textuml

TextUML Toolkit Documentation - Back to main page

TextUML Tutorial

This tutorial shows how to use TextUML for creating UML models. It starts from a minimal model and then increasingly adds more and more features to it.

TextUML is a textual notation for the popular Unified Modeling Language instead of a new language. It is a common misconception that UML is about boxes and arrows. There is a graphical notation for UML, but the semantics are defined completely independently of the graphical notation, so alternative notations, including textual ones such as the TextUML, are possible. All the concepts and mechanisms exposed in TextUML are defined by the UML specification. This is **not** an introduction to UML (if that is what you are looking for, you might be interested in the UML 101 article).

Preparation

There are two tools available that support TextUML to choose from: *Cloudfier* and the (older) *TextUML Toolkit*. The former is a web-based IDE, all you need is a web browser and an internet connection (which I guess you are using right now). The latter is an Eclipse plug-in - you need to have Java, Eclipse and then you must install the plug-in.

Using Cloudfier (nothing to install)

1. go to http://cloudfier.com and hit the Prototype Now button
2. Sign up, confirm email, log in
3. Create a folder (named 'tutorial', for instance)
4. Open the shell page (3rd button on the vertical toolbar
5. if your initial directory is not the one you just created, type 'cd tutorial' to enter it (you can type 'help' for other commands)
6. type: 'cloudfier init-project' (this will mark this directory as a Cloudfier app)
7. type 'cloudfier add-namespace first' (this will create a source file named first.tuml containing an empty package in the current project)
8. type 'ls' - you can edit any of the files by clicking on them. You want to open the 'first.tuml' file yu just created
9. you can now use the (very basic) content assist to add classes, attributes, operations and state machines, just be aware that:

- there must be one top-level package declaration per source file
- classes, datatypes, enumerations and other classifiers must be declared inside packages
- attributes, operations and state machines must be declared inside classifiers
- all types defined in mdd_types are available to you: String, Memo, Integer, Double, Date, Boolean, no need to import the package as suggested later

For more information on how to use Cloudfier, visit the Cloudfier Documentation.

Using the TextUML Toolkit (for Eclipse users)

1. if you haven't yet, install the TextUML Toolkit into Eclipse
2. start Eclipse
3. create a new MDD project named "tutorial" by using the project creation wizard (File > New > Project... > MDD > MDD Project)
4. create a text file named 'mdd.properties' (the repository properties file) at the root of the 'tutorial' project with a single line containing: mdd.enableLibraries=true
5. inside the project you just created, create a new file named 'inventory.tuml'

After the last step, since the file will be empty, there will be a problem marker telling you that a package declaration is expected. That takes us to the first step in this tutorial...

Creating an empty package

The first element to be created in a UML class diagram is a package. The following snippet creates an empty package named 'inventory':

```
package inventory;

end.
```

Notice that when you save the source file, the builder kicks in and a new 'inventory.uml' file appears in the project. That is the actual UML model. It can always be derived from the TextUML source file, or in other words, it is the object code. So if you use a team repository, you do not have to share the compiled model files, only the source files (which are easier to do comparisons, for instance).

Creating a new class

Let's now edit the inventory.tuml file and declare a 'Product' class:

```
package inventory;

(* This is the Product class *)
class Product

end;

end.
```

To declare an interface instead of a class, you just replace the keyword class with interface.

Element comments and source-level comments

Also, note in the previous example how comments are declared. There are actually two kinds of comments in TextUML. Comments that should appear in the resulting UML model (element comments) use Pascal-like delimiters '(*' and '*)'. Ordinary source level comments that are just ignored by the compiler use C-like '/*' and '*/' as delimiters.

Basic types

The following basic data types (among others) are available in mdd_types:

- String
- Memo
- Integer
- Double
- Date
- Boolean

Just import the built-in mdd_types model into your application and they should become available:

```
package myapp;

import mdd_types;

class Person
    attribute name : String;
end;

end.
```

Adding attributes to a class

Now that we defined the basic types we needed, let's see how we can declare attributes on the Product class:

```
package inventory;

class Product
  attribute description : mdd_types::String;
  attribute available : mdd_types::Integer;
  attribute price : mdd_types::Double;
end;

end.
```

The syntax for declaring attributes should be quite obvious. Notice the names for the attribute types are qualified. That is because we didn't import the package defining them. If we import the 'mdd_typesda' package, we don't need to qualify the type names:

```
package inventory;

import mdd_types;

class Product
  attribute description : String;
  attribute available : Integer;
  attribute price : Double;
end;

end.
```

Note that if you don't mind using qualified names, there is no need to import packages, you can always refer to elements in any other model in the same project.

Creating associations

Associations are first-class citizens in UML, differently from what occurs in most programming languages. In TextUML, associations are declared as top-level elements.

In order to demonstrate the syntax for defining associations and aggregations, let's first create two new classes named 'Cart' and 'CartItem' in a new 'shopping_cart' package (in a file named 'shopping_cart.tuml'):

```
package shopping_cart;

import mdd_types;
import inventory;

class Cart
end;

class CartItem
  attribute quantity : Integer;
end;

end.
```

Note you declared two classes in the same source file. That is alright. It actually does not matter in what source file an element is declared. The actual package it will be generated into is defined by the package header.

Now, let's create an aggregation between Cart and CartItem and an unidirectional association from CartItem to Product:

```
package shopping_cart;

import mdd_types;
import inventory;

class Cart
end;

class CartItem
end;

association CartItemProduct
  role item : CartItem[*];
  navigable role product : Product[1];
end;

aggregation CartHasItems
  navigable role item : CartItem[*];
  navigable role cart : Cart[1];
end;

end.
```

For all types of associations, the syntax is basically the same - you specify the kind of association by specifying the corresponding keyword (among association, composition and aggregation).

Adding operations

The syntax for declaring operations is quite straightforward. See, for example, how we'd go about adding operations to the classes in 'shopping_cart':

```
package shopping_cart;

class Cart
  operation createItem(product : Product, quantity : Integer) : CartItem;
  operation removeItem(item : CartItem);
  operation checkout();
end;

/* other elements here */

end.
```

Generalizations (a.k.a. subclassing)

To say a class subclasses another, you use the *specializes* keyword. UML allows for multiple inheritance, so you can specify a comma-separated list of superclasses (incidentally, for declaring interfaces implemented by a class, you use the keyword implements followed by a comma-separated list of interface names). See below for an example:

```
package payment;

class PaymentMethod
end;

class Cheque specializes PaymentMethod
end;

class Paypal specializes PaymentMethod
end;

class CreditCard specializes PaymentMethod
end;

class Visa specializes CreditCard
end;

class AmericanExpress specializes CreditCard
end;

class Diners specializes CreditCard
end;
```

```
end.
```

Note that the keyword *extends* is also supported in TextUML but is used when declaring stereotypes.

Conclusion

This ends our tour on using TextUML for creating a simple class model. If you have any questions or suggestions about the article or the tool, or found any bugs when trying it, just post a question to the users forum or open an issue and it will be addressed asap.

See also

- Install Instructions
- TextUML Toolkit Features
- TextUML Tutorial
- TextUML Structural Notation
- TextUML Behavioral Notation
- Discussion group
- Issue tracker
