Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enhancing the Visual Documentation Artifacts in Envision

Bachelor Thesis

Sascha Beat Dinkel

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

www.pm.inf.ethz.ch

October 28, 2014

Supervised by: Dimitar Asenov
Prof. Dr. Peter Müller

Contents

1 Introduction

Envision is a next generation development environment written in C++. Its development was started by Dimitar Asenov during his Master's Thesis [1]. It aims to improve the development experience with graphical representations of language constructs. Since the beginning of software development creating applications is primarily writing source code in a text editor. Research has shown that code visualizations can benefit developers. The goal of Envision is to make program comprehension and navigation easier and therefore improve productivity by structuring the code graphically.

Apart from representing the executable code, Envision also has the ability to graphically support various kinds of software engineering artifacts. These artifacts are used to document the code and consist of enhanced comments (bold, italic etc.) as well as pictures and diagrams. This functionality has been introduced by Jonas Trappenberg in his Bachelor Thesis [2]. The goal of this thesis is to expand the documentation features of Envision by improving the visualizations and interactions of existing artifacts, introducing support for new artifacts such as tables or source code comments, adding the functionality for generating documentation using Doxygen and implementing a basic functionality for generating Alloy models out of code contracts.

1.1 Motivation and Goals

Envision already supports different documentation artifacts, but the original implementation had some drawbacks. Some artifacts which would be useful are not available at all like tables or source code blocks. At the same time other useful artifacts like the diagram editor are available but uncomfortable to use.

Therefore the goal of this thesis is to improve and extend the current state of the documentation features of Envision. Below we list and motivate all the objectives of this thesis.

Tables
Tables are common building blocks in a variety of different applications. Therefore in addition to the already implemented features in the documentation artifacts, support for tables has been added. Tables can contain simple text or other documentation artifacts. The syntax for the documentation artifacts has been extended to describe tables in a reasonable way. Tables can now be used in a similar way to diagrams.

Support for adding source code blocks in comments
Before this thesis it was only possible to have documentation artifacts in the source code, but it was not possible to have source code in the documentation artifacts. This feature has been added. The commented source code is displayed in an appropriate style using the standard code visualizations of Envision but faded out to indicate that it is not real source code.

Improvements of the diagram editor
Envision already supported diagrams as a documentation artifact before this thesis. This included creating and editing diagrams consisting of simple shapes, connecting lines and text. The functionality of this diagram editor was sufficient but its usage was quite cumbersome. Therefore the diagram editor has been improved to make it easier to use. The goal was to make it conveniently possible to draw a diagram similarly to how it is done in standard diagram drawing programs like Microsoft Visio. To achieve that the usage of the mouse has been expanded and a toolbar with the different shapes and commands has been introduced.

Integration of documentation artifacts
Before this thesis documentation could be inserted only at places where a statement could be inserted. Often, it is necessary to document a programming construct like a class or a function itself and not only some place inside a function. Therefore we implemented the possibility to integrate documentation artifacts with all major programming constructs.

Generation of source code with Doxygen comments
Doxygen is a tool for generating software documentation using special comments in the code. It is widely used in practice and therefore we wanted to support it in Envision. To do this we extended Envision to generate source code with Doxygen comments using the information in its AST about the code and the documentation. In order to achieve this, the different documentation artifacts had to be translated into an appropriate Doxygen comment.

Generation and visualization of Alloy models as documentation
Alloy is a language to describe specifications which can then be analysed by the Alloy tool. Alloy also generates images of possible models out of this specification. This functionality suits well to Envision, since it also helps to improve the development by providing appropriate visualizations. We implemented a prototype plug-in for the Alloy integration which generates Alloy code out of code contracts and uses the Alloy library to generate appropriate images. These images can then be displayed in a pop-up within Envision.

1.2 Comments Framework

As mentioned in the introduction there is already a lot of functionality for comments available. The general approach of this work will therefore be to reuse this framework and design new functionalities in a similar way.

In Envision the code is represented as a tree of programming constructs. Every such element is called a node. To visualize the code every such node has an appropriate visualization class. A comment is also represented by such a node of type `CommentNode` which is the base class of all comments. A comment node can be inserted at any place a statement can be inserted. This node then stores every construct which is used inside this comment. The comments written by the user are parsed and translated into appropriate constructs. Simpler features like bold or italic text are described using markdown syntax and stored as plain text. On the visualization side this syntax is translated into HTML and then rendered by Qt. Larger constructs like diagrams have their own node and visualization. These constructs are stored as separate tree structures and not as text. They are created using special keywords in the comment's markdown. If such a keyword

appears an appropriate node will be created and appended in a list of nodes of that type inside the comment node. If the comment gets visualized, the keyword will be replaced by a visualization of the corresponding construct. Since the comment node has a list of all constructs used inside the comment, there is the functionality to refer to the same construct at multiple places in the comment.

In this work there are two extensions which require the adding of new nodes and visualizations: tables and code blocks. Here a similar approach as with diagrams will be used.

1.3 Related Work

There exist many documentation tools and tools for creating and visualizing software engineering artifacts. In this section we discuss a few of them that are more closely related to our work.

1.3.1 Barista

Barista is an implementation framework which offers data structures, algorithms and interaction techniques for creating sophisticated text editors [3]. These text editors work with plain source code files but interpret the code internally as an abstract syntax tree and are therefore able to offer additional features. From our view the main feature is that they allow programming constructs to have a more readable pretty printed visualization in addition to the editable textual representation. Enhanced comments representing more than simple text are also supported. The comments are described using HTML and are rendered directly in the editor. Like many implementations in this area Barista is based on the model-view-controller architecture.

Barista is similar to Envision since it also tries to improve the development by offering additional visualization. However the functionality of Envision goes beyond the one of Barista. Envision is a lot more interactive, so you can edit programming constructs directly in their visualizations. In Barista you often have to go back to a code like view if you want to edit something.

1.3.2 Programs as visual, interactive documents

French et al. present another approach to combine textual and visual programming [4]. They claim that pure visual programming environments are often more inconvenient than useful and also suffer from the scaling up problem. Therefore they choose a hybrid approach, so only elements which really benefit are visualized in a special way. An open source implementation called the Larch Environment is available.[5]

They state that these visual, interactive objects can be embedded within textual source code and source code can further be embedded in those objects. Therefore it is possible to structure actual source code in visual elements like a table. However apart from this visual elements the code itself is still represented as pure text. This is the main difference to Envision where special visualizations are also used to represent the code in order to make comprehension and navigation easier for the developer.

1.3.3 Doxygen

Doxygen is a tool for generating documentation from source code comments of different programming languages [6]. To do this, Doxygen supports a special syntax for the comments to describe various things like tables, pictures, formatted text and so on. Therefore it also aims to improve the documentation features by allowing more functionality than ordinary text, similar to the documentation artifacts of Envision.

Doxygen supports many common programming languages as input like C++, Java or Python. On the output side, Doxygen supports the generation of commonly used formats like HTML or LaTeX.

A difference to Envision is that documentation is not visualized in-line. You first have to write the code with the comments and run the Doxygen tool on it to actually create the visualizations. Another difference is that Doxygen is a lot less interactive, every documentation is described as pure text.

1.3.4 MIT's Alloy Analyzer

Alloy is a language to describe specifications of a model, and a corresponding tool to analyse such models [7]. It can therefore check if such a model is satisfiable and generate instances of it. Although this does not look related to the topic of this thesis at a first glance, an important ability of Alloy is that it can generate diagrams of possible instances. This allows the user to quickly spot instances which are not desirable and therefore to refine the initial specifications.

In the context of Envision we want to integrate Alloy to generate such diagrams from within Envision and show them in-line just like an ordinary artifact. As a source for Alloy we want to use specifications expressed in Envision using syntax resembling Microsoft's Code Contracts for .Net. [8]

In listing 1.1 you see a reference specification of a linked list. The corresponding image in figure 1.1 shows one possible instance of this specification generated by Alloy.

```
1  module LinkedList
2
3  sig LinkedList { root: lone Node }
4  sig Node { next: lone Node }
5
6  fact nextNotReflexive
7  {
8     no n:Node | n = n.next
9  }
10
11 fact allNodesBelongToSomeLinkedList
12 {
13    all n:Node | one q:LinkedList | n in q.root.*next
14 }
15
16 fact nextNotCyclic
17 {
18    no n:Node | n in n.^next
19 }
20
21 pred show() {}
22 run show for 5 but exactly 2 LinkedList
```

Listing 1.1: Specifications of a LinkedList

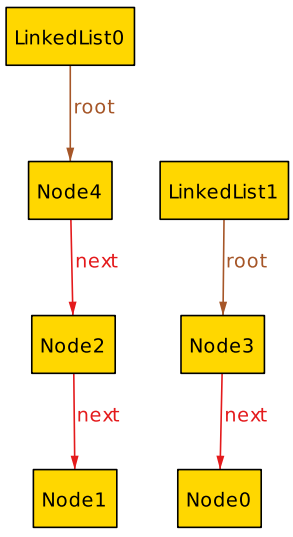

Figure 1.1: Generated instance by Alloy

2 Design

In this section we will describe our work from a high-level perspective. We will outline how we achieved the goals of this work and explain fundamental design decision.

2.1 Diagram editor

In the initial implementation of the diagram editor by Jonas Trappenberg, the command line was used to add shapes or change colors. While using the command line is suitable for creating code in Envision, it is incongruous for manipulating diagrams. The goal of improving the diagram editor is therefore to allow the user to do all operations without using the command line. To do that the main change is to introduce a toolbar which offers all possible operations of selected shapes. Additionally the function of the mouse has been improved, it is now possible to directly drag new shapes into the desired size. Before our work the diagram editor had a separate editing mode, we removed this mode since we did not see a benefit of it. The toolbar offers every manipulation possible in the diagram editor. This includes the following functionality:

- Creating new shapes
- Changing color of background, outline and text of shapes
- Show connection points for connectors

While developing the usability improvements, we discovered that it would be quite easy to implement some useful additional features. Therefore we extended the initial functional requirements with the following ones:

- Arrows for the connectors
- Outline size for the connectors and shapes
- Outline type for the connectors and shapes

To help the user, the toolbar will only offer possible actions according to the current selection. For example the color-selector will only be accessible if a shape is selected.

Another problem was the position of the toolbar and when it will be visible. We decided to use a free floating toolbar which will be displayed at the bottom of an active diagram by default. The user can then move the toolbar to any position they prefer. This functionality is needed if the user works on a large diagram or if one zooms in a lot.

A diagram becomes active as soon as the user clicks on any element of the diagram. If the user clicks on something outside the diagram the toolbar will disappear.

2.2 Source code blocks

For representing code blocks we added a new node. This node consists of a string representing its name and a reference to another node of any type. We call this concept of a node which can contain another node of any type a free node. The contained node can therefore be a code construct like a class or a method as well as another comment. This free node is the basis of code blocks as well as tables.

On the visualization side there is a new class which visualizes the free node. This class just uses the already available visualization of the contained node, but adds a small effect. To indicate that the node referenced is a comment and not part of the actual code, it is faded out.

Ordinarily programming constructs are created using the command line. To separate the creation of real code blocks and code blocks in comments we have chosen another approach. Every free node holds a text field by default. The user can then enter the name of the desired construct and replace the text field with this construct by using a shortcut (ctrl+⏎).

2.3 Tables

For representing tables we added another new node. This node consists of a string representing its name and an array which holds free nodes. By a free node we understand the concept of a node which can contain another node of any type as we introduced in the last section. Additionally there are two integers representing the row and column count of the table.

On the visualization side there is a new class which visualizes the table. This class uses an existing visualization of a grid to display the table. This grid was improved so that it can display grid lines which we use in our table.

We reuse the free node here because the free node offers the functionality to edit and display comments as well as programming constructs. By this we already have all we need to allow the user to create and visualize a variety of different kinds of tables. The user can either use the text box of the free node to write ordinary text, or he can use the functionality to replace the text box with a programming construct or a comment.

2.4 Integration of comments

A limitation of the former implementation was the fact that comments could only be inserted where statements could be inserted. In this section we describe the design decisions we made to assign comments directly to nodes like classes or methods.

2.4.1 Storage of comment nodes

In order to integrate comments with programming constructs, these constructs have to be extended with a comment node. There are two ways to do that. The first is to append this comment node to a higher node up the type hierarchy such that all nodes inheriting from it get a comment. The second is to append this comment node only to specific nodes. We explored both possibilities and list some facts about them below.

Using inheritance
- More flexibility for the user since everything can have a comment
- More beautiful design in terms of object orientation
- More space consumption since every node holds a pointer to a comment
- More care needed to select the right construct to assign the comment

Using specific nodes
- Problem of determining which constructs should have a comment node
- More efficient use of space, since only sensible nodes have a comment

We have chosen a mix between these two options. We want to be flexible and let the users assign comments to everything they want. On the other hand there are types of nodes where comments are seldom necessary and would only waste space. This is the case for really basic construct like integers or text. Additionally the structure containing an integer or text is a `CompositeNode` whose position in the class hierarchy can be seen in figure 2.1. So it is enough to offer the possibility to assign a comment to such a structure. Because of that we decided to extend the `CompositeNode` with a comment attribute.

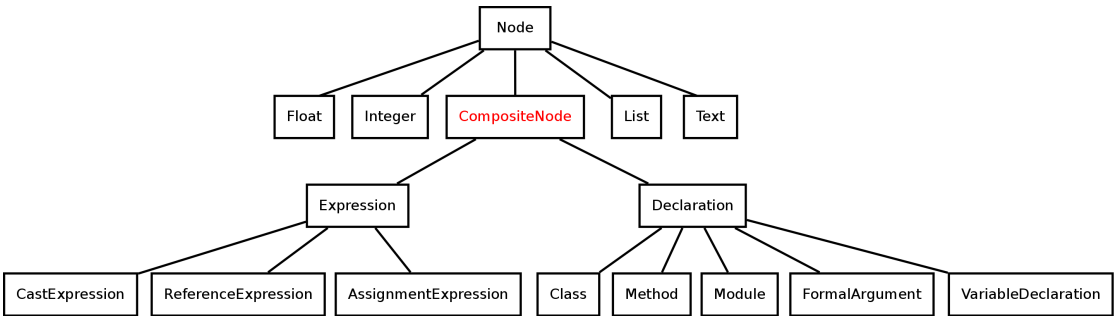

Figure 2.1: Position of the `CompositeNode` in the class hierarchy of Envision

2.4.2 Visualization of integrated comments

The most important design question was how to visualize the integrated comments. We decided to split the programming constructs into two parts where each part has a specific way of visualizing the associated comments.

Bigger constructs (project, module, class, method)
The bigger construct have a visualization of the comment in their header area. Therefore as soon as such a construct has an associated comment it will always be visible. Ordinary comments which are not associated with a particular construct but just appear inside the body of a method will also always be displayed.

Smaller constructs (expressions, statements, etc)
The comments of the smaller construct are not visible by default. They will get visible in a dialogue like overlay if the user presses a shortcut (⇧+F1). In order to know which of the smaller constructs have an associated comment, there is another shortcut to highlight these constructs (F1).

We did this partitioning because it is obvious that we cannot display every comment of every construct all the time. Assume every statement inside a function has a comment, the screen would be crowded with comments. On the other hand the comments of larger constructs are likely more important and there is also more place available for them in the header area of such larger constructs.

2.5 Doxygen Generation

To use Doxygen we have to create an input file which consists of code and special comment features. Since the goal of Doxygen is to describe the interfaces of a project and not implementation details, the generated code does not need to be complete.

Doxygen supports a variety of programming language but we decided to stick to C++, since we are already familiar with it by working on Envision itself. Therefore we generate a single C++ file consisting only of namespaces, classes and empty functions.

2.5.1 Mapping of documentation artifacts to Doxygen

In order to use Doxygen we have to map the documentation artifacts of Envision to an appropriate input for Doxygen. To describe special features in comments, Doxygen is able to understand HTML and markdown. While it would be convenient to only use markdown, some artifacts like tables need to be described with HTML in order to allow nesting.

Since the markdown support of Doxygen is quite new and due to the fact that we discovered some undesired behaviour when mixing HTML and markdown we decided to translate every documentation artifact into appropriate HTML statements. For special artifacts like diagrams or code blocks, there does not exist a direct translation to HTML. For these artifacts an image is generated.

2.5.2 Supported Doxygen functionality

Doxygen offers a lot of special syntax which can be used to describe comments. For our work we only use a small subset of the available Doxygen commands. With this subset we were able to transform all documentation artifacts of Envision to appropriate Doxygen comments. The following table summarizes all features that we used from Doxygen.

feature	Used for
`/**` and `*/`	Ordinary comments about modules, classes and methods
`\param`	Comments about parameters of a method
`\parblock`	Assign multiple lines of comments to a parameter
`\result`	Comment about the result of a method
`\htmlonly`	Tells Doxygen to just accept HTML and do not interpret it
HTML	Used to describe various documentation artifacts for Doxygen

The `\htmlonly` is needed for constructs for which we generate some HTML code and just want Doxygen to accept it and copy it on the generated page. This is especially needed for inline HTML and inline browsers using an iframe.

Note that we only need a few Doxygen features because we express most Envision documentation artifacts directly in HTML. For example insted of using the Doxygen keyword \image we include an image just by using the common HTML code <img src="someImage.png">.

2.6 MIT Alloy Integration

In Envision, specifications can be described using a syntax similar to Microsoft's Code Contracts for .NET. We use Alloy to generate models out of these specifications. To do that we have to translate the code contracts to Alloy code and use the Alloy library to get models from it.

2.6.1 Interaction with the Java based Alloy library

Alloy offers an API to its library. Through this API Alloy's functionality can be used by another program which is exactly what we do. As Alloy is written in Java we cannot directly use it from Envision.

We explored two possibilities to make that interaction possible, namely:

- Use JNI (Java Native Interface) to call Java-code from C++
- Write a small Java program which does all the Alloy operations and communicate on the command line with it

Working with JNI is rather inconvenient because one needs to create special objects which hold all the Java items. For example if you want to pass a string to a Java function from your C++ code you need to create an object of the Java string class and transform your C++ string into it. Therefore you have a crucial overhead in your code by using JNI compared to a pure Java implementation.

Because of that we decided to have a small Java application which does all the operations involving the Alloy library. The question then was interacting with that application by JNI or by command line.

The Alloy library offers functionality to save all all generated models as image files. So we only need to pass three things to our Java application:

- The Alloy model
- The desired output directory
- The maximum number of models we want to generate

The JNI version would give a better performance than doing the detour over the command line, but since the time to pass the arguments is a lot smaller compared to the time required to actually generate the models, this advantage is negligible. Because of that we decided not to use JNI since is does not give us essential benefits.

2.6.2 Displaying the generated model images

We assumed that the usual work flow when using the Alloy functionality would be:

- Write some code contracts
- Go through the models and find undesired instances
- Refine the code contracts

Therefore it is important that you can go quickly through all the models. This can be achieved by displaying all images in a table or grid, or by displaying them in some kind of slideshow. There already exists an overlay which can display some kind of dialog with a visualization in it. Since creating a table, grid or a slideshow can easily be done using HTML and Javascript, we decided to just generate that HTML and use it as an input for this overlay with a browser in it.

Therefore we created a simple HTML page which visualizes the models. This page could be edited outside of Envision and is independent of the rest of the integration. This page is then simply shown whenever the user generates the models. This design is flexible and we can change the presentation of the models by just changing the HTML page and leaving the rest of the integration functionality intact.

3 Instruction manual

3.1 Diagram Editor

Every manipulation on a diagram can now be done by using the toolbar. The toolbar is displayed at the bottom of a diagram as soon as you click on the diagram. In figure 3.1 you can see the toolbar and the features it offers.

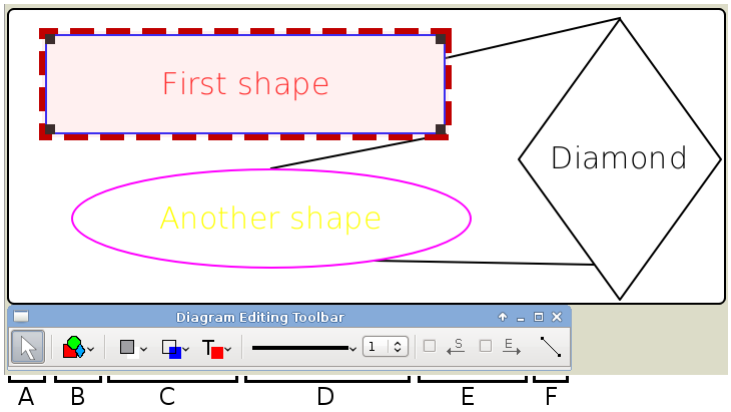

Figure 3.1: The diagram editor and its toolbar

(A) Selection mode	This is the default mode in which you can select shapes or connectors in the diagram and move or resize them. You can also select the text of a shape and modify it.
(B) Shape creation	By clicking on this toolbar item a drop-down menu opens in which you select one of the three supported shapes (rectangle, ellipse, diamond). You can then do a single click on the diagram to create the selected shape at this place with the default size, or you can click and drag it to the desired size.
(C) Color selection	If you select a shape, the toolbar items for the background, outline and text color of that shape become available. By clicking on each of them a color-picker opens in which you can select the desired colors.
(D) Outline selection	If you select either a shape or a connector, the selection for the outline type and the outline size become available. By clicking on each of them a drop-down opens in which you can select the desired parameters.
(E) Arrow selection	If you select a connector, the selection for the arrowheads become available. By setting the check-boxes you can choose if an arrowhead is displayed at the beginning and the ending of the connector.
(F) Connection-points	By clicking on this item, the connection points of the shapes will become visible. You can then click on two such points to establish a connector between them.

3.2 Source code blocks

A source code block can be created by entering the following text in the comment editing window:

```
[code#aName]
```

This will create a free node with the name *aName*. A free node is a node which can contain another node of any type as described in the design chapter. You can use the same name later to refer to that free node.

After creation a text-box appears in which you can write the name of a programming construct. By pressing ctrl + ⏎ the text-box will be substituted with the graphical representation of that programming construct. If you enter something invalid this text will be deleted. The following names for programming constructs are supported.

```
comment, class, method, statement, block, foreach, if, loop, switch,
expression
```

As you can see there is also support for a comment node inside a free node. The purpose of this functionality is mainly to allow comments inside tables, since a table consists of free nodes. You can read more about that in the next section about tables. The programming constructs in a free node are displayed faded out to distinguish them from constructs used in the real environment. However if such a construct has focus the fading will temporarily be removed. The fade out effect is shown on figures 3.2 and 3.3

Figure 3.2: An unfocused code block inside a comment

Figure 3.3: A focused code block inside a comment

To delete the current content of the code block and make it accept new commands, press ctrl + Delete while the code block has focus. This will bring back the text-box.

3.3 Tables

A table can be created by entering the following command in the comment editing window:

```
[table#aName#rowCount#columnCount]
```

This will create a table with the name *aName* and the dimensions *rowCount*columnCount*. If you want to refer later to that table you can use the shorter instruction without the dimensions:

```
[table#aName]
```

After creation, the table consists of empty text-boxes. These text-boxes behave exactly like the ones of the code blocks. So you can write ordinary text or use the functionality to replace them with programming constructs. It is worth mentioning here that you can use this functionality to insert comments. With this you can have documentation artifacts like diagrams or pictures inside a table. You can also use the markdown text features for example to design the headers of a table. You see such an example in figure 3.4.

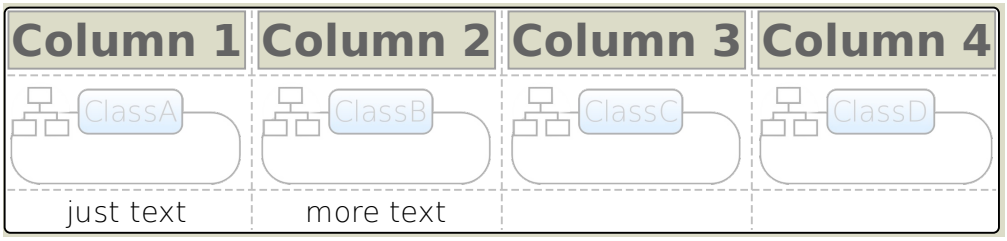

Figure 3.4: Example of a table, the first row are comment fields

3.4 Integration of comments

To create a comment for a specific programming construct, select it and press ⇧+F1. This will create and show a new comment with the default text linked to that programming construct. If the selected programming construct already has a comment, pressing this key combination will just show it.

The comment will be displayed in one of two ways depending on the type of programming construct it belongs to.

If the comment belongs to a large construct like a class, method and so on, it will be displayed in the header of that construct and will therefore always be visible. In figure 3.5 you see an example of such a large construct.

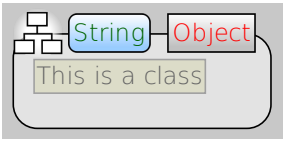

Figure 3.5: Large construct

If the comment belongs to a small construct like an assignment, argument and so on, it will be displayed in a pop-up and will therefore usually be hidden. In figure 3.6 you see an example of such a small construct.

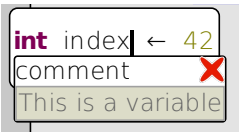

Figure 3.6: Small construct

In order to figure out which of the smaller constructs already have an associated comment you can press F1 to highlight them. You see such an example in figure 3.7.

To close a comment pop-up you can select the appropriate construct and press ⇧+F1 or you can click on the ✖ icon of the pop-up.

To remove the comment you can select the appropriate construct and press ⇧+F2.

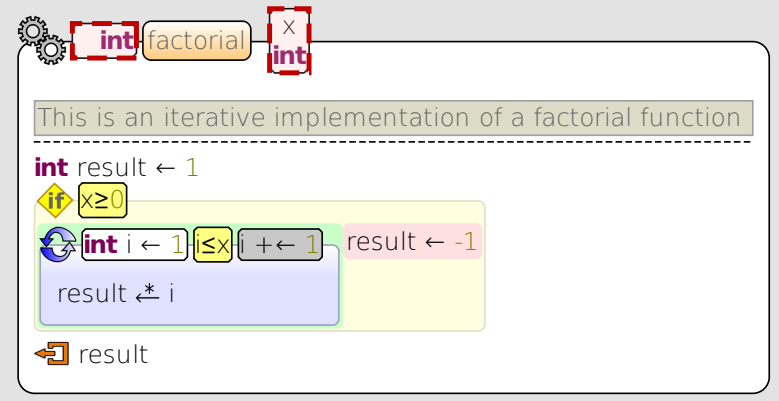

Figure 3.7: Method with highlighted available comments

3.5 Doxygen Export

To export the comments to Doxygen just open a prompt inside Envision by pressing ESC and then enter:

```
doxygen
```

This will create an HTML documentation of the code with all the comments translated to appropriate Doxygen visualizations.

The documentation will be stored in the `doxygen‣html` folder in the directory of Envision's executable.

In figure 3.8 you see a part of a generated HTML page by Doxygen. This part shows the detailed description of a module. There are different documentation artifacts involved in this example: an enumeration, a table, a diagram and a code block.

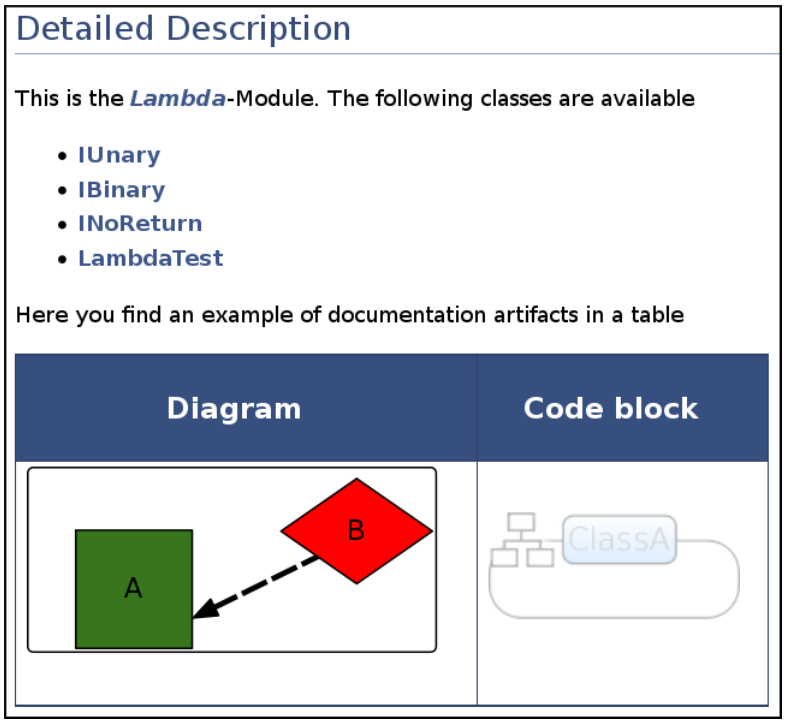

Figure 3.8: Part of an HTML page generated by Doxygen

3.6 Alloy model generation

To generate instances by Alloy open a prompt inside Envision by selecting an object and pressing ESC, and then enter:

```
alloy
```

This will generate instances of the current class (the one which has the cursor) and show the instances in a pop-up. In order to do this Envision will first generate Alloy code out of the code contracts used in the class. After that Envision will call the `AlloyIntegrationCLI` which generates the instances out of this Alloy code using the Alloy library. Be aware that this computation will take some time, typically a few seconds. In the pop-up you can use the following keys to navigate through all the instances.

Key	Action
Page Up or ← or ↑	go to previous instance
Page Down or → or ↓	go to next instance
Home	go to first instance
End	go to last instance

In figure 3.9 you see an example of such a pop-up showing one of the instances generated by Alloy.

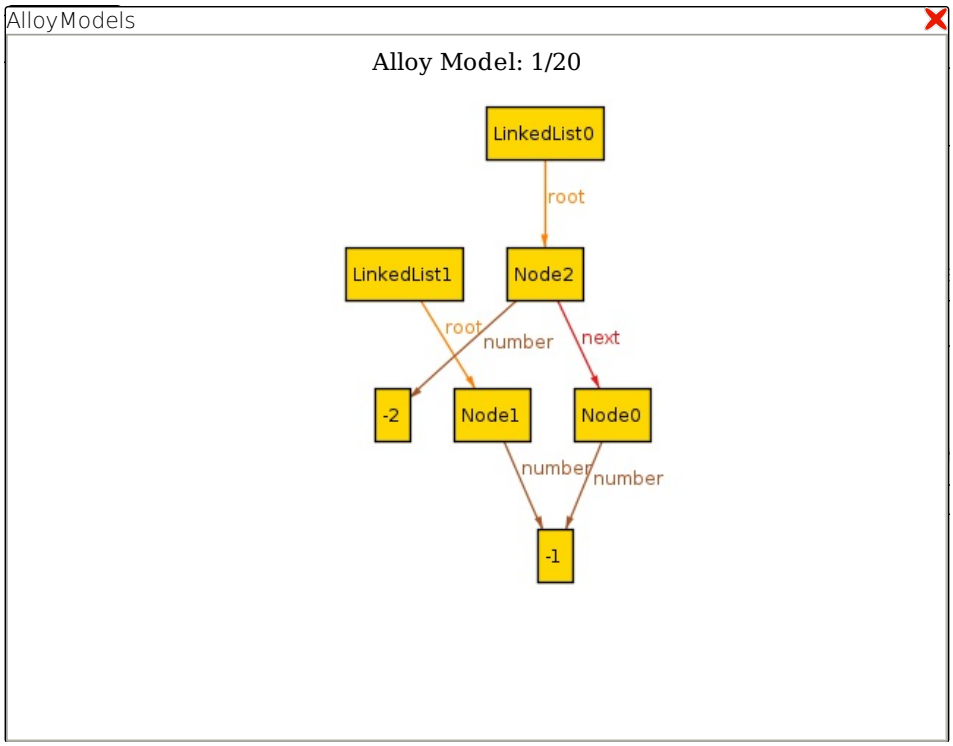

Figure 3.9: One instance out of twenty generated by Alloy

4 Implementation Details

The purpose of this chapter is to give more information about the concrete implementation. To do this we focus on parts of the code which we consider consider important and not straightforward. We explain these parts in details and also provide code snippets.

4.1 Diagram Editor

4.1.1 Arrow drawing

The idea we used was that an arrowhead is a triangle, or better said a polygon of three points. Since Qt allows drawing and transforming polygons easily, we only have to define such a polygon and rotate and translate it to the right place and draw it there.

Since the starting point of this polygon is the tip of the arrowhead, we just have to translate it to one of the two points which defines the line. To rotate it correctly we need the slope of the line which can be calculated by $m = \frac{y_2 - y_1}{x_2 - x_1}$. Fortunately Qt offers some functionality to get that slope quite easily as can be seen in listing 4.1.

```
1 double angle = -QLineF(startPoint_, endPoint_).angle();
```

Listing 4.1: Getting the slope of the line

After we have all the information needed, we can place and draw the arrowhead. To do this we use Qt functions to avoid manual computation. After the arrowhead is placed the corresponding point of the line has to be replaced, otherwise the tip of the arrow would be overlaid by the line.

In listing 4.2 you see the code which implements this functionality. In line 3 and 4 the coordinate system of a matrix is first rotated by the slope of the connector and then applied to the polygon which represents the arrowhead. After that in line 5 the arrowhead is translated to the starting point of the connector and drawn to the scene in line 6. Finally in line 7 and 8 the starting point of the connector is adjusted to avoid the overlay problem.

```
1 if (node()->startArrow())
2 {
3     matrix.rotate(angle);
4     arrow = matrix.map(anArrowhead);
5     arrow.translate(startPoint_);
6     painter->drawPolygon(arrow);
7     newStart = QPointF((arrow[1].x() + arrow[2].x())/2,
8                        (arrow[1].y() + arrow[2].y())/2);
9 }
```

Listing 4.2: Drawing the starting arrow

4.1.2 Color picker

Most functionality of the diagram editor's toolbar is implemented using basic graphical control elements provided by Qt, like `QCheckBox` for setting start and ending arrows. The only exception is the color picker. Qt provides a graphical control element for choosing colors (`QColorDialog`) but this is too heavyweight for our purposes.

Our color picker consists of a grid of buttons which represent the possible colors. You can see an image of the color picker in figure 4.1. To handle the events if a button is pressed we use the Signals & Slots mechanism of Qt. If a button is pressed a signal is emitted containing a `QString` with the rgb value of the color. This signal is send to a slot of the toolbar which then actually applies the rgb value to the selected shape. Since every button emits the same signal just with a different color, we use the `QSignalMapper` to manage this. In listing 4.3 the core part of the signal mapping is shown.

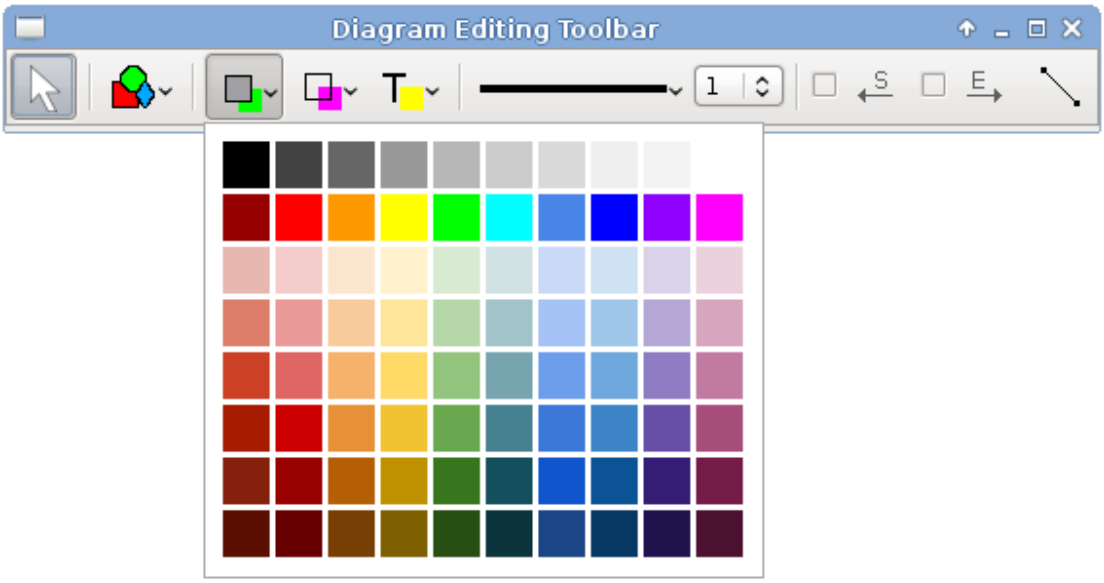

Figure 4.1: The toolbar with the color picker

```
1 QSignalMapper* signalMapper = new QSignalMapper(this);
2 connect(signalMapper, SIGNAL(mapped(QString)), this, SLOT(handleColorPicked(QString)));
3 for (int i = 0; i < colors.size(); i++)
4 {
5         signalMapper->setMapping(aButton, colors.at(i).name());
6         connect(aButton, SIGNAL(clicked()), signalMapper, SLOT(map()));
7 }
```

Listing 4.3: Event handling of the color picker

4.2 Tables

Tables are internally represented as one dimensional lists. This is because the possible types of attributes are limited and something more convenient like a list of lists cannot be used. However the interface to access nodes in a table uses coordinates to simplify its usage.

4.2.1 Resizing of Tables

Resizing a table could be done by creating a new array with the desired size and copying all nodes from the old table within range. Empty elements could be filled up with the default element, in this case a default `CommentFreeNode`.

Unfortunately this does not work, because a copied node still has the old table as parent. We have to use the `replaceChild` method of a node first to replace that node with something arbitrary, in our case a new default `CommentFreeNode`. After that the node does not belong to the old list any more an can be appended to the new one.

At the end we have to use the `replaceChild` method of the table itself to replace the new table with the old one. In listing 4.4 you see the `resize` method of the Table.

```
1  void CommentTable::resize(int m, int n)
2  {
3      auto aList = new Model::TypedList<CommentFreeNode>;
4      for (int i = 0; i < n; i++)
5      {
6          for (int j = 0; j < m; j++)
7          {
8              if (j < rowCount() && i < columnCount())
9              {
10                 CommentFreeNode* aFreeNode = nodeAt(j, i);
11                 nodes()->replaceChild(aFreeNode, new CommentFreeNode(nullptr, ""));
12                 aFreeNode->setName(name()+"_"+QString::number(j)+"_"+
13                     QString::number(i));
14                 aList->append(aFreeNode);
15             }
16             else
17                 aList->append(new CommentFreeNode(nullptr, name()+"_"
18                     +QString::number(j)+"_"+QString::number(i)));
19         }
20     }
21
22     replaceChild(nodes(), aList);
23     setRowCount(m);
24     setColumnCount(n);
25 }
```

Listing 4.4: Resizing a table

4.2.2 The `GridLayout` and its style

We use the available `GridLayout` class to actually visualize the table. We have to create a `QList< QList< Model::Node*> >` from the nodes in our table and pass it to the `GridLayout`.

The only visualization problem here is that we do not have grid lines. To solve that we extended the style file of the `GridLayout` with information about the desired grid lines. Then we added a paint method to the `GridLayout` which actually draws the lines. The default style is that no lines will be drawn, so other parts of Envision which use the grid are not affected.

In the style file of the table we added the relevant information about the grid lines.

4.3 Integration of Comments

4.3.1 `CompositeNode` and Attributes

By default a `CompositeNode` does not have attributes. To allow the functionality of integrated comments this needs to be changed. Fortunately this can be done by simply commenting two lines of code in the `AttributeChain` class. The needed change can be seen in listing 4.5.

```
1 //if (parentChain == &CompositeNode::getMetaData() )
2 //      return; // a null parent indicates direct inheritance from CompositeNode
```

Listing 4.5: Needed change to allow attributes in CompositeNode

As a drawback, every `CompositeNode` now needs more memory for the additional comment attribute.

4.4 Doxygen Export

4.4.1 Generate source file with comments

In order to create a valid input for Doxygen we have to do two parts. First we have to generate valid C++ source code from Envision's AST and second we have to translate the documentation artifacts to valid Doxygen commands which can be inserted in that C++ code. We use a separate visitor pattern for both parts. The first visitor called `DoxygenWholeTreeVisitor` generates dummy C++ code (empty function bodies) and calls the second visitor called `DoxygenCommentsOnlyVisitor` which creates the actual Doxygen comment. We did this separation mainly to allow the reuse of the Doxygen comment generation in other areas where source code is generated. For example the `JavaExporter` could be modified to enhance the generated source code with Doxygen comments.
The next listing summarizes the functions of the two visitors.

DoxygenWholeTreeVisitor
- Generates dummy C++ code
- Includes generation of modules, classes, methods and fields
- Empty method bodies

DoxygenCommentsOnlyVisitor
- Generates the comment in Doxygen syntax for a specific node
- Can be reused when generating real source code
- Uses HTML to describe comments

In listing 4.6 you see the generation of the source code for a class. The `DoxygenCommentsOnlyVisitor` is called before the actual code generation to get the comment in Doxygen syntax in front of the corresponding construct.

```
Visitor::addType<Class>( [](DoxygenWholeTreeVisitor* v, Class* t) -> QString
{
    QString res = "";
    res += aDoxyCommentVisitor_->visit(t);
    res += "public: class " + t->name();
    if (!t->baseClasses()->isEmpty()) res += " : ";
    for (auto node : *t->baseClasses())
        res += StringComponents::stringForNode(node) + ",";
    if (!t->baseClasses()->isEmpty()) res.truncate(res.length()-1);
    res += "\n{\n";
    for (auto node : *t->fields()) res += v->visit(node);
    for (auto node : *t->methods()) res += v->visit(node);
    for (auto node : *t->classes()) res += v->visit(node);
    res += "};\n\n";
    return res;
});
```

Listing 4.6: Generating the source code of a class

4.4.2 Generating images of comments

Elements like diagrams need to be represented as images in the Doxygen export. For this we need the functionality to turn a `Node` into an image.

The first approach we used was getting the visualization of such a node by calling the `findVisualizationOf(aNode)` method. We then used a function in the `Item` class to turn that visualization into an image.

This approach has two drawbacks. First you can see things like a selection or the cursor on the generated images. Second some nodes cannot be handled this way, because there does not exist a visualization of them yet. This is the case for comments which are hidden by default.

To solve that problem we wrote a generic function which takes a node and creates a visualization of it on a new scene. In listing 4.7 you see this method.

```
void ModelRenderer::renderToSVG(Model::Node* aNode, QString path)
{
        auto aScene = new Scene();
        auto anItem = new Visualization::RootItem(aNode);
        aScene->addTopLevelItem(anItem);

        aScene->updateNow();

        QSvgGenerator* svggen = new QSvgGenerator;
        svggen->setFileName(path);
        svggen->setResolution(90);
        svggen->setSize(aScene->sceneRect().size().toSize());
        QPainter svgPainter(svggen);
        svgPainter.setRenderHint(QPainter::Antialiasing);
        aScene->render(&svgPainter, QRectF(), anItem->sceneBoundingRect());
}
```

Listing 4.7: Generating an image of a node

4.5 Alloy

4.5.1 The `AlloyIntegrationCLI`

We explored different ways to use the `AlloyIntegrationCLI`. We first tried to call it by using JNI. We managed to do that but it was quite cumbersome due to a large overhead needed in the code to create the appropriate Java constructs. Therefore we decided to not use the JNI implementation. For completeness you find this approach in listing 4.8.

```
int main()
{
    JavaVM * jvm;
    JNIEnv *env;
    JavaVMInitArgs vm_args;
    JavaVMOption options;
    options.optionString = "-Djava.class.path=/temp/alloy/AlloyIntegrationCLI.jar";
    vm_args.version = JNI_VERSION_1_6;
    vm_args.nOptions = 1;
    vm_args.options = &options;
    vm_args.ignoreUnrecognized = 0;

    char inputhPath[] = "/temp/alloy/model.als";
    char outputDirectory[] = "/temp/alloy/output/";

    int ret = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
    jclass aClass = env->FindClass("AlloyWrapper");
    jmethodID aMethod = env->GetStaticMethodID(aClass, "main",
        "([Ljava/lang/String;)V");

    jclass stringClass = env->FindClass("java/lang/String");
    jobjectArray args = env->NewObjectArray(2, stringClass, nullptr);
    env->SetObjectArrayElement(args,0,env->NewStringUTF(inputhPath));
    env->SetObjectArrayElement(args,1,env->NewStringUTF(outputDirectory));

    env->CallStaticVoidMethod(aClass, aMethod, args);

    return 0;
}
```

Listing 4.8: Calling the AlloyIntegrationCLI with JNI

We therefore decided to just use a command line interface for that. This results in a much shorter code and better decoupling as you can see in listing 4.9.

```
QProcess aProcess;
aProcess.setWorkingDirectory(QDir::currentPath());
aProcess.start("java -jar AlloyIntegrationCLI.jar " + inputFile + " "
               + outputDirectory);
aProcess.waitForFinished();
```

Listing 4.9: Calling the AlloyIntegrationCLI by command line

5 Conclusion and Future Work

5.1 Future Work

With the work done in this thesis the overall documentation features of Envision could be improved and extended. Nevertheless there is still space for further functionality.
A selection of additional features that would improve Envision's documentation features are listed below.

5.1.1 MIT's Alloy Analyzer

We implemented a basic support to generate Alloy models out of code contracts. As a minimum test case for this we have chosen an implementation of a linked list as a reference and tried to get valid models out of it. This target could be achieved.

On the interaction side with the Alloy library we implemented a small Java program which does all the work on the library and can be called by the command line. The visualization of the generated instances could be solved by using HTML and Javascript and show the result in a pop-up.

On the input generation side for Alloy there is a lot of space for improvements. Actually we only support invariants, pre- and postconditions and quantifiers. Currently it is only possible to describe static models. Functions which change some state, cannot be described at the moment.

The flaws could be tackled in a later work to to improve the Alloy integration.

5.1.2 Recognition and visualization of standard patterns

Design patterns are often used in object-oriented software design. Envision could be extended in a way that it graphically displays the used patterns using the concrete information in the code. A UML class diagram of the pattern with the major components and relationships and a sequence diagram depicting the interactions between the different classes of the pattern could be supported.

This functionality can be achieved in the following two ways:

1. By automatically analyzing the class relationships and inferring what patterns are used there.
2. By manually annotating the code. These annotations could then be used by Envision to visualize the pattern.

The first option is clearly preferable since it does not require additional work from the user, on the other hand it requires a lot more effort to implement.

5.2 Conclusion

There was already a substantial implementation for the documentation artifacts available in Envision before this work. Nevertheless there where also some flaws: uncomfortable to use features like the diagram editor or useful but missing features like tables. Overall we were able to contribute substantial new features to the documentation system of Envision.

We improved the diagram editor and made it more user friendly, we added code blocks and tables and we introduced the possibility to integrate comments with all important programming constructs.

As an additional feature we implemented Doxygen generation. All documentation features of Envision can now be translated into appropriate Doxygen comments. As an output we get a website describing all the modules, classes and method using the documentation we created within Envision.

We also implemented a basic integration with MIT's Alloy analyzer. On the interaction side with the Alloy library and the visualization of the generated instances we were able to achieve a good working solution. On the model generation side we were able to implement a working prototype which can be improved.

Altogether we improved the usability of existing documentation features and added several new functionalities. We are confident that our contributions make the documentation features of Envision even better and therefore may help programmers.

References

[1] Dimitar Asenov, Master's Thesis, ETH Zürich, 2011
Design and Implementation of Envision - a Visual Programming System
URL: http://www.pm.inf.ethz.ch/education/theses/student_docs/Asenov_Dimitar/dimitar_asenov_MA_report

[2] Jonas Trappenberg, Bachelor's Thesis, ETH Zürich, 2013
Supporting documentation artifacts in Envision
URL: http://www.pm.inf.ethz.ch/education/theses/student_docs/Jonas_Trappenberg/BA_report_Jonas_Trappenberg

[3] Andrew J. Ko and Brad A. Myers, Montréal, Québec, Canada, CHI 2006
Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors
URL: http://dl.acm.org/citation.cfm?id=1124831

[4] G. W. French, J. R. Kennaway and A. M. Day,
School of Computing Sciences, University of East Anglia, Norwich, Norfolk, 2012
Programs as visual, interactive documents
URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.2182/abstract

[5] The Larch Environment: Visual programming that works
URL: http://www.larchenvironment.com/

[6] Doxygen: Generate documentation from source code
URL: http://www.stack.nl/~dimitri/doxygen/

[7] Alloy: a language & tool for relational models
URL: http://alloy.mit.edu/alloy/

[8] Code Contracts User Manual
URL: http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Enhancing the Visual Documentation Artifacts in Envision

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Dinkel	Sascha

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Sisseln, 28.10.2014	S. Dinkel

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.