Design Document

Wavelength — λ-IDE

Muhammet Gümüs, Markus Himmel, Marc Huisinga,
Philip Klemens, Julia Schmid, Jean-Pierre von der Heydt

March 28, 2018

Contents

1 Introduction

This document contains the design for “Wavelength”, a web-based IDE for the untyped lambda calculus, as described in the *Pflichtenheft*.

The object-oriented architecture of the project roughly follows the model-view-controller (MVC) architectural pattern. It consists of two largely independent sets of packages: `model` and `view`. The architecture differs from MVC in the sense that there is no communication between views and models, all communication between the two passes through the respective controller. Additionally the controller is divided into two parts: `action` and `update`. `action` handles user inputs from the view, while `update` handles concurrent updates from other sources, for instance the model. At the center of the architecture is the `App` class, which initializes and holds the view. `action` and `update` both query `App` for the view elements it holds. As a result of this architecture, the entirety of the model is decoupled from the view and the rest of the application.

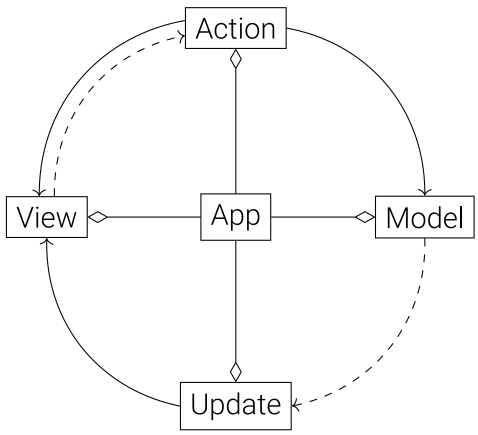


Figure 1: Sketch of the application architecture. Actions are indirectly called by the view and updates are indirectly called by the model. Actions interact with the view and control the model, while updates update the view. `App` initializes the application and provides access to the view to actions and updates.

1.1 The model

The first set of packages (`model` and its subpackages) comprises most of the model and provides means for interactively, but synchronously, operating on lambda terms. It is designed as a library that makes few assumptions about how it is used.

It provides a package with a very general set of classes representing `LambdaTerms` as a tree structure of immutable objects. For operations on these lambda terms, the visitor pattern is utilized. Since many operations performed on lambda terms fall into a few categories (for example displaying terms in some way or transforming a term into another term), some special abstract visitors are provided in order to make these common operations less cumbersome. For the most fundamental operations on lambda terms, concrete visitors are provided, but the `Visitor` interface may be implemented by any entity wishing to operate on lambda terms (for instance reduction orders or the user interface for displaying terms).

In addition to `LambdaTerm`s and visitors and the ability to parse strings into `LambdaTerm`s, the model provides interfaces for reduction orders, output sizes and libraries as well as the implementations of the concrete reductions orders, output sizes and libraries mentioned in the *Pflichtenheft*. These three concepts are each fully contained in separate packages which only use the package providing `LambdaTerm`s.

The central interface of the model to the user interface is the `Executor` wrapping the `ExecutionEngine` class, which manages the state of an interactive reduction of a lambda term. It keeps the current state and history of the execution and allows performing the next reduction step, either of a supplied redex or according to the currently set reduction order.

1.2 The user interface

The second set of packages (`view` and its subpackages) manages building the user interface (UI) and enabling UI elements to interact with the model.

In order to abstract the necessary aspects of UI elements the `view.api` package defines elementary interfaces. For a further narrowing of aspects the `view.webui.component` package wraps some of GWT's widgets using the adapter pattern.

Regarding user interaction the UI elements are able to run actions as specified in the `Action` interface when used (on click for example). Actions are supposed to cover an abundance of manipulations on other UI elements or even the model.

Since providing concurrency for the reduction of lambda terms is crucial for maintaining responsiveness of the UI the `view.execution` package provides a wrapper class for the `ExecutionEngine` class and an associated observer interface.

The communication between the view and the model is handled by the `App` class. It stores all available UI elements and an `Executor` (wrapping an `ExecutionEngine`) and provides access to them. This feature is essential for actions giving them easy access to resources. The general `view` package provides the `App` class but also a `URLSerializer` class for serializing a URL and the associated Observer class.

The `view.update` package provides concrete observers for serialization and execution along with visitors for resolving lambda terms according to different outputFormats.

The `view.export` package provides means of transforming the output to a specific Format.

Lastly there is an exercise mode available represented by the `view.exercise` package providing an interface, a concrete implementation and a static list of all exercises.

2 Packages and classes

2.1 Package `edu.kit.wavelength.client.database`

2.1.1 Interface `DatabaseServiceAsync`

Interface for asynchronous calls to the database according to specifications in `DatabaseService`.

Methods:

- `void getSerialization(String id, <any> callback)`

  Asynchronous call to method specified in `DatabaseServicegetSerialization(String)`.

  `id`: serialization's id

  `callback`: callback handler

- `void addEntry(String serialization, <any> callback)`

  Asynchronous call to method specified in `DatabaseServiceaddEntry(String)`.

  `serialization`: a valid serialization

  `callback`: callback handler

2.1.2 Interface `DatabaseService`

This interface provides methods for accessing and manipulating a SQLite database on the server. The database must provide a means of mapping unique ids to serialization Strings. The ids are generated by the database and returned upon creation of a new entry in the database.

Methods:

- `String getSerialization(String id)`

  Returns serialization belonging to given id if an entry exists, else returns `null`.

  `id`: unique id

  Returns: serialization belonging to given id

- `String addEntry(String serialization)`

  Adds an entry for the given serialization if it is not already in the database by generating an id and returns the assigned id. If the entry already is in the database returns the id assigned to the given serialization. Returns null if an error occurs. Note that the serialization is assumed to be valid.

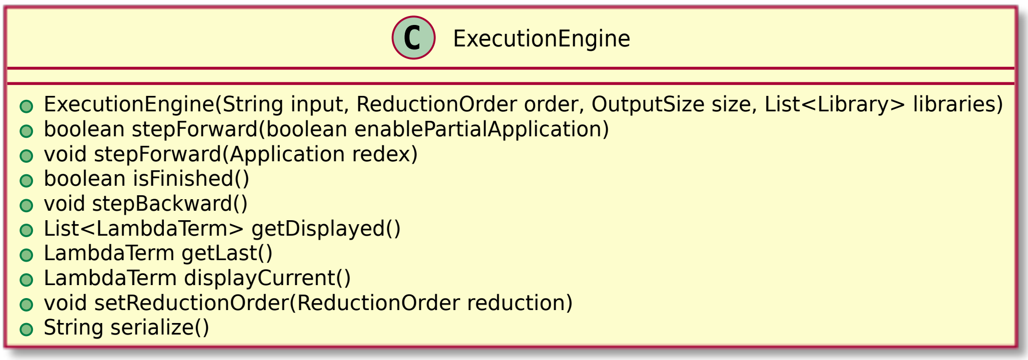
  `serialization`: valid serialization

  Returns: id mapped to given serialization

2.2 Package `edu.kit.wavelength.client.model`

The `model` package contains the `ExecutionEngine` class, which is used to reduce `LambdaTerm`s. It is initialized with an input string, as well as the reduction order (see `model.reduction`), output size (see `model.output`) and included libraries (see `model.library`) and can then be used to interactively operate on the lambda term, by stepping forward (that is, β-reducing according to the reduction order or reducing a provided redex in the current term) and backward (reverting to the previous term that was displayed).

The `ExecutionEngine` is fully synchronous and does not have a notion of “fully” reducing a term.



2.2.1 Class `ExecutionException`

Extends: `Exception`

Thrown when an error during an operation on a lambda term occurs.

Constructors:

- `ExecutionException(String message)`

2.2.2 Class `ExecutionEngine`

An execution engine manages the reduction of a `LambdaTerm`. It keeps the history of terms and which of these terms were displayed and is able to reduce the current term according to a `ReductionOrder` or reduce a specific redex in the current term. It also keeps track of which terms should be displayed and is able to revert to the previous displayed term.

Constructors:

- `ExecutionEngine(String input, ReductionOrder order, OutputSize size, List<Library> libraries)`

  Creates a new execution engine.

  `input`: The textual representation of a `LambdaTerm` to be handled

  `order`: The `ReductionOrder` to be used by default

  `size`: The `OutputSize` to be used

  `libraries`: The `Libraries` to be taken into consideration during parsing

- `ExecutionEngine(String serialized)`

  Instantiates a new ExecutionEngine from its serialization.

  `serialized`: A serialized ExecutionEngine

Methods:

- `List<Library> getLibraries()`

  Returns the libraries that have been used by the execution engine.

  Returns: The libraries that have been used by the execution engine

- `List<LambdaTerm> stepForward()`

  Executes a single reduction of the current `LambdaTerm`.

  Returns: The lambda terms that should be displayed as a result of this step

- `List<LambdaTerm> stepForward(Application redex)`

  Executes a single reduction of the supplied `Application`.

  `redex`: The `Application` to be evaluated. Must be a redex, otherwise an exception is thrown

- `boolean isFinished()`

  Determines whether the execution is finished according to the current `ReductionOrder`.

  Returns: `true` if the current `ReductionOrder` does not provide another redex, `false` otherwise

- `boolean canStepBackward()`

- `void stepBackward()`

  Reverts to the previously output `LambdaTerm`.

- `List<LambdaTerm> getDisplayed()`

  Returns a list of all `LambdaTerm`s that have been displayed.

  Returns: A list of all `LambdaTerm`s that have been displayed

- `boolean isCurrentDisplayed()`

- `LambdaTerm displayCurrent()`

  Displays the currently reduced `LambdaTerm`, adding it to the list of displayed `LambdaTerm`s.

  Returns: the current `LambdaTerm`

- `void setReductionOrder(ReductionOrder reduction)`

  Changes the active `ReductionOrder` to the entered one.

  `reduction`: The new `ReductionOrder`

- `void setOutputSize(OutputSize size)`

  Changes the active `OutputSize` to the entered one.

  In general, this will not be possible seamlessly. Instead, it will just apply to the future.

  `size`: The new output size

- `int getStepNumber()`

  Returns the number of the current step in the computation.

  Returns: The number of the current step in the computation

- `StringBuilder serialize()`

  Serializes the ExecutionEngine by serializing its current `OutputSize`, `ReductionOrder`, `Libraries` and the terms it holds.

  Returns: The ExecutionEngine's serialized String representation

2.3 Package `edu.kit.wavelength.client.model.library`

The `model.library` contains classes representing the different libraries provided by the application. Each library consists of a collection of commonly needed `LambdaTerm`s and corresponding names used to reference them. After enabling a library these names can be used in place of the longer and more difficult to work with terms.

The `Boolean` library contains the terms representing the boolean values "true" and "false".

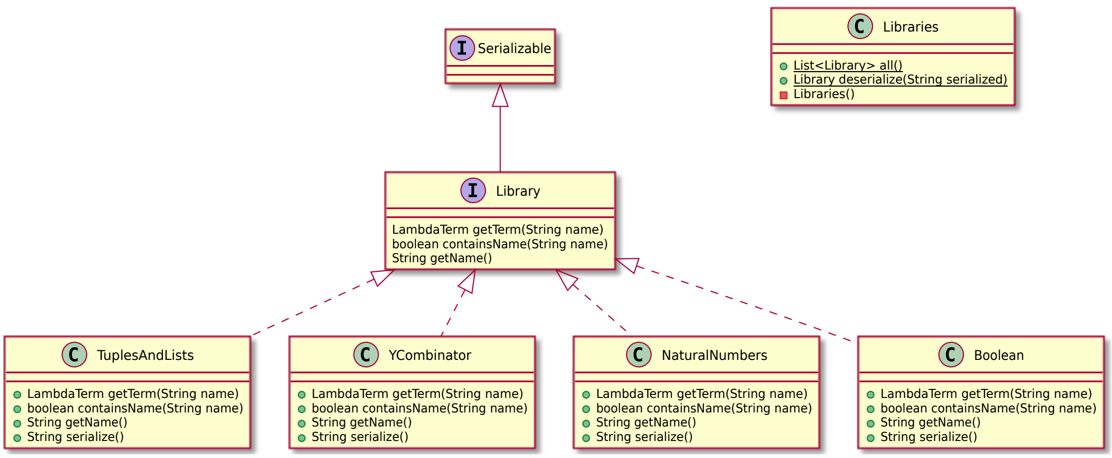
The `TuplesAndLists` library contains the terms implementing tuples and lists and the functions needed to work with them.

The `NaturalNumbers` library contains the church encodings of the natural numbers and terms implementing basic arithmic functions.

The `YCombinator` library contains the Y combinator making recursive function calls possible.

All libraries implement the `Library` interface allowing the application to be expanded with more libraries provided they too implement this interface.

The library classes are used by the `Parser` to access the terms belonging to names in the input String. After it has been successfully parsed each used library term is represented by a `NamedTerm` in the `LambdaTerm` object structure.



2.3.1 Class `YCombinator`

Implements: `Library`

A `Library` containing a definition of the Y combinator.

2.3.2 Class `TuplesAndLists`

Implements: `Library`

A `Library` contains definitions for `LambdaTermss` for tuples (pairs) and lists.

The terms are encoded by the Church encoding. The library contains tuples and the simple operations 'first' and 'second'. In addition this library supports lists. A list node is represented by a pair. The library also supports basic operations on lists.

2.3.3 Class `TermInfo`

Contains the metadata of a term provided by a library.

Constructors:

- `TermInfo(String name, String description)`

  Creates the meta data object for a term with the given name and description.

  `name`: Name of the term

  `description`: Description of what the term does

2.3.4 Class `NaturalNumbers`

Implements: `Library`

A `Library` matching integer literals to church numerals and providing functions for basic arithmetic operations.

Note that the `ExecutionEngine` can try to accelerate the calculation on a `LambdaTerm` that uses this library.

Constructors:

- `NaturalNumbers(boolean turbo)`

  Creates a new NaturalNumbers library.

  `turbo`: `true` if calculations on this Library should be accelerated and `false` otherwise

Static methods:

- `NaturalNumbers fromSerialized(String serialized)`

2.3.5 Interface `Library`

Extends: `Serializable`

This interface is used to interact with the different libraries provided by the application.

Each library contains a set of `LambdaTerm`s and their assigned names. These names can be used in place of terms to both shorten terms and make them easier to understand.

Methods:

- `LambdaTerm getTerm(String name)`

  Returns the `LambdaTerm` with the specified name.

  `name`: The name assigned to the desired term

  Returns: The `LambdaTerm` with the entered name, null if the library does not contain a term with this name

- `boolean containsName(String name)`

  Determines whether the library contains a `LambdaTerm` with the specified name.

  `name`: The name to search the library for

  Returns: `true` if the library contains a `LambdaTerm` with the entered name and `false` otherwise

- `String getName()`

  Returns the library's name.

  Returns: The name of the library

- `List<TermInfo> getTermInfos()`

  Returns info for the terms that this library provides.

  Returns: infos

2.3.6 Class `Libraries`

Static class giving access to all `Libraries` known to the model.

Static methods:

- `List<Library> all()`

  Returns an unmodifiable list of all `Libraries` known to the model.

  Returns: An unmodifiable list that contains exactly one instance of every `Library` known to the model

- `Library deserialize(String serialized)`

  Returns the `Library` referred to by the serialized string.

  `serialized`: A string created by calling serialize on a `Library` known to the Libraries class

  Returns: The `Library` referred to by the serialized string

2.3.7 Class `CustomLibrary`

Implements: `Library`

The CustomLibrary class represents a library which starts out empty, but to which any term and name can be added. It is used by the Parser to store all name assignments defined by the user.

Constructors:

- `CustomLibrary(String name)`

  Creates a new empty custom library.

  `name`: The name of the library

Static methods:

- `CustomLibrary fromSerialized(String serialized)`

  Restores a `CustomLibrary` object from a serialization String.

  `serialized`: The serialization from which to restore the library

  Returns: A CustomLibrary equal to the library processed to the input String

Methods:

- `void addTerm(LambdaTerm term, String name)`

  Adds a new lambda and its name to the library.

  `term`: The lambda term to add to the library

  `name`: The name used to reference the term.

2.3.8 Class `Boolean`

Implements: `Library`

This `Library` contains definitions of `LambdaTerms` for boolean logic.

The terms are encoded by the Church encoding. The library contains the values true and false as well as simple logical operations such as 'and', 'or', 'not' and an if-clause.

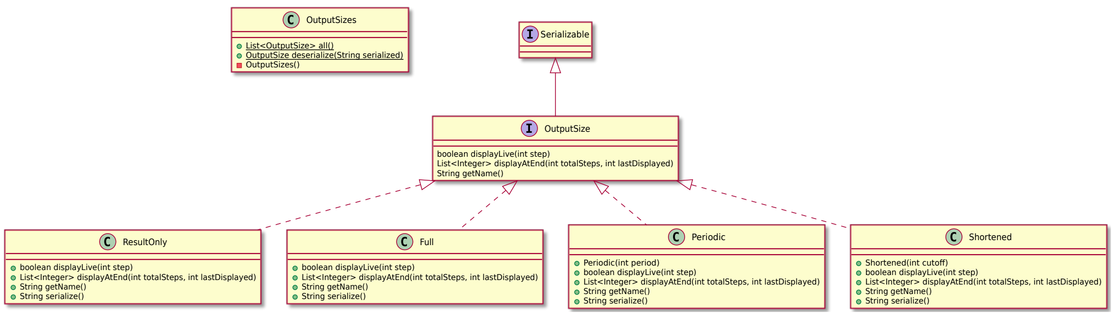
2.4 Package `edu.kit.wavelength.client.model.output`

The `model.output` package contains the `OutputSize` interface, all its implementations that the model provides and the `OutputSizes` static class.

`OutputSize` is implemented by policies that decide which lambda terms should be displayed to the user based on their step numbers. For deciding which terms should be displayed after execution has finished, the total number of steps and the number of the last term that was displayed while the execution was still running is taken into account.

The policies provided by the model show every term (`Full`), only every n-th term for some n (`Periodic`), only the first n and last n terms for some n (`Shortened`) or only the very last term (`ResultOnly`).

Similarly to the analogous classes in the `model.reduction` and `model.library` packages, `OutputSizes` provides instances of the aforementioned classes.



2.4.1 Class `Shortened`

Implements: `OutputSize`

`OutputSize` that only shows a certain number of terms at the beginning and at the end.

Constructors:

- `Shortened(int cutoff)`

  Creates a shortened output size policy with the given cutoff.

  `cutoff`: How many terms are to be shown at the beginning and the end

Static methods:

- `Shortened fromSerialized(String serialized)`

2.4.2 Class `ResultOnly`

Implements: `OutputSize`

`OutputSize` that displays no terms live and only displays the very last term in the end.

2.4.3 Class `Periodic`

Implements: `OutputSize`

`OutputSize` where every n-th term is displayed, for some n.

Constructors:

- `Periodic(int period)`

  Creates a periodic output size with the given period.

  `period`: The period of terms to be displayed

Static methods:

- `Periodic fromSerialized(String serialized)`

2.4.4 Class `OutputSizes`

Static class giving access to all `OutputSize`s known to the model.

Static methods:

- `List<OutputSize> all()`

  Returns an unmodifiable list of all `OutputSize`s known to the model.

  Returns: An unmodifiable list containing all `OutputSize`s known to the model

- `OutputSize deserialize(String serialized)`

  Returns the `OutputSize` referred to by a given string.

  `serialized`: The string to be deserialized

  Returns: The `OutputSize` that the given string represents, if known to the model

2.4.5 Interface `OutputSize`

Extends: `Serializable`

Policy to decide which `LambdaTerm`s should be displayed, both live and at the end of the computation.

Methods:

- `boolean displayLive(int step)`

  Decides whether the step with the given number should be displayed live.

  `step`: The step number to be considered

  Returns: Whether the given step should be displayed live

- `List<Integer> displayAtEnd(int totalSteps, int lastDisplayed)`

  Decides which steps should be displayed after the computation has ended.

  `totalSteps`: The total number of steps the computation took

  `lastDisplayed`: The last step that has been displayed, either according to a policy or through manual step by step execution

  Returns: A list of step numbers, in the order in which they should be displayed The step numbers may not be smaller than (totalSteps - numToPreserve + 1.)

- `int numToPreserve()`

  Returns the number of terms that should be preserved even if displayLive() returns false.

  Returns:

- `String getName()`

  Returns the name of the output size.

  Returns: The name of the output size

2.4.6 Class `Full`

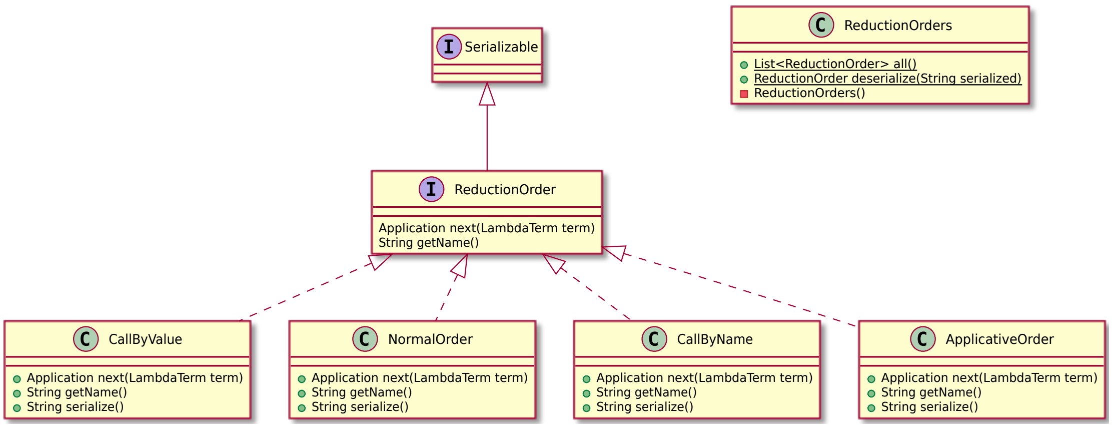
Implements: `OutputSize`

`OutputSize` where every term is displayed live.

2.5 Package `edu.kit.wavelength.client.model.reduction`

The `model.reduction` package contains the classes representing the lambda calculus' different reduction orders. Included in the application are the Applicative Order, Call-by-name, Call-by-value and Normal reduction orders.

Reduction orders are used by the `ExecutionEngine` during the reduction of a lambda term. The reduction orders do not reduce terms, instead each class only determines which redex is to be reduced next in accordance with the reduction order it represents. The actual beta-reduction has to be carried out by a `BetaReducer` visitor. All reduction orders implement the `ReductionOrder` interface, enabling the `ExecutionEngine` to interact with the active reduction order indiscriminately. This also makes it possible to expand the number of supported reduction orders by simply implementing the interface. Since the active reduction order is a part of the applications state the interface also defines a serialize method used in the generation of a String representing the applications state.



2.5.1 Class `ReductionOrders`

Static class giving access to all `ReductionOrders` known to the model.

Static methods:

- `List<ReductionOrder> all()`

  Returns an unmodifiable list of all `ReductionOrder`s known to the model.

  Returns: An unmodifiable list containing exactly one instance of all `ReductionOrder`s known to the model

- `ReductionOrder deserialize(String serialized)`

  Returns the `ReductionOrder` represented by a given string.

  `serialized`: A serialized reduction order

  Returns: The `ReductionOrder` the given string refers to, if known to the model

2.5.2 Interface `ReductionOrder`

Extends: `Serializable`

Represents a reduction order for the untyped lambda calculus. A reduction order is a policy to determine the next reducible expression (redex) to be evaluated.

Methods:

- `Application next(LambdaTerm term)`

  Determines the next redex to be evaluated according to the reduction order.

  `term`: The term whose next redex should be found

  Returns: `null` if there is no redex in the given term. Otherwise, the next redex to be evaluated.

- `String getName()`

  Returns the name of the reduction order, for example for display when selecting a reduction order in a user interface.

  Returns: The name of the reduction order

2.5.3 Class `NormalOrder`

Implements: `ReductionOrder`

The normal reduction order for the untyped lambda calculus.

The leftmost outermost redex is selected for reduction.

2.5.4 Class `CallByValue`

Implements: `ReductionOrder`

The call by value reduction order for the untyped lambda calculus.

The leftmost outermost redex that is not enclosed by an abstraction and whose argument is a value (i.e. an abstraction) is selected for reduction.

2.5.5 Class `CallByName`

Implements: `ReductionOrder`

The call by name reduction order for the untyped lambda calculus.

The leftmost outermost redex that is not enclosed by an abstraction is selected for reduction.

2.5.6 Class `ApplicativeOrder`

Implements: `ReductionOrder`

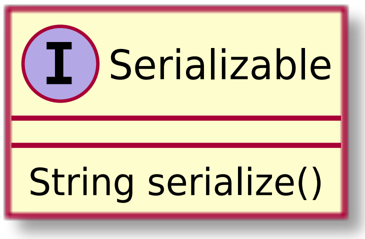
The applicative reduction order for the untyped lambda calculus.

The rightmost innermost redex is selected for reduction.

2.6 Package `edu.kit.wavelength.client.model.serialization`

The `model.serialization` package provides the `Serializable` interface. Classes implementing this interface may be serialized into a string.

Since serialization in Wavelength is ad hoc by design, no entity is intended to hold references to objects of type `Serializable`. It is provided merely to ensure consistent naming of the `serialize` method. Deserialization occurs in the classes providing the specific type (for example `LambdaTerm` or `Libraries` and its analogons in `model.reduction` and `model.output`), which can be done, again, due to the ad hoc nature of serialization, since each class knows which of its components it has serialized.



2.6.1 Class `SerializationUtilities`

Static class that provides tools for serializing and deserializing aggregate data.

Static methods:

- List<String> extract(String input)

  Deserializes a string that was produced by the enclose method into its components.

  `input`: The serialized string

  Returns: The components that were serialized

- `StringBuilder enclose(StringBuilder[] content)`

  Creates a compound serialized string from a set of serialized strings that can be deserialized with extract.

  `content`: The components

  Returns: A string that is a serializations of all component strings

- <T> List<T> deserializeList(String serialized, Function<String, T> deserialize)

  Deserializes a list created by serializeList.

  `serialized`: The serialization string created by serializeList

  `deserialize`: The deserialization method for a single list element

  Returns: A list of deserialized elements

- <T> StringBuilder serializeList(List<T> list, Function<T, StringBuilder> serialize)

  Serializes a list of elements of the same type.

  `list`: The list of elements

  `serialize`: The serialize method for a single element

  Returns: A serialization string for the entire list

2.6.2 Interface `Serializable`

Implemented by objects that may be serialized into a string.

Methods:

- `StringBuilder serialize()`

  Creates a string designating the object's state.

  Returns: A string designating the object's state from which it can be restored using a corresponding deserialization method

2.7 Package `edu.kit.wavelength.client.model.term`

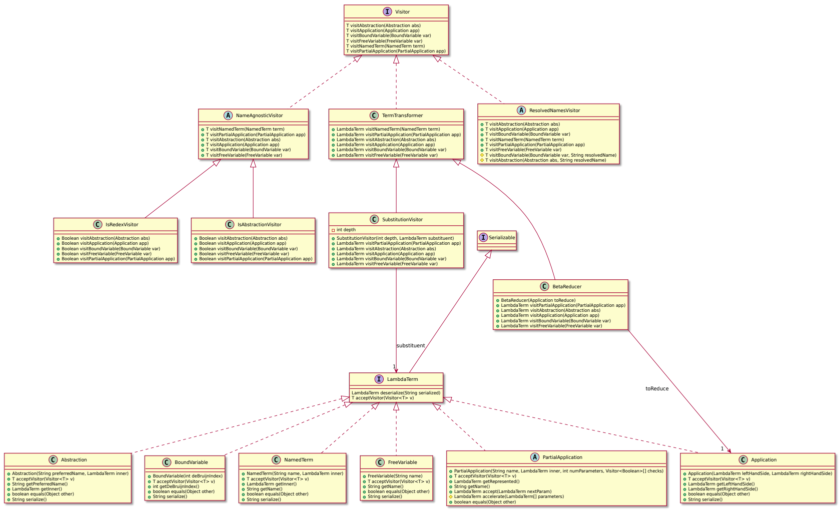
The `model.term` package contains the classes used by the application to represent lambda terms and provides means for operating on them.

Lambda terms are represented as a tree structure of immutable objects implementing the interface `LambdaTerm`. There are four types that directly model their corresponding entities in the untyped lambda calculus: `Abstraction`, `Application`, `BoundVariable` and `FreeVariable`. For abstractions, the preferred name for the variable is stored. Bound variables refer to their corresponding abstraction using De Bruijn indices. Additionally, there are two more types that do not have a direct counterpart in untyped lambda calculus: `NamedTerm` and `PartialApplication`.

`NamedTerm` is a lambda term that has been assigned a name, either by a `Library` or by the user. In some cases (for instance when considered by a reduction order polcy) it should be treated exactly like the term it represents. In other cases (for instance when displaying the term) it should be treated differently.

`PartialApplication` represents a series of nested applications, with the left hand side of the innermost application being a named term that stands for a library function (which is a series of nested abstractions). An application where the left hand side is a partial application can be transformed into a larger partial application. When the partial application has collected as many arguments as the library function its inner named term represents, it will be evaluated in one step and replaced by a lambda term for its result. The purpose of this is to be able to accelerate large computations involving numbers.

Except for testing for equality, all operations on lambda terms are carried out by visitors. In addition to the universal `Visitor<T>` interface, which is implemented by visitors carrying out an operation and returning a result of type `T`, there are several abstract classes implementing this interface to make typical operations on lambda terms easier. `NameAgnosticVisitor<T>` is a visitor that ignores the names of terms (such as reduction orders), `ResolvedNamesVisitor<T>` provides non-colliding names for variable names used in abstractions and is intended for visitors displaying lambda terms, and `TermTransformer` allows transforming a lambda term into a new term while automatically removing the name tag from lambda terms that have been changed by the transformation. It is intended for substitution operations and β-reductions.



2.7.1 Interface `Visitor<T>`

Represents a visitor that visits `lambdaTerm`s and returns objects of a given type upon visiting a `lambdaTerm`.

<T>: The type of object that is returned when visiting a term.

Methods:

- `T visitAbstraction(Abstraction abs)`

  Visit an `Abstraction`.

  `abs`: The `Abstraction` to be visited

  Returns: The return value of the `Visitor` when visiting the given `Abstraction`

- `T visitApplication(Application app)`

  Visit an `Application`.

  `app`: The `Application` to be visited

  Returns: The return value of the `Visitor` when visiting the given `Application`

- `T visitBoundVariable(BoundVariable var)`

  Visit a `BoundVariable`.

  `var`: The `BoundVariable` to be visited

  Returns: The return value of the `Visitor` when visiting the given `BoundVariable`

- `T visitFreeVariable(FreeVariable var)`

  Visit a `FreeVariable`.

  `var`: The `FreeVariable` to be visited

  Returns: The return value of the `Visitor` when visiting the given `FreeVariable`

- `T visitNamedTerm(NamedTerm term)`

  Visit a `NamedTerm`.

  `term`: The `NamedTerm` to be visited

  Returns: The return value of the `Visitor` when visiting the given `NamedTerm`

- `T visitPartialApplication(PartialApplication app)`

  Visit a `PartialApplication`.

  `app`: The `PartialApplication` to be visited

  Returns: The return value of the `Visitor` when visiting the given `PartialApplication`

2.7.2 Abstract Class `TermTransformer`

Implements: `Visitor`

A `Visitor` that performs a transformation of some kind of a `LambdaTerm`, automatically removing names if their inner term has been changed by the transformation.

2.7.3 Class `TermNotAcceptableException`

Extends: `RuntimeException`

Thrown when attempting to construct a lambda term that exceeds the maximum allowed size or depth.

Constructors:

- `TermNotAcceptableException(String message)`

2.7.4 Class `SubstitutionVisitor`

Extends: `TermTransformer`

A `Visitor` that substitutes `BoundVariable`s with a given De Bruijn index with a given substituent.

Constructors:

- `SubstitutionVisitor(LambdaTerm substituent)`

  Creates a new substitution visitor.

  `substituent`: The term that should be substituted with

2.7.5 Abstract Class `ResolvedNamesVisitor<T>`

Implements: `Visitor`

A `Visitor` that gives non-colliding names for bound variables.

<T>: The return value of the visitor

Constructors:

- `ResolvedNamesVisitor(List<Library> libraries)`

  Creates a new ResolvedNamesVisitor that checks the given libraries for names that should not be emitted.

  `libraries`: The libraries that are assumed in the ambient context of the terms being visited

Methods:

- `protected abstract T visitBoundVariable(BoundVariable var, String resolvedName)`

  Visit a `BoundVariable` with an additional non-colliding name for said variable.

  This method is provided merely for convenience, the given name will be the same as the one provided for the given abstraction.

  `var`: The `BoundVariable` to be visited

  `resolvedName`: The resolved name for the `BoundVariable`

  Returns: The return value of the `Visitor` upon visiting the given `BoundVariable`

- `protected abstract T visitAbstraction(Abstraction abs, String resolvedName)`

  Visit an `Abstraction` with an additional non-colliding name for its variable.

  `abs`: The `Abstraction` to be visited

  `resolvedName`: The resolved name for the variable of this `Abstraction`

  Returns: The return value of the visitor upon visiting the given `Abstraction`

2.7.6 Abstract Class `PartialApplication`

Implements: `LambdaTerm`

Represents a `LambdaTerm` that consists of a library function that may be accelerated, as well as zero or more applications with arguments for said library function.

Constructors:

- `protected PartialApplication(String name, LambdaTerm inner, int numParameters, List<Visitor<Boolean>> checks)`

  Creates a new partial application that has not yet bound any parameters.

  `name`: The name of the library function.

  `inner`: The `LambdaTerm` for the non-accelerated library function

  `numParameters`: The number of parameters that the library function takes

  `checks`: For each parameter, a `Visitor` that checks whether the given parameter has the correct format for acceleration

Methods:

- `LambdaTerm[] getReceived()`

  Returns the array of received terms.

  Returns: An array of received terms. Only the first getNumReceived elements are valid.

- `int getNumReceived()`

  Returns the number of received terms.

  Returns: The number of received terms

- `int getSize()`

  Returns the size of this partial application.

  Returns: The size of this partial application

- `int getDepth()`

  Returns the depth of this partial application.

  Returns: The depth of this partial application

- `LambdaTerm getRepresented()`

  Returns the `LambdaTerm` that this partial application represents.

  Returns: The `LambdaTerm` that this partial application represents

- `String getName()`

- `LambdaTerm accept(LambdaTerm nextParam)`

  Accepts a new parameter for the partial application.

  If the parameter does not match the format that can be accelerated, returns a new term representing the unaccelerated application.

  If the parameter matches the format that can be accelerated, returns the result of the operation represented by the partial application if all parameters are now present, or a new PartialApplication representing the partial application including the given parameter.

  `nextParam`: The parameter to be accepted

  Returns: A `LambdaTerm` for the partial application with the new parameter as described above

- `protected abstract LambdaTerm accelerate(LambdaTerm[] parameters)`

  Directly determine the result of the computation given all parameters.

  `parameters`: The parameters for the computation

  Returns: The result of the computation

- `protected void absorbClone(PartialApplication other)`

- `protected StringBuilder serializeReceived()`

- `protected void deserializeReceived(String serialized)`

2.7.7 Class `PartialApplication.Addition`

Extends: `PartialApplication`

Represents an acceleratable addition of church numbers.

Constructors:

- `Addition()`

  Creates a new addition.

Static methods:

- `PartialApplication.Addition fromSerialized(String serialized)`

  Restores a serialized addition.

  `serialized`: The serialized addition

  Returns: The restored addition

2.7.8 Class `PartialApplication.Successor`

Extends: `PartialApplication`

Represents the acceleratable successor operation on church numbers.

Constructors:

- `Successor()`

  Creates a new successor operation.

Static methods:

- `PartialApplication.Successor fromSerialized(String serialized)`

  Restores a serialized successor operation.

  `serialized`: The serialized successor

  Returns: The restored succor

2.7.9 Class `PartialApplication.Multiplication`

Extends: `PartialApplication`

An acceleratable multiplication operation.

Constructors:

- `Multiplication()`

  Creates a new instance of the multiplication operation.

Static methods:

- `PartialApplication.Multiplication fromSerialized(String serialized)`

  Restores a serialized multiplication operation.

  `serialized`: A serialized multiplication operation

  Returns: The restored multiplication

2.7.10 Class `PartialApplication.Exponentiation`

Extends: `PartialApplication`

An acceleratable exponentitation operation.

Constructors:

- `Exponentiation()`

  Creates a new instance of the exponentiation operation.

Static methods:

- `PartialApplication.Exponentiation fromSerialized(String serialized)`

  Restores a serialized exponentiation.

  `serialized`: The serialized exponentiation

  Returns: The restored exponentiation

2.7.11 Class `PartialApplication.Predecessor`

Extends: `PartialApplication`

An acceleratable predecessor operation.

Constructors:

- `Predecessor()`

  Creates a new predecessor operation.

Static methods:

- `PartialApplication.Predecessor fromSerialized(String serialized)`

  Restores a serialized predecessor operation.

  `serialized`: The serialized predecessor operation

  Returns: The restored predecessor operation

2.7.12 Class `PartialApplication.Subtraction`

Extends: `PartialApplication`

Represents an acceleratable subtraction operation.

Constructors:

- `Subtraction()`

  Creates a new subtraction operation.

Static methods:

- `PartialApplication.Subtraction fromSerialized(String serialized)`

  Restores a subtraction operation from its serialization.

  `serialized`: The serialized subtraction

  Returns: The restored subtraction

2.7.13 Class `NamedTerm`

Implements: `LambdaTerm`

Represents a `LambdaTerm` that has a name.

Constructors:

- `NamedTerm(String name, LambdaTerm inner)`

  Creates a new named term.

  `name`: The name of the term

  `inner`: The actual term that is being named

Static methods:

- `NamedTerm fromSerialized(String serialized)`

  Restores a serialized named term.

  `serialized`: The serialization of a named term

  Returns: The restored named term

Methods:

- `LambdaTerm getInner()`

  Returns the term that the named term represents.

  Returns: The term that the named term represents

- `String getName()`

  Returns the name of the term.

  Returns: The name of the term

- `int getSize()`

  Returns the size of this named term.

  Returns: The size of this named term

- `int getDepth()`

  Returns the depth of this named term.

  Returns: The depth of this named term

2.7.14 Abstract Class `NameAgnosticVisitor<T>`

Implements: `Visitor`

A `Visitor` that will treat a `NamedTerm` precisely like the term that it represents.

<T>: The return value of the `Visitor`

2.7.15 Interface `LambdaTerm`

Extends: `Serializable`

Represents a term in the untyped lambda calculus.

Static methods:

- `LambdaTerm deserialize(String serialized)`

  Creates a lambda term from its serialization.

  `serialized`: The serialized representation of the lambda term

  Returns: A lambda term that is equal to the term that was serialized

- `LambdaTerm churchNumber(int value)`

  Returns a named lambda term that corresponds to a church number.

  `value`: The value of the church number to construct. Must be nonnegative.

  Returns: The requested church number

Methods:

- `LambdaTerm clone()`

  Creates a deep clone of the lambda term.

  Returns: A deep clone

- `<T> T acceptVisitor(Visitor<T> v)`

  Accept a `Visitor` by invoking the correct visit* method.

  `v`: The `Visitor` whose correct visit* method should be invoked

  Returns: The return value of the invoked visit* method

2.7.16 Class `IsRedexVisitor`

Extends: `NameAgnosticVisitor`

A `Visitor` that returns a boolean that is true if the given `LambdaTerm` represents a redex (possibly bound to one or more nested names).

2.7.17 Class `IsPartialApplicationVisitor`

Extends: `NameAgnosticVisitor`

A visitor that returns true iff it was invoked on a partial application, possibly bound to one or more nested names.

2.7.18 Class `IsChurchNumberVisitor`

Extends: `NameAgnosticVisitor`

Constructors:

- `IsChurchNumberVisitor(int abstractionsSeen)`

2.7.19 Class `IsAbstractionVisitor`

Extends: `NameAgnosticVisitor`

A `Visitor` that returns a boolean that is true if and only if the given `LambdaTerm` represents an `Abstraction` (possibly bound to one or more nested names).

2.7.20 Class `GetChurchNumberVisitor`

Extends: `NameAgnosticVisitor`

2.7.21 Class `FreeVariable`

Implements: `LambdaTerm`

Represents a free variable in the untyped lambda calculus.

A free variable has a name. Unlike the name of the variable bound during an abstraction, this name is not a preferred name, but fixed, since it is free over the entire lambda term that is being represented. Changing this name would therefore change the lambda term.

Constructors:

- `FreeVariable(String name)`

  Creates a new free variable term.

  `name`: The name of the free variable being referenced

Static methods:

- `FreeVariable fromSerialized(String serialized)`

  Restores a free variable from its serialization.

  `serialized`: A serialized free variable

  Returns: The free variable referred to by the serialization

Methods:

- `String getName()`

  Returns the name of the free variable.

  Returns: The name of the free variable

- `int getDepth()`

  Returns the depth of this free variable.

  Returns: The depth of this free variable

- `int getSize()`

  Returns the size of this free variable.

  Returns: The size of this free variable

2.7.22 Class `BoundVariable`

Implements: `LambdaTerm`

Represents a bound variable in the untyped lambda calculus.

It refers to its corresponding `Abstraction` using its De Bruijn index.

Constructors:

- `BoundVariable(int deBruijnIndex)`

  Creates a new bound variable term.

  `deBruijnIndex`: The De Bruijn index of the term

Static methods:

- `BoundVariable fromSerialized(String serialized)`

  Restores a bound variable from its serialization.

  `serialized`: A serialized bound variable

  Returns: The bound variable that the serialization string describes

Methods:

- `int getDeBruijnIndex()`

  Returns the De Bruijn index of the variable.

  Returns: The De Bruijn index

- `int getSize()`

  Returns the size of this bound variable.

  Returns: The size of this bound variable

- `int getDepth()`

  Returns the depth of this bound variable.

  Returns: The depth of this bound variable

2.7.23 Class `BetaReducer`

Extends: `TermTransformer`

A `Visitor` that transforms a `LambdaTerm` by beta reducing a given redex.

Constructors:

- `BetaReducer(Application toReduce)`

  Creates a new beta reducer that reduces the given redex.

  `toReduce`: The application that should be reduced. This must be a redex, otherwise an exception is thrown.

2.7.24 Class `Application`

Implements: `LambdaTerm`

Represents an application in the untyped lambda calculus.

An application has a left hand side and a right hand side, both of which may be arbitrary lambda terms.

Constructors:

- `Application(LambdaTerm leftHandSide, LambdaTerm rightHandSide)`

  Creates a new application.

  `leftHandSide`: The left hand side of the application

  `rightHandSide`: The right hand side of the application

Static methods:

- `Application fromSerialized(String serialized)`

  Restores an application from its serialization.

  `serialized`: A serialized application

  Returns: The restored application

Methods:

- `LambdaTerm getLeftHandSide()`

  Returns the left hand side of the application.

  Returns: The left hand side of the application

- `LambdaTerm getRightHandSide()`

  Returns the right hand side of the application.

  Returns: The right hand side of the application

- `int getSize()`

  Returns the size of this application.

  Returns: The size of this application

- `int getDepth()`

  Returns the depth of this application.

  Returns: The depth of this application

2.7.25 Class `Abstraction`

Implements: `LambdaTerm`

Represents an abstraction in the untyped lambda calculus.

An abstraction has an inner term and a preferred name for the variable it abstracts. When displaying the abstraction, a different name may be used, since terms referring to this abstraction will use De Bruijn indices to do so.

Constructors:

- `Abstraction(String preferredName, LambdaTerm inner)`

  Creates a new abstraction.

  `preferredName`: The preferred name for the variable that is abstracted

  `inner`: The lambda term that the abstraction encloses

Static methods:

- `Abstraction fromSerialized(String serialized)`

  Restores an abstraction from its serialization.

  `serialized`: A serialized abstraction

  Returns: The abstraction described by the serialization

Methods:

- `String getPreferredName()`

  Gets the preferred name for the abstracted variable.

  Returns: The preferred name

- `LambdaTerm getInner()`

  Gets the inner term of the abstraction.

  Returns: The term that this abstraction encloses

- `int getSize()`

  Returns the size of this abstraction.

  Returns: The size of this abstraction

- `int getDepth()`

  Returns the depth of this abstraction.

  Returns: The depth of this abstraction

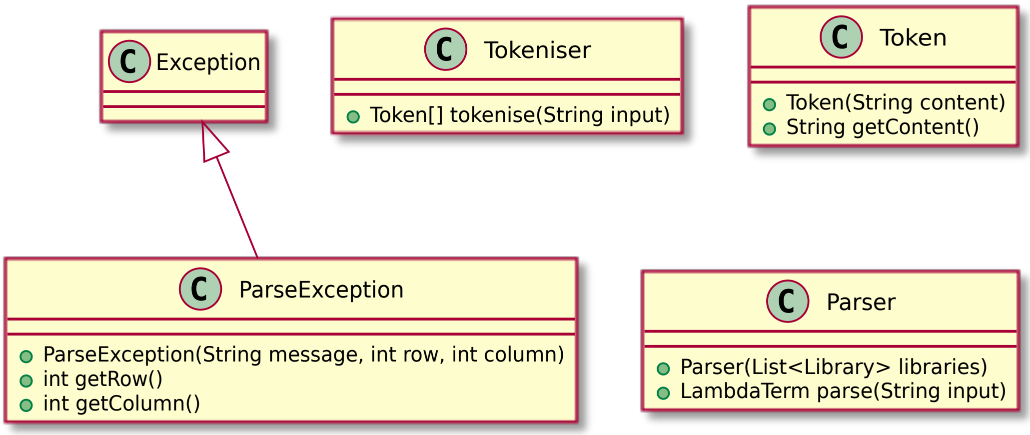
2.8 Package `edu.kit.wavelength.client.model.term.parsing`

The `model.parsing` package contains the classes necessary to turn an input String into a `LambdaTerm` object. This process is initiate by the `ExecutionEngine` instancing a new `Parser` object with the currently activated `Library` and invoking its parse method. The `Parser` in turn uses the `Tokeniser` to turn the input String into a sequence of `Token`s.

Each `Token` contains a section of original input String. This substring is passed to the `Token` on its creation by the `Tokeniser` and can be access trough the `Token`s interface.

From this sequence the `Parser` builds a `LambdaTerm` object structure. If the input contains names referencing library terms and the `Parser` is initialized with the necessary `Library`s, the terms will be incorporated in the `LambdaTerm` object and their names conserved.

If the input String can not be parsed a `ParseEception` is thrown. The `ParseEception` contains an error message describing the error which led to its creation, and where in the input String the error occurred. Both message and location can be accessed and printed to the output window to aid the user in correcting the error.



2.8.1 Class `Tokeniser`

The Tokeniser class is used during the first step of the parsing process to turn the entered input into a sequence of Tokens.

Methods:

- `Token[] tokenise(String input, int offset, int rowPos)`

  Divides a sequence of characters into Tokens.

  `input`: The String to divide into tokens

  Returns: An array containing all tokens

2.8.2 Class `Token`

Token objects are used by both the `Tokeniser` and the `Parser` during the construction of an `LambdaTerm` from an input String. Each token contains a part of the input String which can be accessed through this class' interface.

Constructors:

- `Token(String content, TokenType type, int start, int end)`

  Creates a new Token containing the entered String and type.

  `content`: The String to be stored in the Token

  `type`: The Token's type

`start`: The inclusive start column of the token

`end`: The exclusive end column of the token

Methods:

- `String getContent()`

  Used to access the String that makes up the token.

  Returns: The String making up the token

- `TokenType getType()`

  Returns the type of the token.

  Returns: The type of the token

- `int getStart()`

  Returns the start of the token.

  Returns: The inclusive start of the token

- `int getEnd()`

  Returns the end of the token.

  Returns: The exclusive end of the token

2.8.3 Class `Parser`

This class is used to convert an input String into a `LambdaTerm` object. If any `Library` terms are used in the input, the necessary `Library` have to be passed to the Parser through its constructor.

Constructors:

- `Parser(List<Library> libraries)`

  Initializes a new parser.

  `libraries`: The `Library` to be taken into consideration during parsing

Methods:

- `Library getInputLibary()`

  Gets a library containing the lambda terms and corresponding names defined in the the last invocation of `parse`'s input String.

  Returns: A `Library` containing the terms entered by the user with their assigned names

- `LambdaTerm parse(String input)`

  Parses the input text representation of a lambda term and turns it into a `LambdaTerm` object if successful.

  `input`: The String to parse.

  Returns: The parsed `LambdaTerm` object

2.8.4 Class `ParseException`

Extends: `Exception`

An exception used to indicate an error during the parsing of an entered `LambdaTerm`.

Constructors:

- `ParseException(String message, int row, int columnStart, int columnEnd)`

  Creates a new ParseException with the entered parameters.

  `message`: A message to the user describing the error causing the exception

  `row`: The row containing the source of this exception

  `columnStart`: The inclusive start column of the source of the exception

  `columnEnd`: The exclusive end column of the source of the exception

Methods:

- `int getRow()`

  Gets the row in which the error causing this exception occurred.

  Returns: The row in which the error occurred

- `int getColumnStart()`

  Gets the start column of the token in which the error causing this exception occurred.

  Returns: The inclusive start column of the token in which the error occurred

- `int getColumnEnd()`

  Gets the end column of the token in which the error causing this exception occurred.

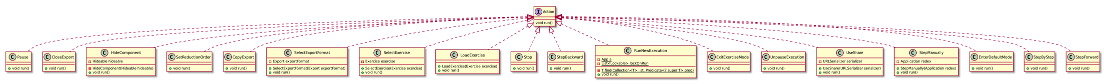
  Returns: The exclusive end column of the token in which the error occurred

2.9 Package edu.kit.wavelength.client.view.action

The action package implements the command pattern to provide various classes that act as event-handlers.

All classes in this package implement the Action Interface. By doing so all the information that is needed to handle an event is encapsulated by the action classes.

Most of the actions are called by components form the view.webui.component package and will handle requests from the user and update the user interface appropriately. However some actions can be invoked on other events (such as the EnterDefaultMode or the LoadExercise class).



2.9.1 Class UseShare

Implements: Action

This action generates the permalink and toggles the dedicated panel. The permalink encodes the current input, output and settings.

Constructors:

- UseShare(List<SerializationObserver> serializationOutputs)

  Creates a new UseShare action with a given list of SerializationObserver that are updated each time the action runs.

  serializationOutputs: observers that are updated with the id of the database entry for the serialization String

2.9.2 Class Unpause

Implements: Action

Continues a paused execution, starting at the current point of execution.

2.9.3 Class `ToggleTermInfo`

Implements: `Action`

Toggles the div that contains the term info for a library.

Constructors:

- `ToggleTermInfo(FlowPanel infoDiv)`

  Constructor

  `infoDiv`: term info div to toggle when button is clicked

2.9.4 Class `StepManually`

Implements: `Action`

Action that initiates a manual step on a particular redex in an output view.

Constructors:

- `StepManually(Application redex)`

  Creates the action with the redex to apply when the user clicks on a redex.

  `redex`: The redex to apply

2.9.5 Class `StepForward`

Implements: `Action`

This action requests and displays the next reduction step of the current execution. It also pauses the ongoing execution.

2.9.6 Class `StepBackward`

Implements: `Action`

This class removes the last shown reduction step from the output and pauses the ongoing execution.

2.9.7 Class `SetReductionOrder`

Implements: `Action`

This action changes the reduction order for the further execution.

2.9.8 Class `SetOutputSize`

Implements: `Action`

Changes the output size to the selected one. This will only affect upcoming display of lambda terms, it has no effect on the already displayed terms.

2.9.9 Class `SetOutputFormat`

Implements: `Action`

Changes the output format to the selected one. This options only affects the last displayed and all further terms. It has no effect on all other displayed terms.

2.9.10 Class `SelectLibrary`

Implements: `Action`

Includes the selected library. This only affects newly started executions.

2.9.11 Class `SelectExportFormat`

Implements: `Action`

This action displays the currently displayed output in the selected export format in a pop up export window.

Constructors:

- `SelectExportFormat(Export exportFormat)`

  Constructs a new action handler for the selection of an export format.

  `exportFormat`: The export format the user chose

2.9.12 Class `SelectExercise`

Implements: `Action`

This action will try to load a new exercise and alerts the user that the content of the Editor would be overwritten.

Constructors:

- `SelectExercise(LoadExercise loadExerciseAction, Exercise selected)`

  Constructor.

  `loadExerciseAction`: action to run with selected exercise

  `selected`: exercise that is selected when this action fires

Methods:

- `Exercise getExercise()`

  Gets the exercise that is selected when this action fires

  Returns: exercise that is selected when this action fires

2.9.13 Class `RunExecution`

Implements: `Action`

This class starts a new reduction process and sets the view accordingly. It reads the users input and all required options needed to start the execution.

2.9.14 Class `Pause`

Implements: `Action`

This class pauses the currently running execution process and allows the user to now navigate through the reduction process himself.

2.9.15 Class `LoadExercise`

Implements: `Action`

This class changes the view from standard input to exercise view to display the selected exercise.

Methods:

- `Exercise getExercise()`

  Gets the selected exercise to load.

  Returns: Exercise

- `void setExercise(Exercise exercise)`

  Sets the selected exercise to load.

  `exercise`:

2.9.16 Class `EnterDefaultMode`

Implements: `Action`

This class changes the view from exercise mode view to standard input view.

2.9.17 Class `Control`

Utility class for updating the UI elements according to the current UI state and resetting the UI to default state.

Static methods:

- `void updateControls()`

  Updates the UI elements according to the current UI state.

- `void wipe()`

  Clears the editor and output area. Terminates the running execution and updates the UI to default state.

2.9.18 Class `Clear`

Implements: `Action`

Terminates the running execution, wipes the output area and updates the UI elements according to the new state.

2.9.19 Interface `Action`

The Action interface encapsulates all events that can occur from the user interacting with the UI.

An Action class is the event handler for one UI event. It is triggered by interacting with the dedicated UI element.

Methods:

- `void run()`

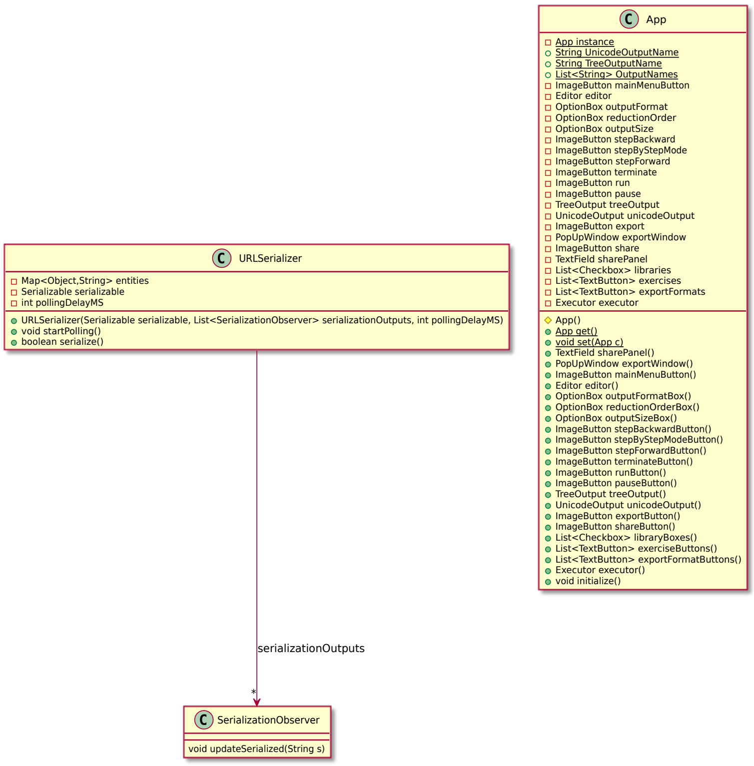
  Called when the Action is triggered.

2.10 Package `edu.kit.wavelength.client.view`

The `view` package manages general aspects of the application. Among these are building the UI, managing the interaction between the `model` and the `view` and serialization.

The `URLSerializer` class manages the serialization of the applications state. Hence it needs the objects that are serialized and the recipients of the serialization. The recipients must implement the `SerializationObserver` interface in order to update the serialization. Note that the `URLSerializer` needs a polling delay time in milliseconds for concurrency.

The `App` class builds the UI components and the associated actions. It also stores an instance of the class. Note that the `App` class works as a singleton, guaranteeing that only one instance exists and is accessed using a static method. Hence each `Action` can access the `App` in order to use the `model` via the `Executor` class or get UI components by calling the respective methods.



2.10.1 Interface `SerializationObserver`

Observer that receives updates containing the most recent id of a serialization.

Methods:

- `void updateSerialized(String id)`

  Updates the observer.

  `id`: identifier belonging to the most recent serialization

2.10.2 Class `App`

Implements: `Serializable`

App is a singleton that initializes and holds the view.

Static methods:

- `App get()`

  Creates a new instance of App if there is none. Returns a singleton instance of App.

  Returns: singleton instance of App

- `void autoScrollOutput()`

  Automatically scrolls to the bottom of the output window.

- `void copyToClipboard(String id)`

Methods:

- `StringBuilder serialize()`

  Serializes the Application by returning a String from which the state of the application can be recreated.

  The String holds information about the `Executor`, the `Editor`, the `OptionBox`es, the selected `Library`s and the selected `Exercise`s in this order.

  Returns: the string representation of the application

- `void deserialize(String content)`

  Deserializes the Application with the given String.

  This includes `Executor`, the `Editor`, the `OptionBox`es the selected `Library`s and the selected `Exercises`. It sets the application into step by step mode if the Executor holds terms and leaves the application in its initial state else.

  `content`: the string representing the state of the application

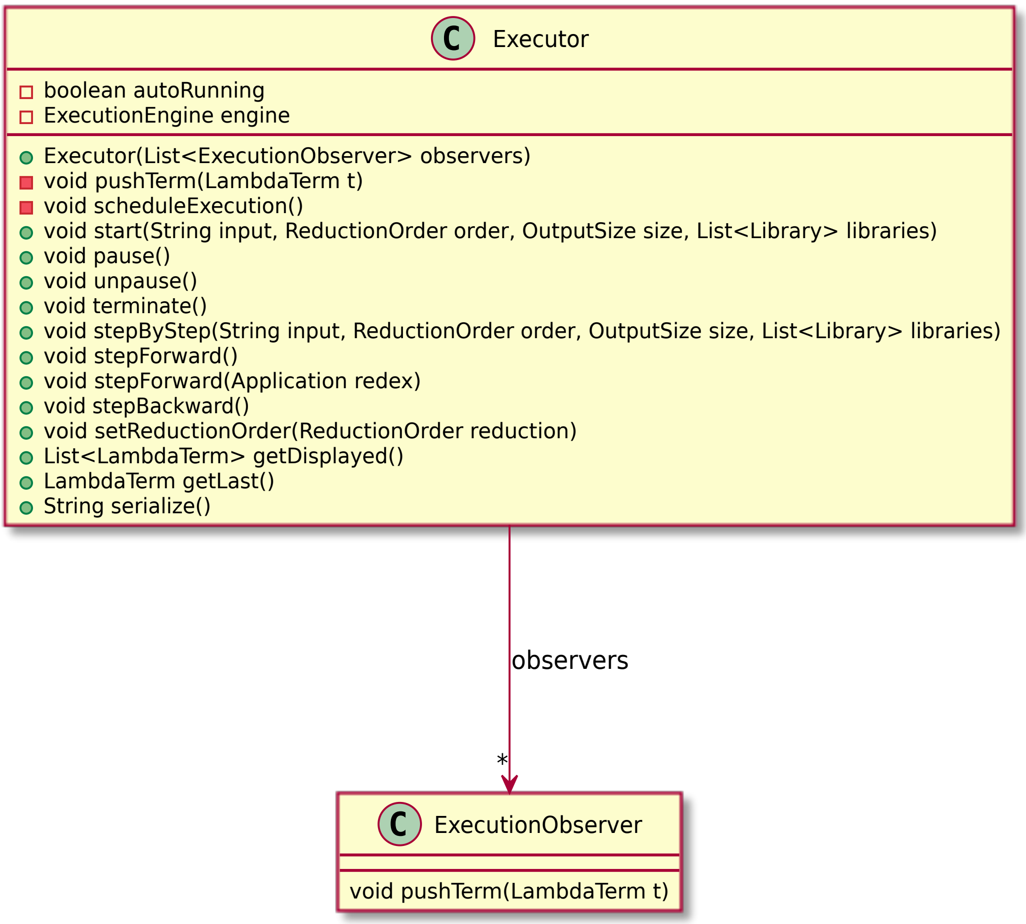
- `DockLayoutPanel mainPanel()`
- `DropDown mainMenu()`
- `Button openMainMenuButton()`
- `DropDownMenu mainMenuPanel()`
- `DropDownHeader mainMenuLibraryTitle()`
- `List<CheckBox> libraryCheckBoxes()`
- `List<Button> libraryTermInfoToggleButtons()`
- `List<FlowPanel> libraryTermInfos()`
- `Divider mainMenuDivider()`
- `DropDownHeader mainMenuExerciseTitle()`
- `List<AnchorListItem> exerciseButtons()`
- `FlowPanel footerPanel()`

- `ButtonGroup exportDropupGroup()`
- `Button openExportMenuButton()`
- `DropDownMenu exportMenu()`
- `List<AnchorListItem> exportButtons()`
- `ButtonGroup shareGroup()`
- `Button shareButton()`
- `TextBox sharePanel()`
- `Button copyShareURLButton()`
- `Label reductionStepCounterLabel()`
- `SplitLayoutPanel ioPanel()`
- `DockLayoutPanel inputPanel()`
- `FlowPanel inputControlPanel()`
- `FlowPanel optionBarPanel()`
- `ListBox outputFormatBox()`
- `ListBox reductionOrderBox()`
- `ListBox outputSizeBox()`
- `FlowPanel controlPanel()`
- `FlowPanel stepByStepControlPanel()`
- `Button backwardButton()`
- `Button pauseButton()`
- `Button unpauseButton()`
- `Button forwardButton()`
- `Label spinner()`
- `FlowPanel runControlPanel()`
- `Button clearButton()`
- `Button runButton()`
- `SplitLayoutPanel editorExercisePanel()`
- `FlowPanel exercisePanel()`
- `FlowPanel exerciseHeaderPanel()`

- `FlowPanel exerciseControlPanel()`
- `Button toggleSolutionButton()`
- `Button closeExerciseButton()`
- `Label exerciseDescriptionLabel()`
- `TextArea solutionArea()`
- `SimplePanel editorPanel()`
- `FlowPanel outputBlocker()`
- `FlowPanel outputArea()`
- `Modal loadExercisePopup()`
- `ModalBody loadExercisePopupBody()`
- `Label loadExercisePopupText()`
- `ModalFooter loadExercisePopupFooter()`
- `Button loadExercisePopupOkButton()`
- `Button loadExercisePopupCancelButton()`
- `Modal closeExercisePopup()`
- `ModalBody closeExercisePopupBody()`
- `Label closeExercisePopupText()`
- `ModalFooter closeExercisePopupFooter()`
- `Button closeExercisePopupOkButton()`
- `Button closeExercisePopupCancelButton()`
- `Modal exportPopup()`
- `ModalBody exportPopupBody()`
- `TextArea exportArea()`
- `ModalFooter exportPopupFooter()`
- `Button exportPopupOkButton()`
- `MonacoEditor editor()`
- `Executor executor()`
- `Exercise currentExercise()`
- `void setCurrentExercise(Exercise e)`

2.11 Package `edu.kit.wavelength.client.view.execution`

The package `view.execution` holds the `Executor`, which adapts a `ExecutionEngine`. This adaption is necessary because the reduction of a lambda term is concurrent with the UI interaction itself. As GWT runs in a single threaded browser environment, it has its own concurrency mechanism: A `Scheduler` that allows one to schedule a specific action at the end of the event loop of the browser. As a result of this the Executor schedules one reduction step at the end of every event loop iteration and provides methods to control this concurrent execution. The package also contains an `ExecutionObserver`, which is used by the `Executor` to push reduced terms that are intended to be displayed in the `view` to observers observing it.



2.11.1 Interface `ReductionStepCountObserver`

Receives the current amount of reduction steps.

Methods:

- `void update(int count)`

  Receives the current amount of reduction steps.

  `count`: - new count

2.11.2 Class `Executor`

Implements: `Serializable`

Concurrently reduces lambda terms.

Constructors:

- `Executor(List<ExecutionObserver> executionObservers, List<ControlObserver> controlObservers, List<ReductionStepCountObserver> reductionStepCountObserver)`

  Creates a new Executor.

  `executionObservers`: Observers to update with reduced lambda terms

  `controlObservers`: Observers to notify when executor reaches certain states

Methods:

- `void start(String input, ReductionOrder order, OutputSize size, List<Library> libraries)`

  Starts the automatic execution of the input, parsing the term and then reducing it.

  `input`: code to parse and reduce

  `order`: order with which to reduce

  `size`: which terms to push to observers

  `libraries`: libraries to consider when parsing

- `void stepByStep(String input, ReductionOrder order, OutputSize size, List<Library> libraries)`

  Initiates the step by step execution, allowing the caller to choose the next step.

  `input`: code to parse and execute

  `order`: order with which to reduce

  `size`: which terms to push to observers

  `libraries`: libraries to consider when parsing

- `void pause()`

  Pauses the automatic execution, transitioning into the step by step mode.

- `void unpause()`

  Unpauses the automatic execution, transitioning from step by step mode into automatic execution.

- `void terminate()`

  Terminates the step by step- and automatic execution.

- `void stepForward()`

  Executes a single reduction of the current lambda term.

- `void stepForward(Application redex)`

  Executes a single reduction of the supplied redex.

  `redex`: The redex to be evaluated. Must be a redex, otherwise an exception is thrown

- `void stepBackward()`

  Reverts to the previously output lambda term.

- `boolean canStepBackward()`

  Checks whether stepBackward is possible.

  Returns: whether stepBackward is possible

- `boolean canStepForward()`

  Checks whether stepForward is possible.

  Returns: whether stepForward is possible

- `boolean isPaused()`

  Checks whether the engine is paused.

  Returns: whether the engine is paused

- boolean isTerminated()

  Checks whether the engine is terminated.

  Returns: whether the engine is terminated

- boolean isRunning()

  Checks whether the engine is running.

  Returns: whether the engine is running

- List<LambdaTerm> getDisplayed()

  Returns the currently displayed lambda terms.

  Returns: lt

- List<Library> getLibraries()

  Returns the libraries in use by the engine.

  Returns: libraries

- StringBuilder serialize()

  Serializes the Executor by serializing its ExecutionEngine.

  Returns: The Executor serialized String representation

- void deserialize(String serialization)

  Deserializes the Executor by deserializing its ExecutionEngine. Also loads the correct content into OutputArea.

  serialization: serialized Executor

- void setReductionOrder(ReductionOrder reduction)

  Changes the active reduction order to the entered one.

  reduction: The new reduction order

- void updatedOutputFormat()

  Causes the last displayed term to be reloaded if a new output format was selected.

- void setOutputSize(OutputSize s)

  Changes the active output size to the entered one.

  s: The new output size

2.11.3 Interface `ExecutionObserver`

Observer that receives reduced terms. Necessary because Executor is concurrent with UI.

Methods:

- `void pushTerm(LambdaTerm t)`

  Pushes the most recent displayed term.

  `t`: the most recent term

- `void removeLastTerm()`

  Removes the last displayed term.

- `void clear()`

  Resets the observer.

- `void reloadTerm()`

  Reloads the last displayed term.

- `void pushError(String error)`

  Pushes an error message.

  `error`: - message

2.11.4 Interface `ControlObserver`

Observer that is notified when the Executor executes certain state transitions.

Methods:

- `void finish()`

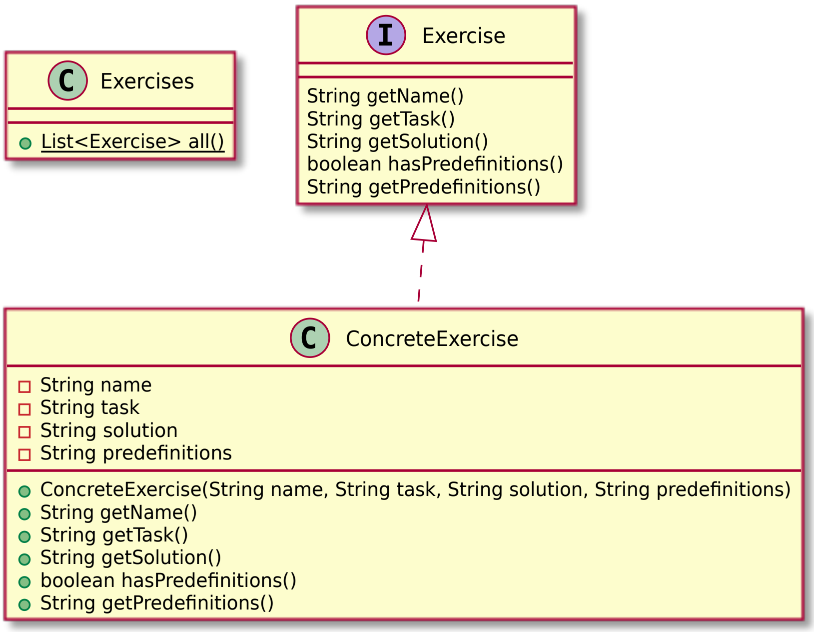
  Called when the Executor finishes its execution, i.e. the last term is reduced.

2.12 Package `edu.kit.wavelength.client.view.exercise`

The `view.exercise` package contains the applications exercise system:

The `Exercise` interface specifies that exercises consist of a name, a task, a solution and (optional) predefined variables. The `ConcreteExercise` class gives an implementation allowing for a simple generation of exercises. Note that exercises use String representations of λ-Terms which must be transformed to `LambdaTerm` objects when used.

The `Exercises` class static has only one method which statically returns a list of all available exercises.



2.12.1 Class `TermGenerator`

This class is used to create random lambda terms for use in exercises.

Methods:

- `LambdaTerm getNewTerm(int minDepth, int maxDepth)`

  Creates a new random LambdaTerm. The input minimal and maximal depth are used to limit the terms size.

  `minDepth`: The minimal term depth

`maxDepth`: The maximal term depth

Returns: The newly created term

- `LambdaTerm getNewTerm(int minDepth, int maxDepth, int termDepth, int abstractionDepth)`

  Create a new random sub term.

  `minDepth`: The minimal term depth

  `maxDepth`: The maximal term depth

  `termDepth`: The current depth

  `abstractionDepth`: The number of abstractions enclosing this term.

  Returns: The newly created sub term.

- `protected int nextInt(int bound)`

  Return a random integer in [0,bound).

  `bound`: The upper bound to use

  Returns: a random integer

- `String getRandomVarName()`

  Return a random lower case character.

  Returns: a random lower case character

2.12.2 Class `RedexExercise`

Implements: `Exercise`

Exercise which generates Lambda Terms randomly and asks the user to search for reducible expressions.

Constructors:

- `RedexExercise(ReductionOrder reduction)`

  Creates a new random redex exercise that uses the given `ReductionOrder`.

  `reduction`: The `ReductionOrder` the user should use

- `RedexExercise(ReductionOrder reduction, TermGenerator generator)`

  Creates a new random redex exercise that uses the given `ReductionOrder` and the given `TermGenerator`.

  `reduction`: The `ReductionOrder` the user should use

  `generator`: The `TermGenerator` used for term generation.

Methods:

- `void reset()`

  Resets the exercise by randomly creating a new term and updating predefinition and solution.

2.12.3 Class `Exercises`

Static class giving access to all available exercises.

Static methods:

- `List<Exercise> all()`

  Returns an unmodifiable list of all available exercises.

  Returns: An unmodifiable list that contains exactly one instance of every available exercise

- `Exercise deserialize(String serialized)`

  Deserializes an `Exercise`.

  `serialized`: supposedly a serialization of an Exercise

  Returns: Exercise belonging to serialization

2.12.4 Interface `Exercise`

Extends: `Serializable`

An exercise consists of a task specifying what the User is supposed to do and a solution specifying what the result should look like. Additionally exercises may provide a basis for a given task.

Methods:

- `String getName()`

  Gets the name of the exercise.

  Returns: The name of the exercise

- `String getTask()`

  Returns the explanation of the exercise.

  Returns: The description of the task

- `String getSolution()`

  Returns the sample solution. Note that this may not be the only possible solution.

  Returns: The solution of the exercise

- `boolean hasPredefinitions()`

  Returns whether this has predefined code or not.

  Returns: `true` if this Exercise has predefined code

- `String getPredefinitions()`

  Returns initial definitions that are supposed to be of help for the User. Note that this may be empty.

  Returns: The predefined code

2.12.5 Class `ConcreteExercise`

Implements: `Exercise`

This class is a concrete implementation of the `Exercise` interface. The needed method's return values are set in the constructor.

Constructors:

- `ConcreteExercise(String name, String task, String solution, String predefinitions)`

  Creates a new Exercise.

  `name`: - name of the exercise

  `task`: - problem task to display

  `solution`: - intended solution for the problem

  `predefinitions`: - initial code to load into the editor

2.12.6 Class `BoundVariableResolver`

Implements: `Visitor`

This visitor is used by the `RegexExercise` class to ensure the sub term making up the predefined term's redex can be exported to a String.

Methods:

- `LambdaTerm resolveVariables(LambdaTerm term, LambdaTerm subTerm)`

  Turn every bound variable in the entered sub term whose corresponding abstraction is outside the sub term into a FreeVariable with the the bound variables preferred name. The resulting sub term is a correct lambda term, containing no faulty indices.

  `term`: The term containing the subterm

  `subTerm`: The sub term whose bound variables to replace

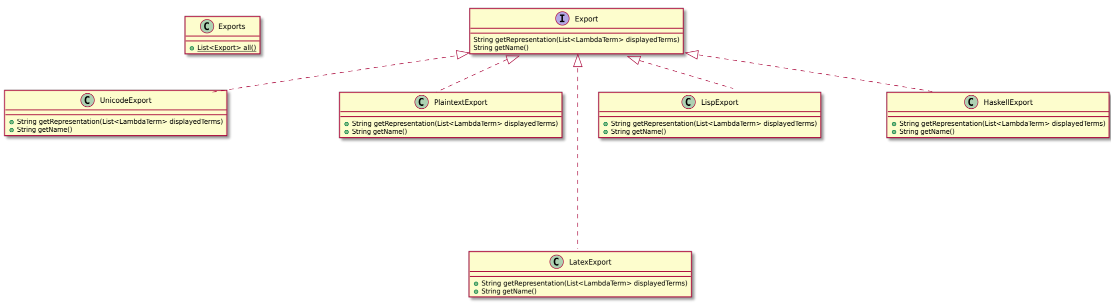
  Returns: The sub term with every bound variable adjusted.

2.13 Package `edu.kit.wavelength.client.view.export`

The `export` package provides the classes used by the application to transform given `LambdaTerms` into a String representation depending on the selected format. Also the `export` package contains a `Exports` class that provides information about the available export formats.

All format classes implement the interface `Export`. The representation of a given `LambdaTerm` can thus be requested via a unique method. The second method of the interface returns the name of the export format to the caller. Since these classes just transform the given terms, it is not guaranteed that the generated representation is executable. This concerns mainly the classes `HaskellExport` and `LispExport`.

All available export formats can be requested by calling the `Exports` class. Its single method returns a collection of the export format classes.



2.13.1 Class `UnicodeExport`

Implements: `Export`

This class translates the given lambda terms into text using a unicode lambda letter and a unicode arrow.

2.13.2 Class `PlaintextExport`

Implements: `Export`

This class translates the given lambda terms into plain text. The lambda symbol is represented by a backslash.

2.13.3 Class `LispExportVisitor`

Extends: `BasicExportVisitor`

This class is a visitor to translate a lambda term into a string using Lisp syntax. However it is not guaranteed that the generated representation is executable Lisp code.

Constructors:

- `LispExportVisitor(List<Library> libraries)`

2.13.4 Class `LispExport`

Implements: `Export`

This class translates the given lambda terms into Lisp code. Since it is only a syntactic translation, it is not guaranteed that the generated output is executable Lisp.

2.13.5 Class `LaTeXExportVisitor`

Extends: `BasicExportVisitor`

This class is a visitor to translate a lambda term into a string using LaTeX syntax.

Constructors:

- `LaTeXExportVisitor(List<Library> libraries)`

2.13.6 Class `LatexExport`

Implements: `Export`

This class translates the given lambda terms into LaTeX code. The generated representation assumes math mode when being pasted into an existing LaTeX document.

2.13.7 Class `HaskellExportVisitor`

Extends: `BasicExportVisitor`

This class is a visitor to translate a lambda term into a string using Haskell syntax. However it is not guaranteed that the generated representation is executable Haskell code.

Constructors:

- `HaskellExportVisitor(List<Library> libraries)`

2.13.8 Class `HaskellExport`

Implements: `Export`

This class translates the given lambda terms into Haskell code.

Since it is only a syntactic translation, it is not guaranteed that the generated representation is executable Haskell code.

2.13.9 Class `Exports`

Static class giving access to all `Export`s known to the model.

Static methods:

- `List<Export> all()`

  Returns an unmodifiable list of all available export formats.

  Returns: An unmodifiable list that contains exactly one instance of every export format

2.13.10 Interface `Export`

This interface encapsulates the available export formats. It translates the current output into the corresponding format.

Methods:

- `String getRepresentation(List<LambdaTerm> displayedTerms, List<Library> libraries)`

  This method transforms the given lambda terms into the dedicated format.

  `displayedTerms`: the terms that should be translated

  `libraries`: the libraries of the application that are used in the terms

  Returns: the String representation of the given terms

- `String getName()`

  This method returns the name of the export format.

  Returns: the name of the export format

2.13.11 Class `BasicExportVisitor`

Extends: `ResolvedNamesVisitor`

A basic Visitor to create a string representing a given lambda term. The Visitor sets a minimal number of brackets to correctly describe the lambda term.

It also provides a means of varying the representation of a lambda term by overwriting its strategy methods.

Constructors:

- `BasicExportVisitor(List<Library> libraries, String lambdaRepresentation)`

  Creates a new Visitor for `LambdaTerm`s.

  It will build a String (represented by a `StringBuilder`) from the lambda term with a custom representation for the lambda-letter.

  `libraries`: the libraries of the application that are used in this term

  `lambdaRepresentation`: the string representation of the lambda letter

Methods:

- `protected void setFlags(Boolean set)`

  Sets all Flags for brackets. Flags should be false at the end of each visit method.

  `set`: `true` if the Flags should be set and `false` otherwise

- `protected StringBuilder formatText(StringBuilder text)`

  A strategy method to allow inheriting classes to define the representation of text in the constructed String.

  `text`: the text of the lambda term

  Returns: the text as it should be represented in the final string

- `protected StringBuilder formatLambda(StringBuilder absVariable)`

  A strategy method to allow inheriting classes to define the representation of an abstraction in the constructed String.

  The default setting produces a string containing the lambda letter, followed by the abstraction variable and a dot (e.g. '.').

  `absVariable`: the variable of the abstraction

  Returns: the left part of an abstraction as it should be represented in the final string

- `protected StringBuilder formatApplication(StringBuilder leftSide, StringBuilder rightSide)`

  A strategy method to allow inheriting classed to define the representation of an application in the constructed String.

  The default setting seperates the left and right side of the application by a space.

  `leftSide`: the left Side of the application

  `rightSide`: the right Side of the application

  Returns: the application as it should be represented in the final string

2.14 Package `edu.kit.wavelength.client.view.gwt`

2.14.1 Class `VisJs`

Wrapper class for the java script library vis.js. It is used for pretty printing lambda terms as syntax trees.

Static methods:

- `void loadNetwork(String nodes, String edges, Panel parent)`

  Renders a new network graph in the given panel element.

  `nodes`: String representation of the nodes

  `edges`: String representation of the edges

  `parent`: The panel to wrap the network in

2.14.2 Class `Notify`

Static methods:

- `void error(String msg)`

2.14.3 Class `MonacoEditor`

Wrapper for the monaco-js library. Provides a subset of functions of the library that is useful to the application.

Static methods:

- `MonacoEditor load(Panel parent)`

  Loads the editor into the specified parent and creates a wrapper to control the editor through GWT.

  `parent`: - parent to load into

  Returns: wrapper

Methods:

- `void setLibraries(List<Library> libraries)`
- `String read()`

  Reads the contents of the editor.

  Returns: contents

- `void write(String s)`

  Writes the specified contents to the editor.

  `s`: - string to replace editor content with

- `void lock()`

  Disables editor input.

- `void unlock()`

  Enables editor input.

- `boolean isLocked()`

  Checks whether the editor is editable.

  Returns: whether the editor is editable

- `void error(String message, int startLineNumber, int endLineNumber, int startColumn, int endColumn)`

  Displays an error in the editor.

  `message`: - message of the error

  `startLineNumber`: - start line number position of the error

  `endLineNumber`: - end line number position of the error

  `startColumn`: - start column number position of the error

  `endColumn`: - end column number position of the error

- `void unerror()`

  Removes all error indicators from the editor.

2.15 Package `edu.kit.wavelength.client.view.update`

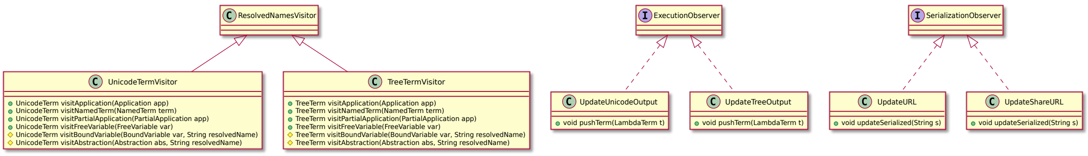
The `view.update` package contains observer implementations that update the `view`. In the view, observers are primarily used for pushing concurrent updates to the view. The package contains two kinds of observer implementations: `SerializationObserver` and `ExecutionObserver`. Implementations for the former update the view with a new serialized URL every time one is produced. Implementations for the latter update the view with a new `LambdaTerm` every time one is produced by the `Executor` and chosen to be displayed by the given `OutputSize`. There are two `SerializationObserver` implementations:

- `UpdateShareURL`: Updates the text panel that is opened when pressing the share button with the new URL.
- `UpdateURL`: Updates the browser URL with the new URL.

There are also two `ExecutionObserver` implementations:

- `UpdateUnicodeOutput`: Updates the unicode text output panel with a new term if the panel is visible.
- `UpdateTreeOutput`: Updates the tree output panel with a new term if the panel is visible.

Both `UpdateUnicodeOutput` and `UpdateTreeOutput` use a `Visitor` to generate the respective `UnicodeTerm` and `TreeTerm`. `UnicodeTermVisitor` is the `Visitor` that traverses the `LambdaTerm` and generates a `UnicodeTerm` while `UpdateTreeOutput` generates a `TreeTerm`.



2.15.1 Class `UpdateShareURL`

Implements: `SerializationObserver`

Observer that updates the URL in the share panel.

2.15.2 Class `UpdateReductionStepCounter`

Implements: `ReductionStepCountObserver`

Updates the reduction step counter.

2.15.3 Class `UpdateOutput`

Implements: `ExecutionObserver`

Updates the output.

Constructors:

- `UpdateOutput()`

  Creates a new Output class.

Methods:

- `void reloadTerm()`

  This method is triggered if the user wants to change the output by selecting a different format or a different reduction order.

2.15.4 Class `UnicodeTuple`

This class represents a tuple of a gwt FlowPanel widget and a gwt anchor widget.

Constructors:

- `UnicodeTuple(FlowPanel panel, Anchor a)`

  Creates a new tuple with the given parameters.

  `panel`: The panel of this tuple

  `a`: The anchor of this tuple

2.15.5 Class `UnicodeTermVisitor`

Extends: `ResolvedNamesVisitor`

Visitor for generating the output of a `LambdaTerm` for the `UnicodeOutput` view.

Constructors:

- `UnicodeTermVisitor(List<Library> libraries, Application nextRedex, FlowPanel parent)`

  Creates a new ResolvedNamesVisitor for unicode pretty printing.

  `libraries`: The libraries to take into account.

  `nextRedex`: The redex that is reduced next with the current `ReductionOrder`

  `parent`: The panel this term will be wrapped in.

2.15.6 Class `TreeTriple`

This class represents a triple of a string for the trees nodes, a string for the trees edges and an integer for the node id.

Constructors:

- `TreeTriple(String nodes, String edges, int idFirst)`

  Creates a new triple with the given parameters.

  `nodes`: The nodes represented as string

  `edges`: The edges represented as string

  `idFirst`: The id of the first node

2.15.7 Class `TreeTermVisitor`

Extends: `ResolvedNamesVisitor`

Visitor for generating the output of a `LambdaTerm` for the `TreeOutput` view.

Constructors:

- `TreeTermVisitor(List<Library> libraries, Application nextRedex)`

  Creates a new ResolvedNamesVisitor for tree pretty printing.

  `libraries`: The libraries to take into account.

  `nextRedex`: The redex that is reduced next with the current `ReductionOrder`

2.15.8 Class `TermFormatTuple`

This class represents a tuple of a `LambdaTerm` and the `OutputFormat` that was selected at the time of printing the term.

Constructors:

- `TermFormatTuple(LambdaTerm term, OutputFormat format)`

  Creates a new tuple by taking a LambdaTerm and a format.

2.15.9 Class `OutputFormat`

Provides all available output formats.

Static methods:

- `OutputFormat[] values()`
- `OutputFormat valueOf(String name)`

2.15.10 Class `FinishExecution`

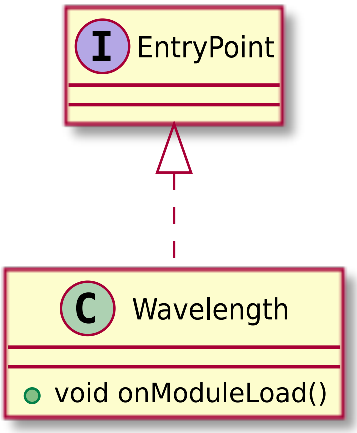
Implements: `ControlObserver`

This class is called when the execution has finished. It adjusts the UI elements according to the new state.

2.16 Package `edu.kit.wavelength.client`

The `client` package holds all other packages and classes that are used in client-side code. It also contains the `Wavelength` class which marks the entry point of the application.

Because the whole web application is run on the client-side and there are no server-sided calculations this package contains the complete code of the application.



2.16.1 Class `Wavelength`

This class marks the entry point of the application.

Methods:

- `void onModuleLoad()`

  This method is called when the application is first started. It initializes the application's `App` class.

2.17 Package `edu.kit.wavelength.server.database`

2.17.1 Class `DatabaseServiceImpl`

Extends: `RemoteServiceServlet`

Implements: `DatabaseService`

Implementation of `DatabaseService` running on server.

This implementation uses `UUID` objects as identifiers for serializations. Note that this class uses the try-with-resources statement to close resources upon finishing.

Constructors:

- `DatabaseServiceImpl()`

  Initialize connection to database located at url given by @value databasePath.

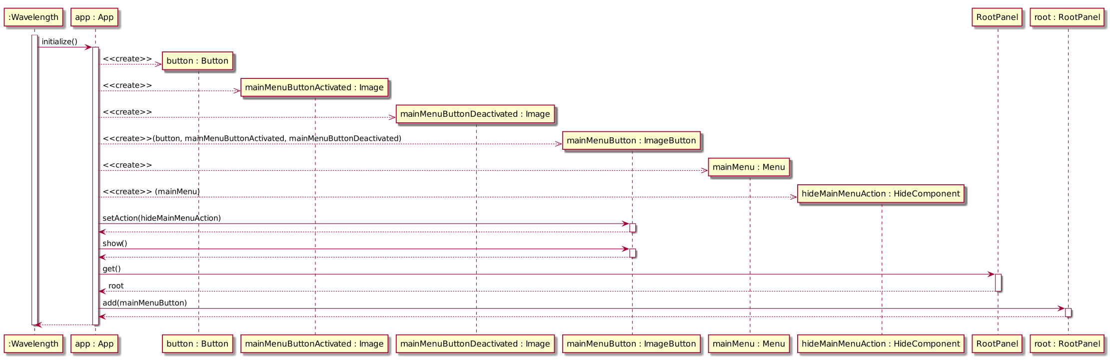
3 Amendments to the requirements laid out in the *Pflichtenheft*

Regarding entry F17 in the “Pflichtenheft”: Enabling test cases raises complexity too much to be considered at this point of time. According to F17 it is possible that an exercises gives predefined variables which are set in the test cases. This leads to the problem that in order for a substitution of those variables some kind of pre-parser needs to be utilized. Hence the decision to not implement test cases for Exercises.

4 Typical processes

4.1 Initialization of application

This sequence diagram shows how the application is initialized by way of example. The `Wavelength` class is called by GWT and initializes `App`. `App` then initializes the view by creating all necessary components, configuring and composing them as necessary and wrapping them in adapter classes that streamline and abstract access to these UI components. Finally, `App` initializes the components with their respective actions, initializes the `Executor` and adds the top level component to the GWT root panel to start rendering the UI.



4.2 Example action: Select export format

This sequence diagram states as example for a simple action handler. In this action the user wants the currently displayed output translated into a specified format. The sequence diagram starts when the user selects the `Export` format in the UI. The `SelectExportFormat` class poses as an action handler for the selected UI element. Thus when this element is clicked, the actions `run()` method is called. All invoked instances are already existing, none is being created in this scenario. The actions `run()` method generates the dedicated representation by requesting the currently displayed `LambdaTerms` from the `Executor` instance and generating their representation. The representation is then written into the export window that will be displayed to the user. Writing before showing this export window ensures, that at no point in time an empty window is displayed.

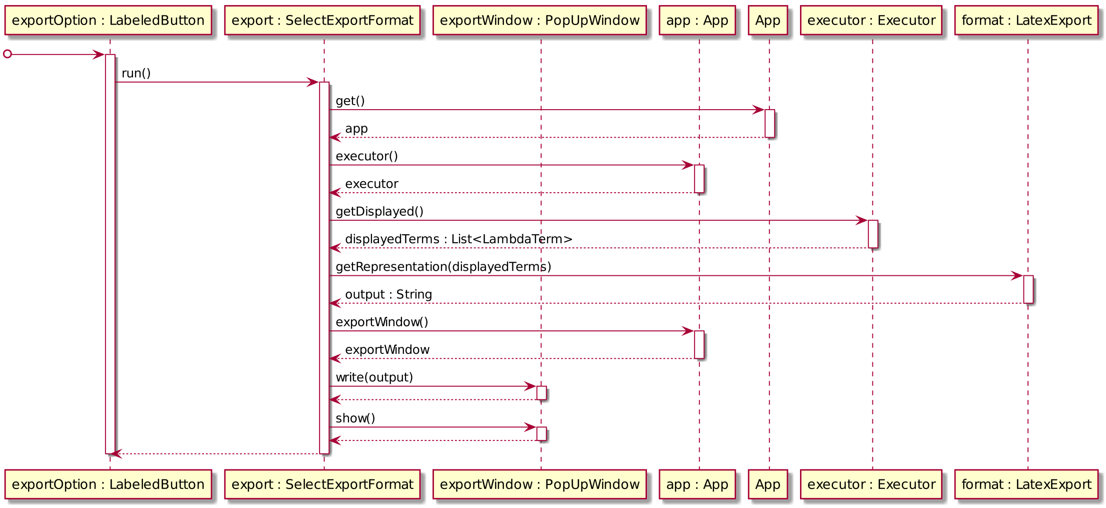


Figure 2: Sequence diagram for the select export format action.

4.3 Example action: Select exercise from main menu

The `SelectExercise` action handles the UI elements, when the user selects an exercise from the main menu.

This action only prepares the UI to display the content of the selected `ConcreteExercise`, it does not load the actual content into the UI. If the content of the `Editor` would be overridden when the selected exerciser is loaded, the action displays a warning message. Otherwise it just calls the `LoadExercise` action which will in return load the selected exercise.

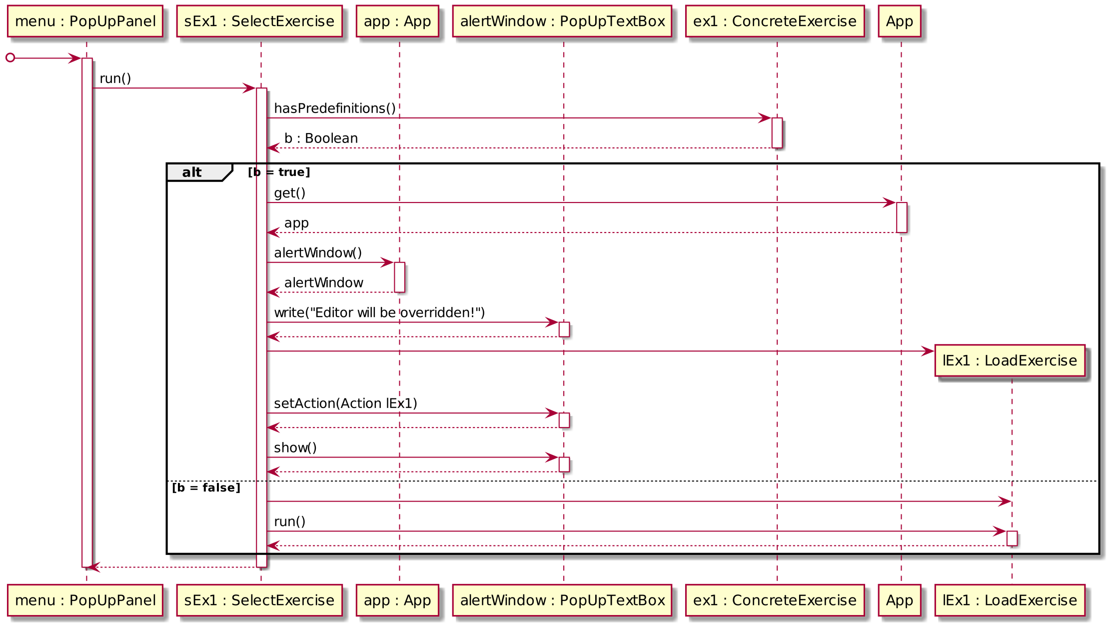


Figure 3: Sequenze diagram for selecting an exercise from the main menu.

4.4 Example action: Load selected exercise

The `LoadExercise` action loads a selected `ConcreteExercise` into the UI.

It updates the width of the `Editor` and displays `TextFields` that are responsible for displaying solution and explanation of the exercise. The action also writes the correct content into the `TextFields` and updates the content of the `Editor` if necessary.

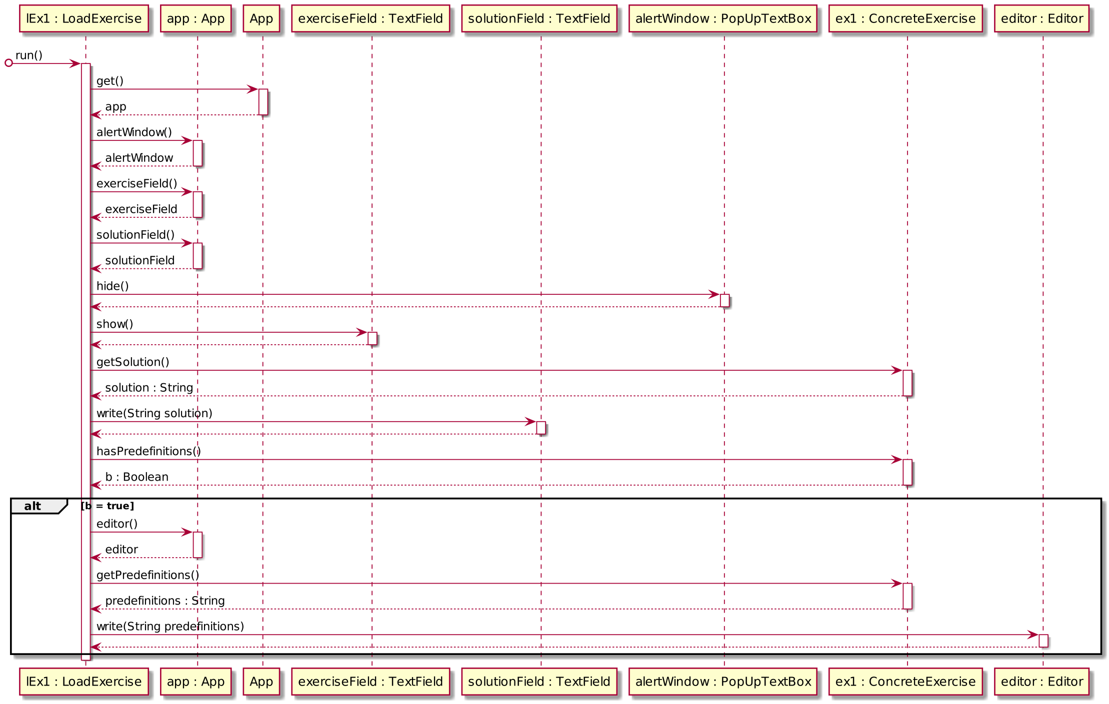


Figure 4: Sequenze diagram for loading a selected exercise into the UI

4.5 Next term

In order to retrieve the next redex to reduce from its reduction order, the `ExecutionEngine` calls the `next` method on the current reduction order. This method creates a new visitor which traverses the lambda term, looking for the correct redex. In order to identify redexes, specialized visitors that determine the type of a given lambda term are used. These visitors extend `NameAgnosticVisitor`, so that for example a named redex is still recognized as a redex.

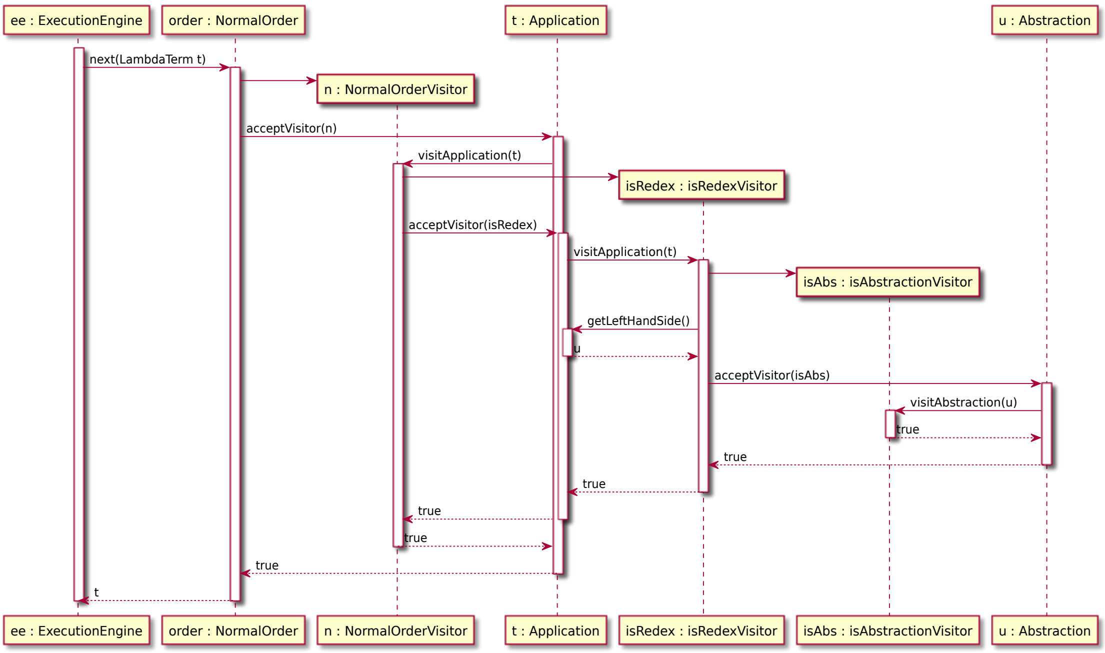


Figure 5: Sequence diagram for choosing the next term to reduce. In this example, we have $t = (\lambda x.x)(\lambda x.x)$ and $u = \lambda x.x$.

4.6 β-reduction

After the next redex to reduce has been retrieved from the reduction order (see **??**), the β-reduction of the redex can take place. Reduction is performed by the `BetaReducer` class, which traverses the tree and reassembles it unchanged until it finds the redex returned by the reduction order. It then creates a `SubstitutionVisitor` which traverses the abstraction and performs the actual substitution. Both classes extend `TermTransformer`, so they make sure that if they encounter a `NamedTerm` whose body changed as a result of the substitution, the name tag is removed.

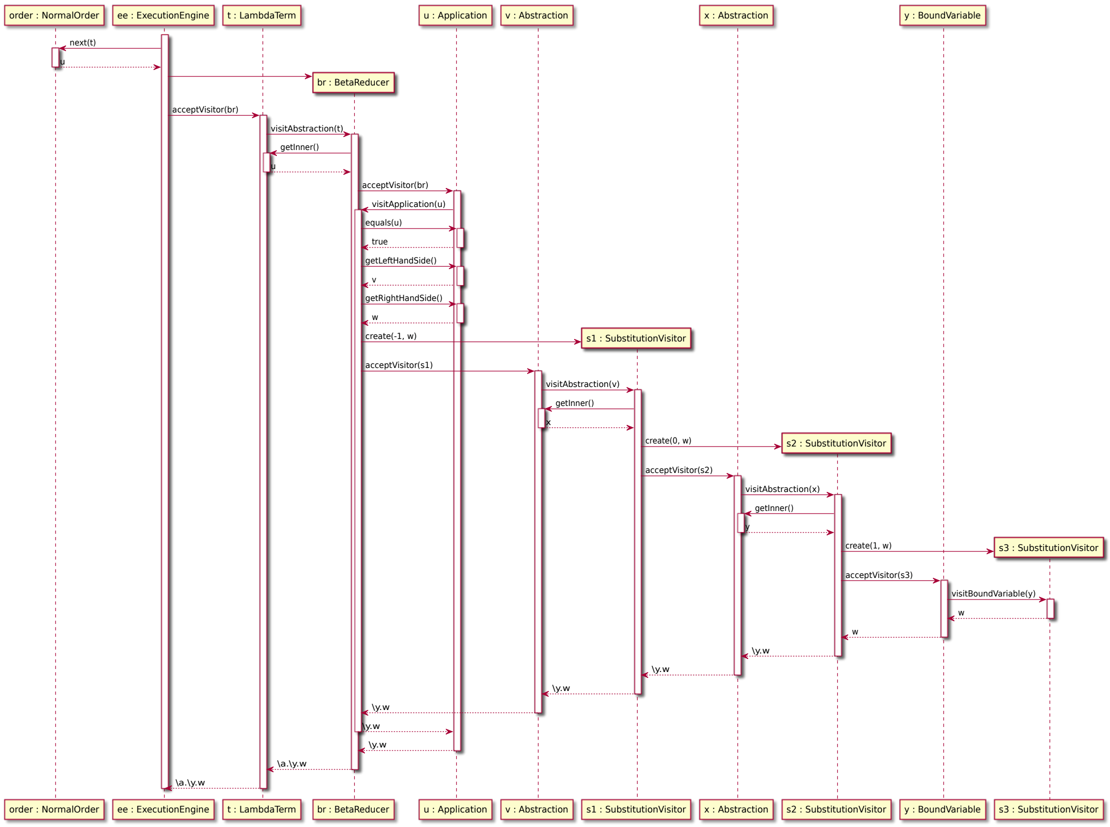
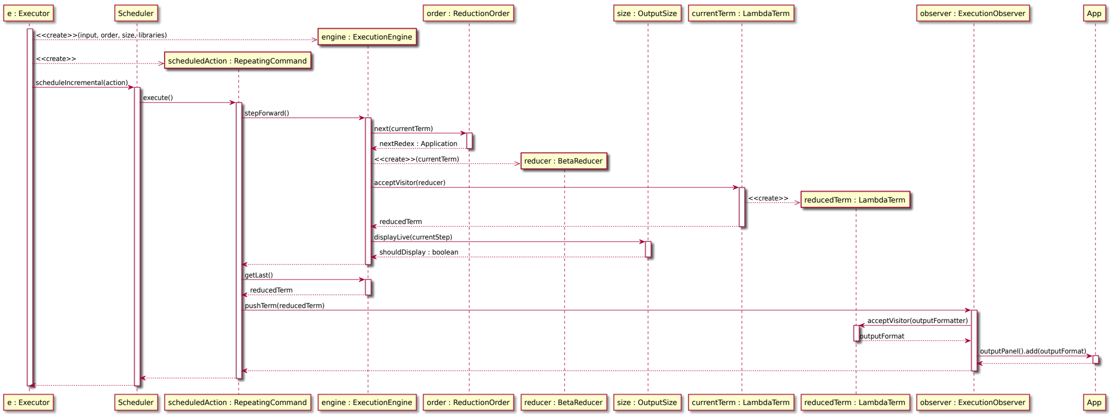


Figure 6: Sequence diagram for a β-reduction. In this example, we have $t = \lambda a.((\lambda x.\lambda y.x)(\lambda x.x))$, $u = (\lambda x.\lambda y.x)(\lambda x.x)$, $v = \lambda x.\lambda y.x$, $w = \lambda x.x$, $x = \lambda y.x$ and $y = x$. Note that in the terms called x and y, the variable x is actually represented as the De Bruijn index 2 (and not as a free variable).

4.7 Interaction between model and view for a reduction

This diagram shows how `model` and `view` interact when a reduction is performed. Upon calling `start()` on `Executor`, a new `ExecutionEngine` is created with the given options. It then schedules an action on the GWT scheduler to run at the end of every event loop iteration. The action executes a step on the engine, gets the new reduced term from the engine and updates all observers. The observers then create a view for the new term with their own visitor and add it to the view.


e : Executor
Scheduler
order : ReductionOrder
size : OutputSize
currentTerm : LambdaTerm
observer : ExecutionObserver
App
<<create>>(input, order, size, libraries)
engine : ExecutionEngine
<<create>>
scheduledAction : RepeatingCommand
scheduleIncremental(action)
execute()
stepForward()
next(currentTerm)
nextRedex : Application
<<create>>(currentTerm)
reducer : BetaReducer
acceptVisitor(reducer)
<<create>>
reducedTerm : LambdaTerm
reducedTerm
displayLive(currentStep)
shouldDisplay : boolean
getLast()
reducedTerm
pushTerm(reducedTerm)
acceptVisitor(outputFormatter)
outputFormat
outputPanel().add(outputFormat)
e : Executor
Scheduler
scheduledAction : RepeatingCommand
engine : ExecutionEngine
order : ReductionOrder
reducer : BetaReducer
size : OutputSize
currentTerm : LambdaTerm
reducedTerm : LambdaTerm
observer : ExecutionObserver
App