
F21 Automatical routing of connections	Graphical Representation
F22 Adjusting the connection when rotating or moving a module instance	Graphical Representation
F23 Displaying data types on the connections	Graphical Representation
F24 Checking the data types on creation of connections	ModelFacade
F25 Providing rigid connections	Graphical Representation, Pins and Connections
F26 Providing arrows as a connection type	Graphical Representation, Pins and Connections
F27 Splitting arrows between two module instances	Graphical Representation, Pins and Connections
(W)F70 Connecting module instances by dragging	Controller -> CommandBuilder
(W)F71 Routing-algorithm for better routes	(Graphical Representation)
(W)F72 Filtering connection types	-

functional requirements	implementation
Simulation	
F30 Perform simulation cycle	Simulation
F31 Start simulation	Simulation
F32 Pause the simulation	Simulation
F33 Reset simulation	Simulation
(W)F110 Jumping back to previous states of the simulation	-
Store and Load	
F40 “Save” a flow chart	Store and Load
F41 “Save as” for a flow chart	Store and Load
F42 Loading a flow graph	Store and Load
F43 Adding a module to the module list	Store and Load
F44 Updating the module list	Store and Load
(W)F80 Importing a flow graph	Store and Load
(W)F81 Exporting a flow graph	Store and Load
(W)F100 Displaying a preview picture of the flow graph	-
Others	
F50 Info-tab of the module sidebar	ModuleInfoTabView
F51 Representation-tab of the module sidebar	ModuleGRTabView
F52 Module list sidebar	ModuleListView
(W)F120 Undo and redo	Global Design -> Caller
(W)F130 Search function in the module list sidebar	-
(W)F131 Search function for the flow graph	-
(W)F140 Move, zoom or center the view	-
(W)F141 Switch grid on and off	Graphical Representation
(W)F150 Warning button/ warning window	Warning System
(W)F170 Label elements	Graphical Representation