



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Supporting documentation artifacts in Envision

Jonas Trappenberg

Bachelor's Thesis

Chair of Programming Methodology  
Department of Computer Science  
ETH Zürich

<http://www.pm.inf.ethz.ch/>

**Supervisors:**

Dimitar Asenov  
Prof. Dr. Peter Müller

November 10, 2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation and Goals . . . . .	3
1.2	Related work . . . . .	4
<b>2</b>	<b>Design</b>	<b>6</b>
2.1	Design Goals . . . . .	6
2.2	Functional Requirements . . . . .	6
2.3	Usability . . . . .	7
2.3.1	WYSIWYG . . . . .	7
2.3.2	HTML . . . . .	7
2.3.3	MarkDown . . . . .	7
2.4	Performance . . . . .	8
2.4.1	HTML and inline browsers . . . . .	8
2.4.2	Specialized implementation . . . . .	9
2.4.3	Qt's optimized text rendering . . . . .	9
<b>3</b>	<b>User interface</b>	<b>10</b>
3.1	Comments . . . . .	10
3.1.1	Syntax . . . . .	11
3.1.2	Inline Markdown . . . . .	11
3.1.3	Block Items . . . . .	12
3.2	Interaction . . . . .	14
3.2.1	Images . . . . .	15
3.2.2	Diagrams . . . . .	15
<b>4</b>	<b>Implementation Details</b>	<b>17</b>
4.1	Model representation . . . . .	17
4.1.1	Comments . . . . .	17
4.1.2	Diagrams . . . . .	17
4.1.3	Diagram shapes . . . . .	17
4.1.4	Diagram connectors . . . . .	18
4.2	Implementation side . . . . .	18
4.2.1	Diagrams . . . . .	19
<b>5</b>	<b>Conclusion and Future Work</b>	<b>20</b>
5.1	Future work . . . . .	20
5.1.1	Formatting . . . . .	20
5.1.2	Code Connections . . . . .	20
5.1.3	UML diagrams . . . . .	21
5.1.4	MIT's Alloy analyzer . . . . .	21
5.2	Conclusion . . . . .	21

# 1 Introduction

## 1.1 Motivation and Goals

Many studies have been conducted to research on human perception of images and text. From these, we can conclude that human beings are first and foremost visual creatures. While we read text in a linear sequential fashion, we perceive images in two dimensions simultaneously, i.e. a lot faster. This observation even led to companies which base their whole business model on it, such as e.g. *Billion Dollar Graphics* [1], which endorses replacing text by graphics wherever possible. Their site also offers a nice overview of reasons why to prefer graphics over text [2], citing many interesting papers on the way that discuss this very topic.

Knowing this, one has to wonder why programming as of today still consists of inputting a large number of lines of text, just as it has for the past 60 years. Even though program sizes have been increasing steadily ever since then, leading to vast projects today that one single mind can barely grasp in their entirety, the method of describing programs has not dramatically changed. This discrepancy has led to more and more attempts over the last few decades at enhancing the construction of programs by graphical elements. These try to better exploit the visual nature of humans, with the ultimate goal of facilitating the reading and writing of programs.

One such approach is researched by *Envision*[3]. Envision is a new programming environment that aims to become a next generation software development system. Envision researches graphically inputting and arranging program code in new meaningful ways. Instead of presenting the user with mere plain text, every language construct has a corresponding two-dimensional graphical representation, which can be styled individually, without recompiling the whole codebase, through the means of XML files.

Envision's approach clears the way for a number of useful applications, such as locally cumulating logically related classes. We think this will enhance the process of reading large chunks of code. For more information about Envision, see Dimitar Asenov's Master's thesis [4].

However, what is even more important than the way of describing programs is documentation. Even the most naturally structured code is hard to understand if there is no documentation that accompanies it. But even comments may not only be textual descriptions of the program code. Many projects include images and diagrams in their documentation (e.g. [5], [6] or [7]). Clearly, the need of including diagrams and images exists, as there is even a helper program, called *PyNsource* [8], that was created solely for the purpose of generating ASCII diagrams. Part of such a diagram that was generated by PyNsource can be seen in figure 1.

Envision already offers support for most semantic structures of Java and some C++, but in both cases, there is no support for documentation at all. However, Envision's graphical nature provides a unique opportunity to embed a wide range of documentation artifacts directly within the program's source. The goal of this project is to use this opportunity to tackle Envision's current lack of documentation. In doing so, instead of only being able to display comments that

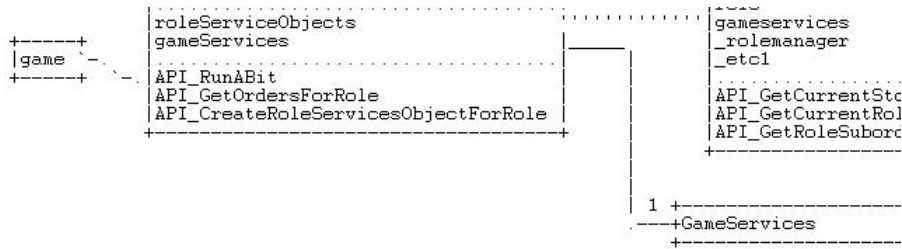


Figure 1: An ASCII UML map, generated by PyNsource

contain nothing but the historical plain text, we would also like to display text structure, i.e. titles and separating lines, richly formatted text, including bold, underline and italic text, lists, images, diagrams and inline browsers for directly exploring external documentation without having to leave Envision.

In the next section, we will shortly discuss other existing approaches that have been made in this direction. In the following chapters, we will first discuss some design issues that we had and decisions that we made to account for these. We will then continue with a short guide on how to use our implemented documentation framework from the standpoint of the user. Afterwards, we go into more detail on the implementation and internal representation of the comment structures. Finally, we draw some conclusions about this project and take a peek at some future extensions that could be built on top of this framework.

## 1.2 Related work

Many programs have been developed in order to present documentation in different ways. Among the more traditional ones are Doxygen and wikis:

- **Doxygen** [9] automatically generates different output formats from a special syntax that can be used inside comments, that describes functions, their arguments, links between functions, etc.
- **Wiki** [10], is a kind of collaboration platform which emphasizes everyone working together on the same articles, gathering the knowledge of many people in one place. This is especially useful in the programming domain, as this corresponds to the same workflow many developers go through, adding to code that other developers have started. Rather than one single wiki implementation, there are many different ones, Wikipedia [11] being the most prominent one.

These approaches have in common that the developer must always keep open a second window, typically a browser window, that explains the code. While reading the code, whenever one does not understand part of the code, one is required to switch to the documentation, read up on what the original developer wrote about that particular piece of code and switch back.

Other approaches have been developed, which are meant to facilitate code comprehension. Some notable of these approaches are listed below.

- **Light Table** [12] offers real time feedback about data flow through the program while coding. The developer can change a variable and follow the path of this data down

to its presentation and therefore gain wider understanding of the underlying program architecture.

- **IPython** [13] exploits the visual nature of the human eye and adds graphical items to the typical python terminal. Objects can not only be represented as strings but actually be formatted with full HTML, CSS and JavaScript support, allowing for any imaginable object visualization. This again facilitates the understanding of the program by improving the representation of individual objects.
- **Larch** [14] extends the Python environment by “visual, interactive objects to be embedded within textual source code”. This eases understanding of the code by being able to experiment with it and immediately seeing graphically rich results. As opposed to Envision however, apart from these rich results representations, Larch is still purely textual.
- **Barista** [15] was created as a framework for embedding interactive tools, annotations and alternative views in the code itself. As such, it offers some of the functionality that we provide as well and may have developed in a similar direction as Envision, but was not pursued further after 2008.

None of these approaches so far are satisfying, as code documentation should first and foremost be as easily and comfortably accessible, i.e. as close to the code as possible, in order to make its understanding faster and therefore easier.

## 2 Design

### 2.1 Design Goals

In designing our system, we are aiming for three different goals, which, in short, are:

- **Functionality** – We want the user to be able to structure comments, i.e. add titles and lines, formatted text, lists, diagrams, images and browsers.
- **Usability** – Inputting text comments should be as easy as usual, even when including formatting. Creating diagrams should be intuitive and direct, without having to rely on ASCII art. Images and browsers should be specified in a convenient way that the user can easily remember.
- **Performance** – We want to be able to display millions of comments at the same time without having too huge of an impact on the overall performance.

Of these design goals, we prioritize performance, as there is no point in having a perfectly user-friendly or fully functional interface if it is too heavy-weight to run smoothly. We would rather leave out some of the functionality than build an irresponsive system.

For the rest of this chapter, we will first elaborate in more detail on what exactly we want the different documentation items to support. Afterwards, we will discuss the performance and usability of different approaches, including which functionality compromises we would have to make in each case.

### 2.2 Functional Requirements

We have decided to support a number of different documentation items. These are:

**Text** The user shall be able to input formatted text, which may consist of italic, bold, underlined and colored text.

**Titles and Lines** To ease the understanding of longer comments, individual paragraphs shall be able to have titles as well as be separable by lines in order to add internal structure.

**Images and Browsers** Images and browser windows shall be displayed and be resizable by the user, in the comment itself as well as with the mouse once it's been created.

**Diagrams** Envision shall also offer basic support for simple diagrams. To create these diagrams, the user should be able to add different shapes, having labels and other properties, and connections between shapes. The shapes' label and colors shall be intuitively modifiable.

## 2.3 Usability

To encourage the usage of documentation, adding new comments should be easy. The interface for creating, editing and, most importantly, reading comments should be as intuitive and user-friendly as it can get. We will now discuss the three different approaches that we considered.

### 2.3.1 WYSIWYG

*WYSIWYG* is an acronym for *What you see is what you get*. The idea is to have an interface in which input and output representation are very close to each other, if not the same. One such editor and very lightweight at the same time is the JavaScript-based TinyMCE [16].

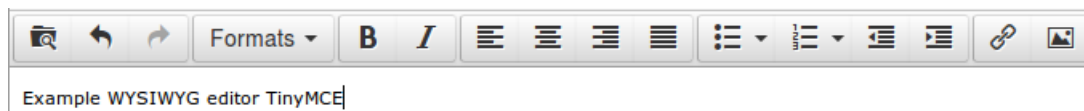


Figure 2: Screenshot of TinyMCE.

It has a common interface, much similar to that of popular writing programs like Microsoft Word or LibreOffice Writer, and thus clearly provides the most user-friendly interface of the three approaches discussed here. Implementing such a user interface, though, would have required a lot more effort and time than we had. We hence decided not to go with this solution, for now. We instead decided to pursue another approach that is easier to implement, admittedly also not quite as user-friendly, to be able to concentrate more on the documentation features than on the input method. Ultimately though, Envision should provide an interface, much like the one pictured above. We will explore more user-friendly input methods in future work.

### 2.3.2 HTML

HTML is a mighty and expressive language, yet this very expressiveness makes the syntax more elaborate and difficult than we need for our simple requirements. Its verbosity makes it very prone to errors: One must e.g. type 7 characters only to indicate that one wants some part of the text to be bold. And if any of these 7 characters are wrong, the parsing is likely to fail. Also, having many tags, i.e. a lot of text formatting, adds a lot of noise to the code, making its input gradually harder to read. It may be plain impossible for an HTML novice to understand what a given piece of HTML code is meant to picture. What is more, we should really not require the developer to learn HTML in order to be able to use Envision. And what if the developer forgets a closing tag, possibly in a large nested list of tags? The error will be visually impossible to miss, but may be hard to spot in the source code.

With this list of difficulties and with another approach in mind that greatly simplifies writing, we decided not to use HTML either, but to instead go for another approach.

### 2.3.3 Markdown

Markdown [17] is a lightweight markup language that is, unlike HTML, based on plain text syntax elements. Its goal is to later be converted to structurally valid (X)HTML. Markdown was

created in 2004 by John Gruber, who, on his website, sums up its philosophy, as: “Markdown is intended to be as easy-to-read and easy-to-write as is feasible. Readability, however, is emphasized above all else.” We are precisely looking for a representation that is easy to write and to read. With Markdown emphasizing readability, this appears like the easiest solution that meets our requirements.

Unfortunately, Markdown was never standardized, which explains the large number of different Markdown implementations. There are a few standard rules at the foundation, but every single one of these implementations adds their own syntax elements as they see fit. Among these are e.g. code highlighting, tags, spoilers, and so on. Likewise, the bare-bones Markdown offers no support for diagrams or inline browsers. We had thus to add some new syntax rules as well, which we will elaborate on in section 3.1.1.

Furthermore, since Markdown was designed to be translated to HTML, it supports inline HTML directly as is. Whenever you wish to use HTML because Markdown’s syntax is not expressive enough, just inline your HTML tags and exhaust all the possibilities of HTML. With our current design, this is only possible within the boundaries of `QStaticText`. Only a small subset of inline HTML is supported, such as colors, bold and italic. We can however reuse the existing browser implementation, but populate it with custom HTML rather than an existing webpage. How to do this will be discussed in more detail in section 3.1.1.

## 2.4 Performance

Bigger projects today typically consist of several million lines of code. For example, at the time of writing, the popular web browser Firefox consists of 9.7 million lines of code and 1.7 million comments, both not including blank lines.<sup>1</sup> What is more, projects keep growing over time as new features are implemented and the number of comments can be expected to only ever increase even more. Clearly, if we want to be able to support projects of any size, we must prepare our system to be able to deal with such a vast number of comments. Of course, resources are limited and we can not guarantee to gracefully handle an arbitrary greater number of comments, but we would like to be able to handle at least a few million comments, as are not uncommon today, without a noticeable hit on performance.

For evaluating their actual performances within the scope of Envision, which is built on top of Qt, we compared Qt’s built-in web browser as well as another built-in mechanism of Qt, designed for efficiently rendering text. Additionally to the benefits and disadvantages of these two existing approaches, we will also discuss the advantages of an own specialized approach. For any general questions or details about Qt, please refer to its documentation [18].

### 2.4.1 HTML and inline browsers

The most straight-forward approach would be to use Qt’s built-in web browser. It instantiates a webkit [19] browser instance that comes with full support for HTML, CSS and JavaScript. Having access to all of these languages, we could easily meet all our specified functionality requirements, including images and even diagrams that could be represented as inline scalable vector graphics (SVG), with almost no additional effort.

---

<sup>1</sup>[http://www.ohloh.net/p/firefox/analyses/latest/languages\\_summary](http://www.ohloh.net/p/firefox/analyses/latest/languages_summary)



On the downside, the more features it provides, which in this case are virtually endless, the more overhead it brings along. We tried instantiating several tens of thousands of comments but even this comparably small amount of comments already brings Envision to its knees. This performance impact is too huge for our purposes, rendering this approach ineligible.

## 2.4.2 Specialized implementation

The other extreme would be to come up with our own specialized implementation, tailored to support exactly what we want and nothing more. Limiting it to only a very tiny subset of all of HTML and leaving all of CSS and JavaScript aside would allow for an optimized implementation that could run at a fraction of the running time of a `QGraphicsWebView`.

Having our very own specialization, however, would also require our own text positioning and our own text parsing, among many more subtle details. While this would clearly be the most performant approach and even though Qt offers some facilities for laying out text manually, e.g. `QTextLayout` [20], we decided not to reinvent the wheel, as it would move the focus of this project away to text rendering.

## 2.4.3 Qt's optimized text rendering

A compromise of the two above mentioned approaches is Qt's built-in `QStaticText` [21]. According to Qt's documentation about `QStaticText`, the “`QStaticText` class enables optimized drawing of text when the text and its layout is updated rarely.” With several thousands of comments and by their nature of documentation that does not change when the code does not, we can expect the comments to stay the same most of the time, making Qt's described use case apply perfectly to our scenario.

Its downside is that “`QStaticText` can only represent text, so only HTML tags which alter the layout or appearance of the text will be respected.” Space of non-textual elements like images will be reserved and the text will float around it, but the image won't be displayed. Likewise, text inside tables will be laid out in table rows and columns but no actual table lines will be drawn. Unfortunately though, this also deprives us from a basic capability like underlined text.

We evaluated the performance of several thousands instances of comments consisting of static text and found that they outperform the equally evaluated web browser comments significantly. Given their high performance, we are willing to cut back on our requirements and accept the lack of underlined text. We may add this in future work, but it will require a different approach than `QStaticText`, resulting in quite some effort.

For performance reasons, we decided to pursue this approach as it is the most viable approach at this stage.

## 3 User interface

In this chapter, we will explain how a user of Envision can most effectively use the newly created documentation framework. This will be close to a user manual, including all possible keyboard and mouse interactions there are to manipulate comments and diagrams, as well as the syntax for the new commands.

### 3.1 Comments

Comments can be placed anywhere where a normal statement can be placed. We would ideally like to put comments anywhere in the program, with the ability of associating them with individual or sets of declarations, but this is not possible yet.

To create a new comment, first put the text cursor inside a statement block at the end of a line or between two lines.

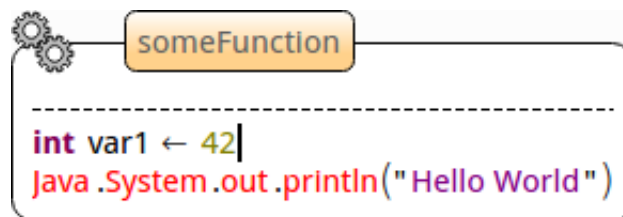


Figure 3: Cursor placement prior to inserting a new comment

Press `↵` to start a new line, then type `//` followed by a space. This will create a new comment that contains one empty line which you can start editing. To create new lines, simply press `↵` again.

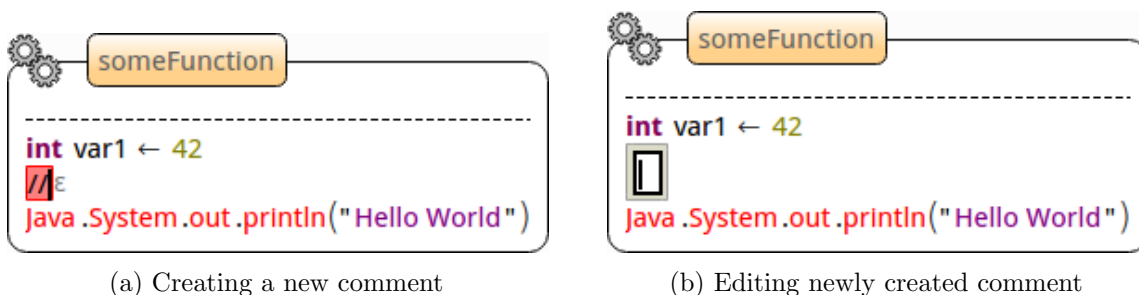


Figure 4

Once you have created this first line, you can start specifying what you want the comment to contain. Simply typing plain text will appear as is. For everything else, i.e. formatted text, titles, images, browsers and diagrams, there is special syntax that we will explain next.

### 3.1.1 Syntax

Markdown already provides some of the features that we want, especially for text formatting. Its original syntax specifies e.g. titles, lines, bold and italic text, which we will happily reuse. We will extend these by some additional syntax rules to include other features that we need.

### 3.1.2 Inline Markdown

#### Text

Text can be formatted with bold and italic text. To get italic text, enclose the text in single stars:

```
This is italic text
```

This is *italic text*

For bold text, use double stars:

```
And some bold text
```

And **some bold** text

#### Headers

The documentation framework supports headers of 6 different levels. To declare a line as a header, prepend it with one to six number signs (#), corresponding to HTML titles from <h1> to <h6>.

```
# Top level header  
##### Sixth level header
```

**Top level header**  
Sixth level header

#### Lists

Lists can be declared by prefixing every list item with “ \* ” (that is: Space followed by an asterisk, followed by a space), which looks like this

```
* First item  
* Second item  
* And so on...
```

- First item
- Second item
- And so on...

Lists can thus far only contain items with single lines, as we do not support line breaks yet. This is a technical restriction from the Envision framework that will be addressed in the near future.

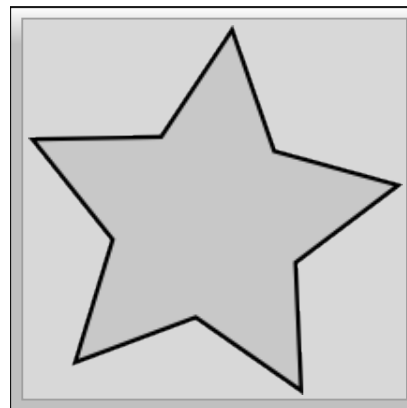
### 3.1.3 Block Items

Some items can not be printed inline. To reflect this, we require the following items, namely images, browsers and diagrams, all to be on individual lines. This is explained in more detail below.

#### Images

Display an image with its original dimensions:

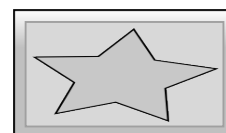
```
[image#star.png]
```



The image formats you can specify here are those accepted by Qt's `QImage` [22]. By default, Qt supports BMP, GIF, JPG, JPEG, PNG, PBM, PGM, PPM, TIFF, XBM and XPM.

Display an image after scaling it to a width of 100 pixels and a height of 50 pixels:

```
[image#star.png|100x50]
```



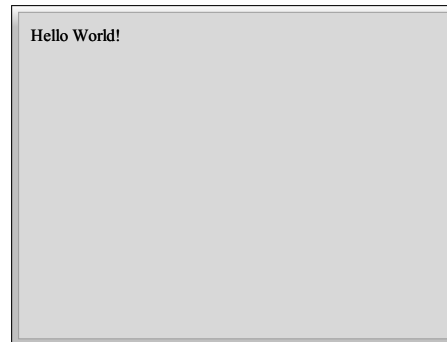
Scale the height (width) to 300 (400) pixels and compute the other dimension such that the side ratio stays the same:

```
[image#star.png|300x]  
[image#star.png|x400]
```

#### Browsers


To display a local file with full HTML, JavaScript and CSS support, use the `browser` command:

```
[browser#/var/www/index.html]
```



The default size, if not otherwise specified, is 400x300 pixels. To change this size, proceed like with images:

```
[browser#/var/www/index.html|150x35]
```

A small rectangular browser window with a thin border. The text "Hello World!" is displayed in the top-left corner of the window's content area.

Likewise, to display any given website, use:

```
[browser#http://www.ethz.ch]
```

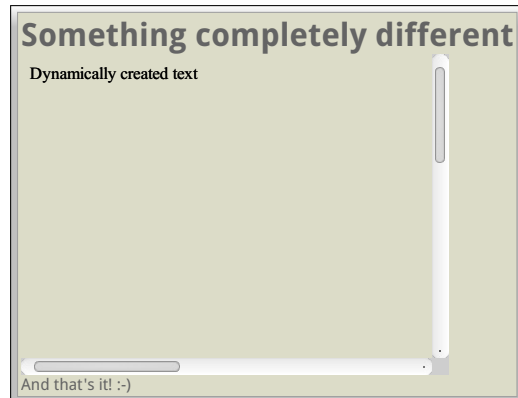


To be able to load external documents, make sure to always include the protocol prefix with the URL, e.g. `http://` or `https://`. If you leave these out, the documents will be considered to be local files and thus not be found.

## HTML

Just like external websites, you can also explicitly ask for a browser instance to express anything that this framework can't represent by using full HTML support. To get this, enclose custom HTML in `<html>` and `</html>` tags, with each on their own line, as shown below. You can even include any JavaScript and CSS.

```
# Something completely different
<html>
<script type="text/javascript">
document.write
  ("Dynamically created text");
</script>
</html>
And that's it! :-)
```



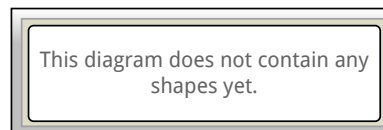
The required size of browsers and HTML blocks can not be determined automatically yet. Both will be created with a default size of 400x300 pixels. To manually specify the size of a custom HTML block, include the size in the starting HTML tag like this:

```
<html 200x100>
```

## Diagrams

To insert a diagram, use the `diagram` command:

```
[diagram#main]
```



This will create an empty diagram that is identified by the name `main`. To add and manipulate shapes and connections, see subsection [3.2.2](#) about manipulation of diagrams.

You can reference the same diagram several times to display it in different places. To avoid name clashes, this only extends to the current comment. Editing a diagram that is referenced multiple times will propagate all changes to all referencing diagrams. E.g. adding, removing and moving of shapes will instantaneously be reflected in all other referencing diagrams. The moving of connectors, however, is not propagated correctly yet.

## 3.2 Interaction

While Envision was designed to always be useable with the keyword, we have decided to add a number of mouse-only interactions to some of the comment items where keyword interaction would fall short. Commands can be executed by rightclicking the shape that you want to operate on and inputting the desired command in the prompt that pops up. Let's look into this in more detail.

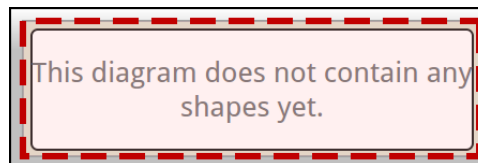
### 3.2.1 Images

Images can be resized by keeping **Ctrl** pressed, clicking the right mouse button and dragging it to the desired position. The text line that references the image will be updated.

Shortcut	Action
<b>Shift</b> + Rightclick	Drag to resize the image

### 3.2.2 Diagrams

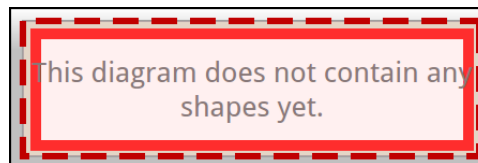
Starting from an empty diagram, that was created as described in section 3.1.3, to add shapes and connectors, you must first focus the diagram by selecting it.



To switch the diagram to editing mode, use the following keyboard shortcut.

Shortcut	Action
<b>Ctrl</b> + <b>E</b>	Toggle the diagram's editing mode. Required for all actions below.

The editing mode is visually highlighted by adding a thick red border around the diagram that is being edited.



This diagram is still quite small. To fit shapes inside, press **Shift** and drag with the right mouse button to resize the diagram.

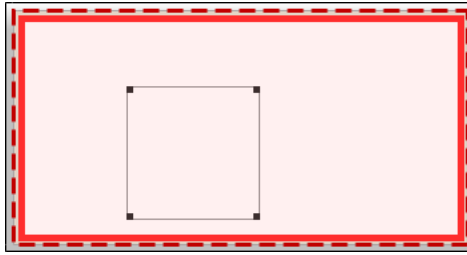
Shortcut	Action
<b>Shift</b> + Rightclick	Drag to resize the diagram

#### Diagram shapes

To create a new shape inside a diagram, rightclick in the diagram to pop up the command prompt. In here, type one of the following commands to create a shape of the specified type. Each one of them gets created with a default size of 100x100 pixels and no label, around the position where you opened the command prompt.

Command	Action
rectangle shape	Create a new rectangle shape
ellipse shape	Create a new ellipse shape
diamond shape	Create a new diamond shape

After resizing and adding a rectangle shape, the diagram will look similar to this:



To add a label to a shape, click the center of the shape in order for the cursor to appear and type its label. The shapes' text, background and border color can be changed with the following commands, respectively.

Command	Action
textcolor <u>color</u>	Sets the shape label's text color to <u>color</u>
bgcolor <u>color</u>	Sets the shape's background color to <u>color</u>
bordercolor <u>color</u>	Sets the shape's border color to <u>color</u>

The background and border colors can be any of Qt's provided `GlobalColor` [23], the text colors need to be defined individually as their own text style in Envision. The currently defined colors are black, blue, green, lime, purple, red, white and yellow.

To move a shape around, click anywhere in the shape and drag it to the desired position. To resize the shape, first click one of the black squares in the corners and drag it until the shape has the wanted size.

Please note that while any moving and resizing of shapes can be undone, every single resize operation, i.e. every movement that the mouse does during a resize, is recorded as a single undo command. This means that undoing a single resize will actually take many more undo steps before the shape is back to its original state. This also means that the default undo stack size of 100 operations will be populated very quickly by moving around, rendering the undo/redo handling rather awkward. This is an issue in Envision's core that will need to be fixed by allowing to merge several related undo operations into one.

### Diagram shape connectors

To display all shapes' connector points, press `[Shift ↑]`. To create a new connector, click the first point of the connector. It will be visually highlighted by being drawn much bigger.

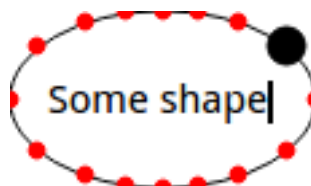


Figure 5: Connecting two shapes by a connector, after selecting the first connection point.

To create the connector, click the second point that you want to connect. Note that both connector points can not belong to the same shape.



## 4 Implementation Details

### 4.1 Model representation

This framework extends the existing model nodes that Envision's standard model provides, allowing for a seamless integration into the existing AST. These are first and foremost the comments themselves as well as the diagrams they enclose.

#### 4.1.1 Comments

Every single comment has its own individual node of type `CommentNode` in the AST. As only the AST is persisted over several sessions, the node, which is part of this AST, must contain any data that we don't want to lose when Envision is closed. We thus had to consider for every element that we added whether it would contain any information that we should keep between sessions. As a rule of thumb, anything the user can change should be kept in the node.

It turns out that this only includes the textual description of the comment and the diagrams that are referenced from inside this description.

#### 4.1.2 Diagrams

Diagrams are of type `CommentDiagram` and contain a list of `CommentDiagramShape` objects and a list of `CommentDiagramConnector` objects. These define the shapes and the connections between the shapes that make up the diagram.

We did not want to include the diagram shapes' and connectors' position and properties in the comment description as describing graphical systems as text is an overly cumbersome task. Assuming the user will not want to input diagrams textually and rather edit them in a nicely presented graphical interface, we decided to leave out the exact shapes' and connectors' properties such as position, size, etc. from the comment's description. We however still have to allow the user to specify where in the comment the diagram shall appear so that we can display some default diagram which the user can interact with. This is the only place where the diagram is referenced.

#### 4.1.3 Diagram shapes

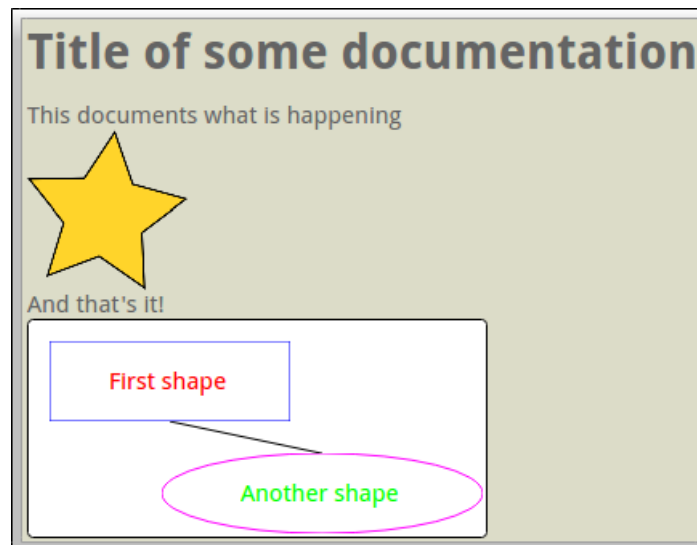
Every shape provides 16 different points which a connector can snap to in order to form shape connections.

#### 4.1.4 Diagram connectors

Every connector's node stores first the index of both shapes that it connects as it appears in the list of shapes in the diagram. Second, it contains the indexes of the connector points of both shapes that it connects.

## 4.2 Implementation side

The comment's description will be parsed at runtime to build up one parent `CommentNode` that includes a number of child items. Some of the child items are e.g. `Text` and `VCommentImage` items, among others. To make this point clearer, take a look at the following comment.



This visualized comment was generated from the following comment description.

```
# Title of some documentation
This documents what is happening
[image#documentation.png]
And that's it!
[diagram#main]
```

The referenced diagram has been manipulated interactively to yield the diagram from the screenshot. We added two shapes, changed some of their colors, and then added one connection between them.

Consecutive lines of text are gathered in one big text block. Anything else is stored as an individual item. The above comment would internally be represented as one big parent `CommentNode` that encompasses all other items, as shown in figure 6.

On the left side, you see the structure of a `VComment`, encompassing all visual child items. On the right side the corresponding internal structure of a `CommentNode`.

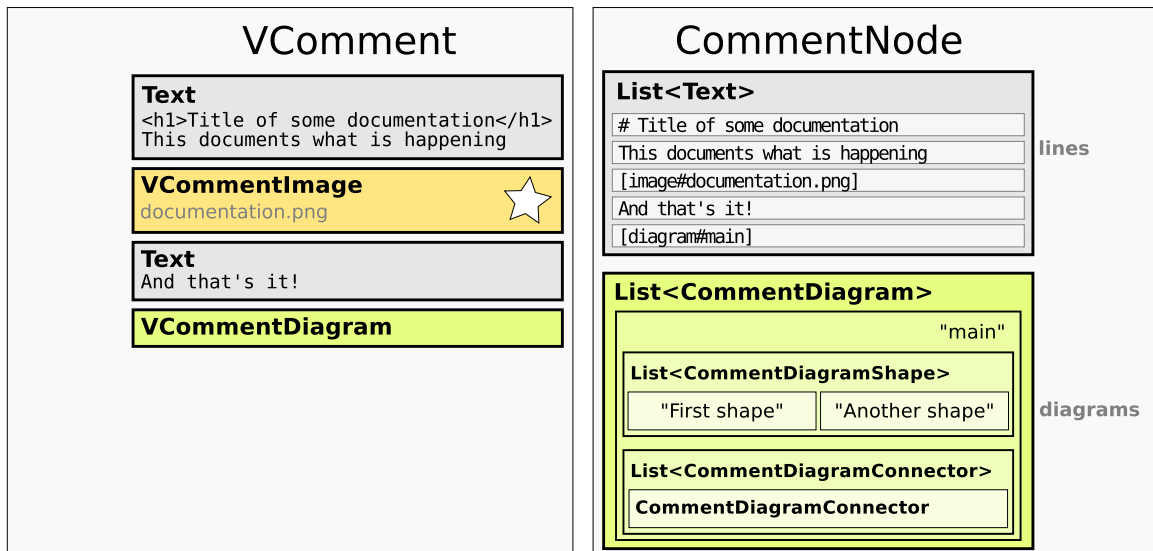


Figure 6: Internal representation of a `CommentNode`. The visual structure on the left, the model representation on the right.

#### 4.2.1 Diagrams

Having a syntax that only says `[diagram]` is not enough as we would not be able to determine which of several diagrams was deleted if their total amount has suddenly shrunk. To track diagrams, we first considered consecutively numbering them using the same syntax, i.e. `[diagram#<number>]`, where each diagram would have its own index in the comment's list of diagrams as `<number>`. But once again, as soon as the user wants to delete a diagram, we would have to reorder all the following diagrams. To not have to deal with this complexity, we decided to assign every diagram a name that is unique within a comment, and track these instead. These can even convey some information about what the diagram contains. Having it say `[diagram#uml-map]` instead of `[diagram#5]` is much easier to relate to.

One issue that we face is that every comment that is deleted will automatically be cleaned from memory right away. This means that an accidental change in name of a diagram that is referenced only once may result in data loss. One might add a timer to each diagram so that its memory gets freed only after a certain amount of time after all references have been deleted, but this sounds like a rather clumsy approach. Ideally, one could simply undo the last change of the comment's description but that's not possible yet, for not perfectly clear reasons. This framework will altogether need some more attention in the future to fully rely on the Envision's command framework, ultimately allowing to undo any change, be it in a comment or in a diagram.

# 5 Conclusion and Future Work

## 5.1 Future work

While very flexible, the infrastructure that was created so far is still quite rudimentary and allows for a lot of polishing. Especially the possibilities for improving the usability of diagram editing are virtually endless. To fully exploit the potential of documentation, we would naturally also like to allow comments anywhere in Envision and not only mixed up with other statements. These could be styled differently depending on the context they are placed in.

A selection of more features that we would like to provide in the future are listed below in more detail.

### 5.1.1 Formatting

Additionally to the existing markup, we would like more block items that could easily be integrated into the current framework without having to come up with a specialized implementation. Among these are

**Tables** Adding tables to the existing documentation would include e.g. coming up with a new syntax for tables while making sure they can't get out of hands like the ASCII diagram shown in the introduction. On the more user-friendly side, it would be desirable though to have some mouse and keyboard interaction to create and change tables comfortably, maybe similar to the current way of managing diagrams.

**Source Code** It would be another nice addition to have the possibility of adding blocks of source code to comments, including syntax highlighting.

There are also some more possibilities which do not actually rely on the framework we created thus far, but which also serve documenting purposes. In the remote future, we imagine to have more complicated features such as the following.

### 5.1.2 Code Connections

Documentation by its nature references the code it documents. Code by its nature however changes frequently, usually requiring all comments that describe this code to be manually updated. Actually enabling parts of a comment to be intertwined with blocks of code could allow for automatically updating documentation to always reflect the latest changes. This might include some refactoring measures such as the renaming of classes or functions or marking comments as outdated when functions or classes they refer to no longer exist.

### 5.1.3 UML diagrams

For starters, the user should be able to create UML diagrams manually and embed them into other comments. Pushing this even further, Envision could automatically generate UML diagrams from parts of the code that the user selected for diagram generation. One could actually link the class and method names to reposition the view to the corresponding piece of code when clicked.

### 5.1.4 MIT's Alloy analyzer

Alloy [24] is a language for describing structures and a tool for exploring them. It has been used for various applications from finding holes in security mechanisms to designing telephone switching networks.

Applied to Software Engineering, one can model a given class structure using the Alloy language and have Alloy generate instances of this structure that satisfy a given set of relationship constraints. This allows the developer to determine whether she has modeled the classes in question correctly.

Ultimately, an Envision user could mark certain classes for model generation and have Envision automatically translate its own class representation into an Alloy model. Given these classes and invariants that are formulated as Code Contracts, Envision could generate possible instances of this class structure and display them in a comment.

## 5.2 Conclusion

We were unfortunately not able to meet all our requirements. Since we wanted a responsive system, capable of dealing with many millions comments at the same time, we had to make compromises in functionality. The inline HTML support for text formatting is very limited, such that we e.g. don't have underlined text. For all other shortcomings, though, the user can rely on block HTML items, enabling him to exploit all of HTML, CSS and JavaScript, allowing for much more functionality. As long as these are not excessively used, the system will keep performing reasonably fast.

As for usability, we decided to spend our efforts more on the functionality side, including diagrams that can be manipulated right from within Envision. The features we implemented are all functional and can be manipulated at runtime, the usability however came a bit short. How to improve this will be explored in future work.

On the implementation side, the adding of new comment items, such as e.g. tables or source code, still has to be parsed manually. Each item also requires its own class with voiler plate code, but doing so is pretty straight-forward. This too may be redesigned to better accomodate for future changes in follow-up work.

To sum up, we have started work on a very flexible documentation framework for Envision, thus far showcasing a number of applications for presenting documentation right next to the code. With this project, we have laid the ground for a novel way of having documentation in a graphical environment. We implemented several types of documentation, clearing the way for

many more exciting features to come. We are eagerly looking forward to new features that will be built on top of this framework.

# Bibliography

- [1] *Website of Billion Dollar Graphics*. URL: <http://www.billiondollargraphics.com>.
- [2] *The Power of Visual Communication*. URL: <http://www.billiondollargraphics.com/infographics.html>.
- [3] *Project website of Envision*. URL: <http://www.pm.inf.ethz.ch/research/envision>.
- [4] Dimitar Asenov. “Design and implementation of *Envision* - a visual programming system”. MA thesis. ETH Zürich, 2011.
- [5] *Qt’s Graphics View Framework*. URL: <http://qt-project.org/doc/qt-4.8/graphicsview.html>.
- [6] *List of libtorrent’s features*. URL: <http://www.rasterbar.com/products/libtorrent/features.html#merkle-hash-tree-torrents>.
- [7] *Android’s Property Animation documentation*. URL: <http://developer.android.com/guide/topics/graphics/prop-animation.html>.
- [8] *PyNSource: a code scanner and UML modelling tool*. URL: [http://www.atug.com/andypatterns/pynsource\\_ascii\\_uml.htm](http://www.atug.com/andypatterns/pynsource_ascii_uml.htm).
- [9] *Doxygen: Generate documentation from source code*. URL: <http://www.stack.nl/~dimitri/doxygen/>.
- [10] *Wikipedia about Wikis*. URL: <http://en.wikipedia.org/wiki/Wiki>.
- [11] *The free encyclopdia Wikipedia*. URL: <http://www.wikipedia.org>.
- [12] *Lighttable: A new interactive IDE*. URL: <http://www.lighttable.com/>.
- [13] Fernando Pérez and Brian E. Granger. “IPython: a System for Interactive Scientific Computing”. In: *Comput. Sci. Eng.* 9.3 (May 2007), pp. 21–29. URL: <http://ipython.org>.
- [14] G.W. French, J.R. Kennaway, and A.M. Day. “Programs as visual, interactive documents”. In: *Software: Practice and Experience* (2013). ISSN: 1097-024X. DOI: [10.1002/spe.2182](https://doi.org/10.1002/spe.2182). URL: <http://dx.doi.org/10.1002/spe.2182>.
- [15] Andrew J. Ko and Brad A. Myers. “Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors”. In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. CHI ’06. Montréal, Québec, Canada: ACM, 2006, pp. 387–396. ISBN: 1-59593-372-7. DOI: <http://doi.acm.org/10.1145/1124772.1124831>. URL: <http://doi.acm.org/10.1145/1124772.1124831>.
- [16] *TinyMCE: A javascript-based WYSIWYG text editor*. URL: <http://www.tinymce.com/>.
- [17] *Markdown Syntax Guide*. URL: <http://daringfireball.net/projects/markdown/syntax>.
- [18] *Qt 4.8 documentation*. URL: <http://qt-project.org/doc/qt-4.8/>.
- [19] *The WebKit Open Source Project*. URL: <http://www.webkit.org/>.
- [20] *Qt’s QTextLayout Class Reference*. URL: <http://qt-project.org/doc/qt-4.8/qtextlayout.html>.
- [21] *Qt’s QStaticText Class Reference*. URL: <http://qt-project.org/doc/qt-4.8/qstatictext.html>.
- [22] *Qt QImage’s supported image formats*. URL: <http://qt-project.org/doc/qt-4.8/qimage.html#reading-and-writing-image-files>.

- [23] *Qt's predefined QColor objects*. URL: <http://qt-project.org/doc/qt-4.8/qt.html#GlobalColor-enum>.
- [24] *MIT's Alloy Analyzer*. URL: <http://alloy.mit.edu/alloy/>.