The mbeddr Documentation Language

Markus Voelter

March 18, 2013

Abstract

This document explains how to use the documentation language of mbeddr. It supports writing prose text with sections, figures etc. It also supports embedding program nodes into the prose text. For example, references to other sections or figures are actural (refactoring-safe) references. Using a separate extension language, it is also possible to reference mbeddr code and even to embed mbeddr code as images or as text. mbeddr visualizations can also be rendered in real-time and embedded into the document. Documents can be output to HTML and Latex. The document you are currently reading is is itself written with the documentation language: another extension module can be used to document itself by embedding documentation language code into documentation documents.

Contents

List of Figures

1 Introduction

When writing prose documents that integrate with code, it is tough to actually create this integration between the prose text and the code. You can either put the prose in JavaDoc-like comments, but then it is hard to tell a story. Alternatively, you can write a Word or Latex document, but then the integration with the code artifacts is tough, boiling down essentially to copy and paste and screenshots.

The mbeddr documentation language provdides a better alternative. It supports writing prose inside of MPS, supporting a tight integration between the prose and the code. In this document we explain how this works. Of course, this document is itself written in the documentation language.

As you can see from the document you are reading, the documentation language supports rendering to Latex. It also supports rendering to HTML.

The documentation language supports several different extensions, each supporting the integration with differnet code artifacts. We will explain all of this in this document. The languages that support these features are all named `com.mbeddr.doc.*` , the devkit you may want to include is called `com.mbeddr.documentation` .

2 Writing Regular Documents

2.1 Simple Text

The most fundamental concept is the `document` . It has a name and references a `configuration` , about which we will take some more later.

Inside a document, the basic document structure consists of sections and various kinds of paragraphs. The simplest kind of paragraph is the text paragraph (entered by typing a `p`). It has vertical brackets on both sides to denote its boundary. Below in fig Fig. 1 is an example, that also embeds this same paragraph as an image.

2.2 Formatting Text

Of course, it is possible to format words as *emphasized* as well as as `code` , and more formatting options will show up over time. You can press `Ctrl-Space` in the text paragraph to see which formatting options are available. Formatting options all start with a backslash. Fig. 2 shows the paragraph you're reading here as a screenshot so you can see the way to format words.

```
section 1.1 normalDocs.simpleText: Simple Text {

  [ The most fundamental concept is the \code(document). It has a name and
  references a \code(configuration), about which we will take some more later. ]

  [ Inside a document, the basic document structure consists of sections
  and various kinds of paragraphs. The simplest kind of paragraph is
  the text paragraph (entered by typing a \code(p)). It has vertical brackets
  on both sides to denote its boundary. Below in fig @fig(exSection) is an
  example, that also embeds this same paragraph as an image. ]

  embed doc section simpleText as exSection
    location: imgTemp
    scaling:  width is 100 % of page
  [ An example section from a document. It is embedded as an image. ]

}
```

Figure 1: An example section from a document. It is embedded as an image.

```
section 1.2 normalDocs.formatting: Formatting Text {

  [ Of course, it is possible to format words as \emph(emphasized) as well as as
  \code(code), and more formatting options will show up over time. You can press
  \code(Ctrl-Space) in the text paragraph to see which formatting options are
  available. Formatting options all start with a backslash. @fig(formatting) shows
  the paragraph you're reading here as a screenshot so you can see the way to
  format words. ]

  embed doc section formatting as formatting
    location: imgTemp
    scaling:  width is 100 % of page
  [ This paragraph shows how to format text. ]

}
```

Figure 2: This paragraph shows how to format text.

2.3 Embedding Figures

You can also embed images that are not rendered from within MPS, but are supplied externally. Below, im Fig. 3 , is an example. The code to embed an image is shown in Fig. 4 .

When embedding an image, you have to specify a name (so the image can be referenced from within the document), a path (defined via a path definition in the document configuration), the actual image file (code completion is available in the editor), as well as a size specification (among others, a scaling factor or a specification relative to page size).

The path definitions are made in the document configuration, and include a path that is valid while the document is edited; MPS path variables can be used. When the document is exported (see below), these are mapped to

Figure 3: An ASH 26E glider.

paths relative to the location at which the document is located. Fig. 5 shows the document config for this document. Note that you can also define size specifications there that can be referenced from images within the document (to reuse the size specs).

2.4 Embedding Other Things

Other artifacts can also be embedded, not just images. The approach is always the same, in particular, you typically specify a path and a size, as well as a name so it can be referenced. The embeddings of the document sources (as screenshots) are examples. In many cases, the artifacts are actually only created during the creation of the document. For example, the screenshots that represent the document source code are created *from the live code* during the generation of the document. This way, they are always up to date. Other extensions to the basic documentation language can contribute their own embedded resources. We will see examples below.

2.5 Exporting Documents

Exporting the document (as HTML, PDF, or possibly in other formats) involves two steps. First, you likely wrote the overall text in several actual documents. To create a big, contiguous HTML or Latex document you probably want to join them. You can do this by creating another document and including others. Fig. 6 shows an example how to do that. Note that you can only include documents for which you specify a dependency in the document header.

The second ingredient is the actual export configuration, as shown in Fig. 7 . There, you specify a document title, optionally an abstract, a root document, as well as a renderer. You also specify path mappings: the path definitions from the configuration (Fig. 5) now have to be mapped to paths relative to the output folder of the generated document (most likely you have to manually create a script that copies these resources into this directory).

```
section 1.3 normalDocs.figures: Embedding Figures {

  [ You can also embed images that are not rendered from within MPS, but are supplied
  externally. Below, im @fig(glider), is an example.
  The code to embed an image is shown in @fig(figures). ]

  image glider from images:/ash26.png
    scaling: width is 100 % of page
    center: false
    has border: false
  [ An ASH 26E glider. ]

  embed doc section figures as figures
    location: imgTemp
    scaling:  width is 100 % of page
  [ The code that embeds an image ]

  [ When embedding an image, you have to specify a name (so the image can be referenced
  from within the document), a path (defined via a path definition in the document
  configuration), the actual image file (code completion is available in the editor),
  as well as a size specification (among others, a scaling factor or a specification
  relative to page size). ]

  [ The path definitions are made in the document configuration, and include a path
  that is valid while the document is edited; MPS path variables can be used. When
  the document is exported (see below), these are mapped to paths relative to the
  location at which the document is located. @fig(config) shows the document config
  for this document. Note that you can also define size specifications there that
  can be referenced from images within the document (to reuse the size specs). ]

  embed doc section Config as config
    location: imgTemp
    scaling:  width is 100 % of page
  [ An example document configuration with path definitions. ]

}
```

Figure 4: The code that embeds an image

To create the document, you simply generate the respective MPS model. The HTML or Latex file(s) will be generated.

- For Latex, you specify a document class as well as a prolog file. The prolog file is included at the beginning of the document, and it can define all the style customizations you want.
- For HTML, you specify a style sheet. This style sheet can format the HTML code in any way you want. Take a look at the generated HTML to learn about the style classes used in the generated HTML.

```
document configuration Config
  default temp path <no defaultTempPath>
  additional paths
    imgTemp ->
        ${mbeddr.github.core.home}/code/languages/com.mbeddr.doc/solutions/com.mbeddr.doc.doc/temp
    images ->
        ${mbeddr.github.core.home}/code/languages/com.mbeddr.doc/solutions/com.mbeddr.doc.doc/images
  size specifications
    smallCodeShot: scale by 55 %
  shortcuts
```

Figure 5: An example document configuration with path definitions.

```
authors:
document Root config Config depends on A_Introduction    {
                                       B_SimpleDocuments
                                       C_EmbeddingMbeddr
                                       D_Extending


  include A_Introduction
  include B_SimpleDocuments
  include C_EmbeddingMbeddr
  include D_Extending

}
```

Figure 6: A root document that includes other documents in a specific order.

2.6 Embedding Documents

The documentation language is extensible. It can embed all kinds of other things. In the previous section Section 2 we have already implictly seen how to embed screenshots of documentation artifacts. This is probably a bit weird and meta, but it is useful for documenting the documentation language. It also shows off the flexibility of the approach itself.

2.7 Tables

The documentation language supports tables. The table below shows an example. This is an `inline table` , there are also `floating table` , that can be referenced with the `@fig` reference. The code for tables is shown in Fig. 8 .

Name	Alter	Adresse
Markus	38	voelter@acm.org
Bernd	30	kolb@itemis.de
Peter	30	peter@friese.de

In a table, you specify the number of columns (and a name for floating tables). You then add rows and cells. Currently we support only text cells (denoted

```
document export DocumentationDocumentation {
  title: The mbeddr Documentation Language
  abstract: [ This document explains how to use the documentation language of mbeddr.
              It supports writing prose text with sections, figures etc. It also supports
              embedding program nodes into the prose text. For example, references to other
              sections or figures are actural (refactoring-safe) references. Using a separate
              extension language, it is also possible to reference mbeddr code and even to
              embed mbeddr code as images or as text. mbeddr visualizations can also be
              rendered in real-time and embedded into the document. Documents can be output
              to HTML and Latex. The document you are currently reading is is itself written
              with the documentation language: another extension module can be used to
              document itself by embedding documentation language code into documentation
              documents. ]
  root doc: Root
  renderer: Latex documentClass article
                  prologFile mbeddr-prolog.ltx
                  table of contents true
                  list of figures true
    inactive renderer: HTML(stylesheet = htmlexport.css)
  path mappings
    imgTemp -> figures/doc
    images -> figures/doc
}
```

Figure 7: The export configuration for this document.

by the parens) and text block cells (denoted by the angle bracket, just as in text paragaphs in general). Additional cell types will be supported in the future.

For each cell, you set if it is a header using a setting in the inspector. For each row, you can specify whether there should be lines above or below the line. Currently, there is *always* a line between the columns; this may be changed in the future.

3 Embedding mbeddr Artifacts

A more interesting use case is the ability to work with mbeddr code. In fact, the ability to tightly integrate with mebddr code was the reason for building this documentation language, as we have said in Section 1 . In this section we explain how it works.

3.1 Referencing Code

The simplest way of integrating documentation prose is to use references to mbeddr code. Why would you do this? Of course to be refactoring-safe: as you rename the referenced element, the text in the documentation changes with it. If you delete the element, the reference breaks, and you know you have to change something.

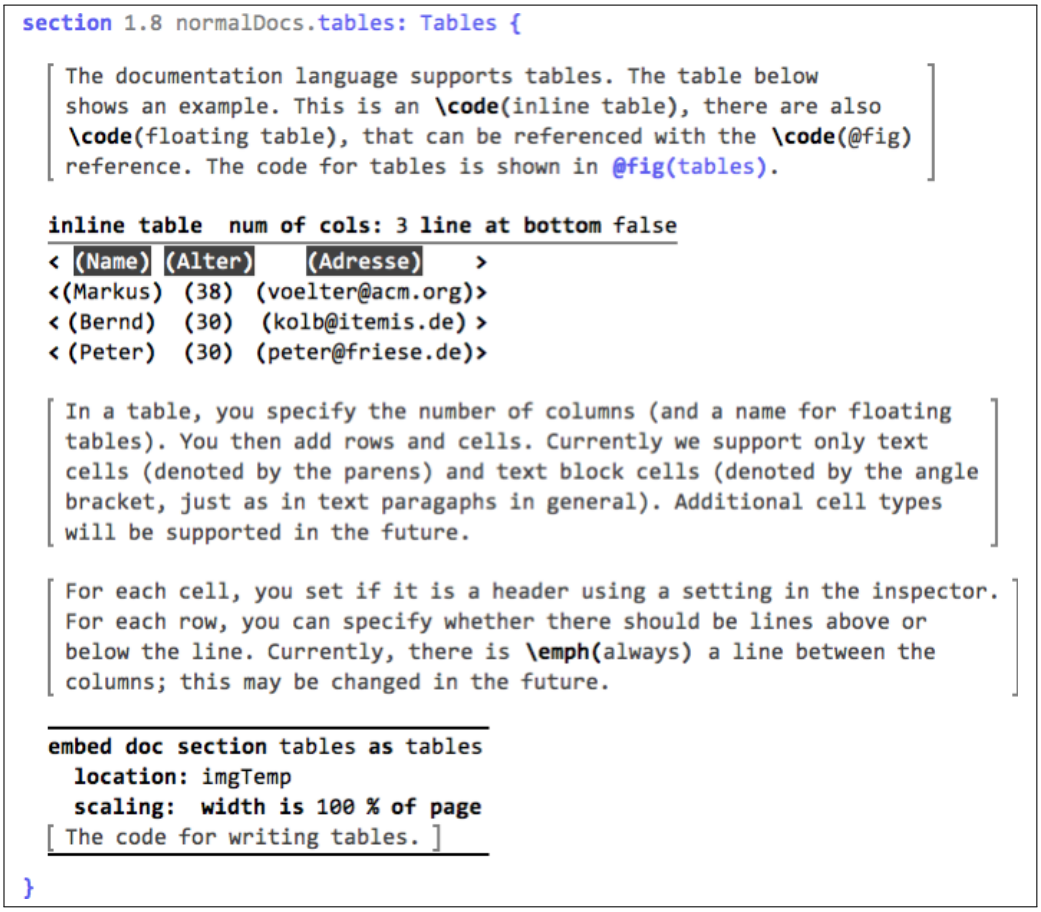

Figure 8: The code for writing tables.

For example, you reference the interface `Calculator` using the `@cc` embedded node. You can also refer to any named child of a top level content by selecting that child after the slash in the `@cc` element. For example, you can refer to an argument `x` . If you want to reference things that do not have a name, you can attach a name label to an element (using the `Attach Name` intention; you need to use the `com.mbeddr.doc.c` language in the respective mbeddr model to get the intention. For example, we can refer to a precondition . Fig. 9 shows how this looks in the code.

In addition, you can also refer to modules using the `@cm` node. For example, here we refer to the `ExampleCode` module.

Fig. 10 shows the source for the referencing examples.

```
exported cs interface Calculator {
  int8 add(int8 x, int8 y)
    post(0) result == x + y
  int8 divide(int8 x, int8 y)
    pre(0) y != 0 // ^aPreCondition
    post(1) result == x / y
}
```

Figure 9: An interface with a marker on a precondition, so it can be referenced.

3.2 Embedding Code

Embed as Image You have already seen in the previous paragraph how to embed mbeddr code as an image into the document. In that example, Fig. 9 embedded a complete top level construct, an interface in this case. But what if you wanted to embed only a smaller section, such as a state in a state machine or a single operation in an interface? Fig. 11 shows an example of embedding only an operation. The code to do that is shown in Fig. 12 ; essentially you mention the `add` operation after the slash in the `embed image` tag.

Embedding as Text You can also embed mbeddr code as text. This is interesting in particular for Latex export, since you can configure the `listings` package to provide syntax highlighting for your code. The following paragraph shows how to embed the interface as text; not that this is not a floating entity and cannot be referenced, it is inlined with the text. Also note that in the inspector for the `embed as text` tag you can specify the language name used for highlighting. By default, it is `mbeddr` .

```
1 exported cs interface Calculator {
2   int8 add(int8 x, int8 y)
3     post(0) result == x + y
4   int8 divide(int8 x, int8 y)
5     pre(0) y != 0 // ^aPreCondition
6     post(1) result == x / y
7 }
8
9
```

3.3 Embedding Visualizations

Some elements in mbeddr implement the `IVisualizable` interface, so they can provide one or more visualizations. You can see these visualizations by selecting the `Visualize` menu item from the context menu. Alternatively you can also embed such visualizations into a generated document; the visu-

```
section 1.2 workingWithMbeddr.referencingCode: Referencing Code {

  [ The simplest way of integrating documentation prose is to use references
  to mbeddr code. Why would you do this? Of course to be refactoring-safe: as
  you rename the referenced element, the text in the documentation changes with it.
  If you delete the element, the reference breaks, and you know you have to change
  something. ]

  [ For example, you reference the interface @cc(Calculator/) using the \code(@cc)
  embedded node. You can also refer to any named child of a top level content by
  selecting that child after the slash in the \code(@cc) element. For example, you
  can refer to an argument @cc(Calculator/x). If you want to reference things that
  do not have a name, you can attach a name label to an element (using the
  \code(Attach  Name) intention; you need to use the \code(com.mbeddr.doc.c) language
  in the respective mbeddr model to get the intention. For example, we can refer to
  @cc(Calculator/aPreCondition). @fig(calculator) shows how this looks in the code. ]

  embed image ExampleCode.Calculator/ as calculator
    location: imgTemp:/
    scaling:  smallCodeShot
  [ An interface with a marker on a precondition, so it can be referenced.  ]

  [ In addition, you can also refer to modules using the \code(@cm) node. For example,
  here we refer to the @cm(ExampleCode) module. ]

  [ @fig(refCode) shows the source for the referencing examples. ]

  embed doc section referencingCode as refCode
    location: imgTemp
    scaling:  width is 100 % of page
  [ Example code for referencing C code from documents.  ]
}
```

Figure 10: Example code for referencing C code from documents.

alization is rendered on the fly (like the code screenshots discussed in Section 3.2 .

An example for such a visualization is shown in Fig. 14 . As with other images, you have to specify the size/scaling, and the location of the temporary files. Obviously, you have to reference the visualizable element, and you also have to select which of its visualizations you want to render. You can select them via code completion after the slash in the `visualize` element.

Note that (at least as of now) you have to manually render the images with `plantuml` . The following listing shows how we render the images using `plantuml` and how we copy them into a an `images` folder. This folder is the one from which the images are read by the generated Latex file.

```
1 echo ===== Rendering Visualizations using plantuml
2 cd temp
3 java -jar ../plantuml.jar *.puml
4 cd ..
5
6 echo ===== Copying Rendered Images and Screenshots
```

```
int8 add(int8 x, int8 y)
  post(0) result == x + y
```

Figure 11: An example of how to embed only a part of a module content as an image.

```
7  cd source_gen/main
8  mkdir doc_images
9  cd ../..
10 cp temp/*.png source_gen/main/doc_images
11
```

The above example also shows how to embed a listing as text. You can add a `listing` paragraph and paste the actual textual code into a text area in the inspector.

4 Extending the Documentation Language

Just as any other mbeddr language, the documentation language is extensible. There are two main extension points: enw kinds of paragraphs and new embedded nodes.

4.1 New Paragraphs

To create new paragraphs, you should extend the `AbstractParagraph` concept from the `com.mbeddr.doc` language. For example, the regular text paragraphs as well as the sections and images are subconcepts of `AbstractParagraph` .

4.2 New Embeddable Nodes

Concepts that should be embeddable in "regular" text paragraphs (such as the one your are reading right now) must implement the `IWord` concept interface. This way they can be embedded in any text paragraph. Of course, this is not what you might want; if you want to restrict their usability to within actual `Document` , you have to write a `can be child` constraint, or, alternatively, extend the `DocumentWord` abstract concept.

In addition to extending the respective interface or concept, embeddable concepts must also define a `transformationKey` property. It is the text that is used to instantite the node from the code completion menu.

As an example, take a look at the following paragraph. It uses an extension that can be used for embedding variables and equations.

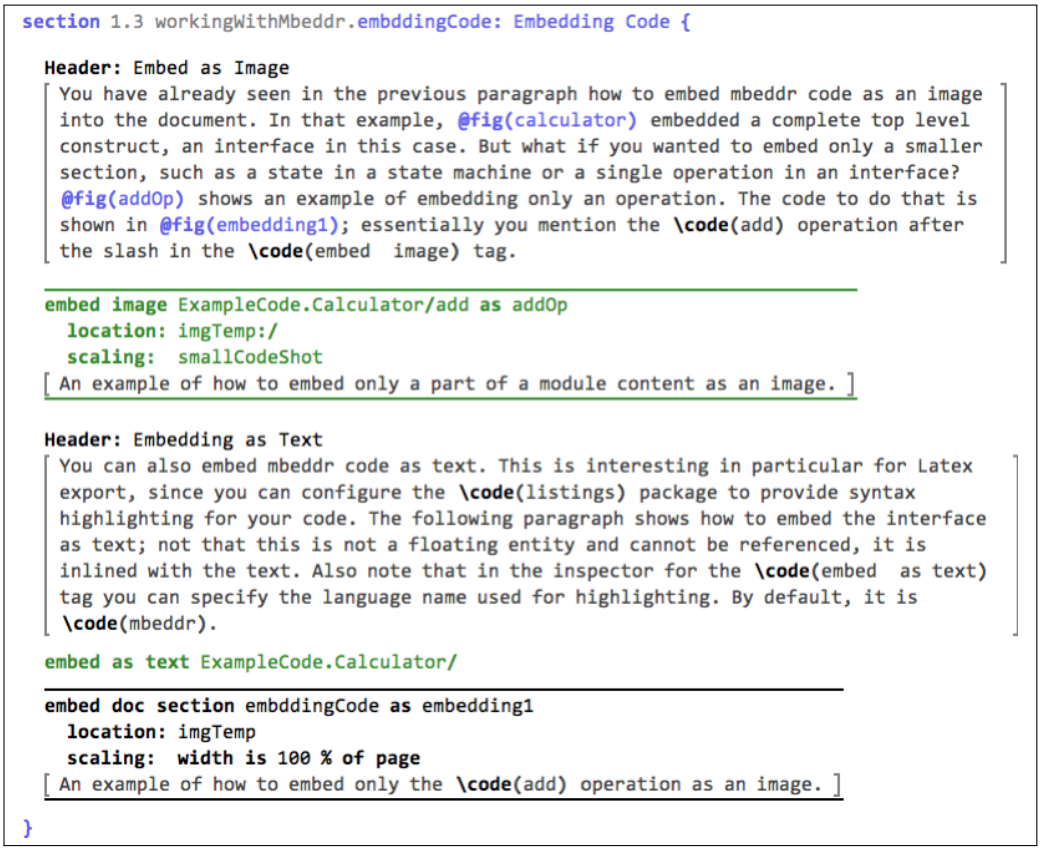

Figure 12: An example of how to embed only the `add` operation as an image.

The Drake Equation The Drake equation calculates the number of civilizations N in the galaxy. As input, it uses the average rate of star formation SF , the fractios of those stars that have planets fp and the average number of planets that can potentially support life ne . The number of civilizations can be calculated as $N = SF * fp * ne$

Note that the variables are typed, the equations are type checked and you can directly use the variables and equations from mbeddr code if you want to. To learn how this works, take a look at the `com.mbeddr.doc.expressions` language.

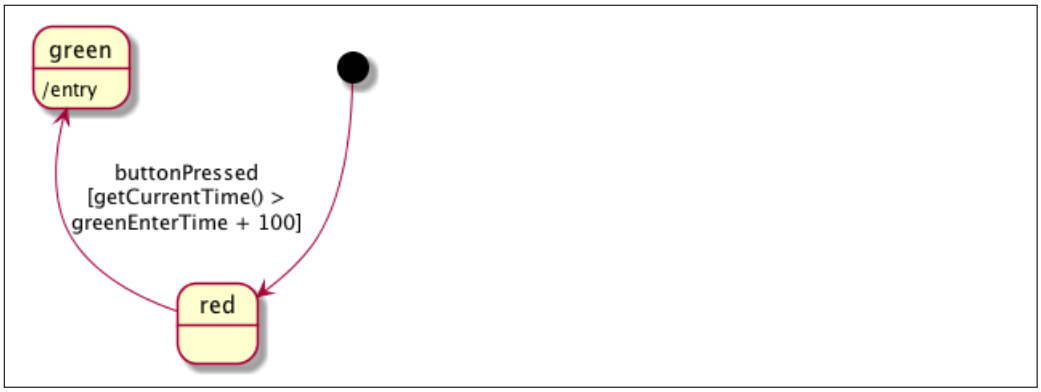

Figure 13: The `TrafficLights` state machine as a graph.

```
section 1.4 workingWithMbeddr.visualizations: Embedding Visualizations {

  [ Some elements in mbeddr implement the \code(IVisualizable) interface, so they can
  provide one or more visualizations. You can see these visualizations by selecting
  the \code(Visualize) menu item from the context menu. Alternatively you can also
  embed such visualizations into a generated document; the visualization is rendered
  on the fly (like the code screenshots discussed in @sect(embddingCode). ]

  [ An example for such a visualization is shown in @fig(vis). As with other images,
  you have to specify the size/scaling, and the location of the temporary files.
  Obviously, you have to reference the visualizable element, and you also have to
  select which of its visualizations you want to render. You can select them via
  code completion after the slash in the \code(visualize) element. ]

  visualize ExampleCode.TrafficLights/statechart (2D) as tl
    location: imgTemp:/
    scaling:  scale by 60 %
  [ The @cc(TrafficLights/) state machine as a graph. ]

  embed doc section visualizations as vis
    location: imgTemp
    scaling:  width is 100 % of page
  [ Example document code to embed a visualization. ]

  [ Note that (at least as of now) you have to manually render the images with
  \code(plantuml). The following listing shows how we render the images using
  \code(plantuml) and how we copy them into a an \code(images) folder. This
  folder is the one from which the images are read by the generated Latex file. ]

  listing (pasted) language = bash

  [ The above example also shows how to embed a listing as text. You can add a
  \code(listing) paragraph and paste the actual textual code into a text area
  in the inspector. ]
}
```

Figure 14: Example document code to embed a visualization.