

Bridging Javadoc and design documentation via UML diagram image maps

Asko Soukka, Tuukka Hastrup, Tuomas J. Lukka and Benja Fallenstein
Hyperstructure Group
Agora Center, University of Jyväskylä
P.O. Box 35, FIN 40014

humppake@iki.fi, Tuukka@iki.fi, lukka@iki.fi, b.fallenstein@gmx.de

ABSTRACT

We present a navigational aid for documentation used in software development. Based on using readily-authored UML diagrams as multi-ended links, we hypertextually connect two distinct areas of documentation: design documents and javadoc program code documentation. Connecting the distinct areas is essential because it could help new developers in getting productive and supports development processes where both design and implementation change continuously.

We also describe the lightweight implementation as a supplement to a Free Software toolchain. To achieve bi-directional linking, the implementation after-treats HTML pages generated by other tools, injecting into referred pages links back to the referring imagemap of each UML diagram. Each diagram also serves as a spatial context for navigation within target nodes.

The software is currently a part of and in use at our software project. We mention how the scopes of the navigation system and software could be widened to hypertext document types outside the needs of our software project.

Categories and Subject Descriptors

H.5.4 [Information Interfaces and Presentation]: Hypertext/Hypermedia—*Navigation*

; D.2.2 [Software Engineering]: Design Tools and Techniques—*Evolutionary prototyping*

General Terms

Design, Documentation, Human Factors

Keywords

imagemap, maps, bi-directional linking, link authoring, UML, spatial hypertext

1. INTRODUCTION

Software projects manage a large base of evolving documentation, which is inter-related on many levels. Design documentation gives architectural views on a more general level, while the program code source files contain minute interface specifications and often also embedded documentation giving details on how the interfaces should be used. Although these two parts of documentation are semantically dependent, linking them by hand is tedious and error-prone when they are authored using different software.

In this article, we present a navigational aid for software documentation. Based on using UML diagrams as multi-ended links, our tool connects two distinct areas of software engineering documentation: the general design documents and the detailed method-level javadoc documentation.

In the following sections, we first discuss the role of documentation in software engineering in general, then bring the problem of the separate documentation bodies into focus and present our solution to linking and its implementation. Finally, we discuss our experience from using the tool and conclude.

2. BACKGROUND

2.1 Software engineering process

In their article “A rational design process: how and why to fake it” [?] Parnas and Clements argue that rational software design process is not generally possible, but acceptable results could be achieved by faking “the ideal process”. Their ideal software design process contains the following steps [?, pp. 252-255]:

- establish and document requirements
- design and document the module structure
- design and document the module interfaces
- design and document the use hierarchy
- design and document the module internal structures
- write programs
- maintain (redesign and redevelopment, keeping documentation up to date)

It should be clear that documentation plays a major role in the ideal software design process, and at different levels of abstraction the type of documentation varies. The steps listed above should produce documentation which records requirements and design decisions, and could be referenced throughout the building of the software [?].

A programmer joining a project may need a lot of mentoring by other group members before being able to contribute productively to the project. This results in the Mythical Man Month effect [?, p. 16]: adding a new member into the software design process delays rather than speeds up the project.

“Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication.” [?, p. 18]

“The added burden of communication is made up of two parts, training and intercommunication. Each worker must be trained in the technology, the goal of the effort, the overall strategy, and the plan of work. — Intercommunication is worse.” [?, p. 18]

Obviously, the greater the amount of programmers or the higher the turnover rate, the more important it is to have good documentation. When new programmers join the project they shouldn’t have to depend completely on the old staff for finding their way around. An up to date and rational set of documents available for them could ameliorate the Mythical Man Month effect [?, p. 255]. Improving the learning curve of new programmers may have significant effects.

In our own development effort, Gzz, different parts of the system have different needs for the development process: the core parts of the system are frozen and require a more formal process for changes, as in Boehm’s Spiral Model of software development and enhancement [?]. On the outer edges of the system, new research is taking place and should not be hampered by requiring formal process or documentation beyond the immediate needs of the group members involved in the development of that particular part. For this reason we have adopted some conventions from Extreme Programming, such as continuing analysis of design objectives along the implementation progress [?]. This calls for cross-linkage between design and program code documentation, as they both evolve continuously.

2.2 UML diagrams

The Unified Modeling Language (UML) is the standard way to visually describe software architectures and constructs in diagrams [?]. It was originally developed for descriptions on an abstract level (many constructs of it cannot be directly expressed in any programming language) [?, p. 12], but the current trend in research is to use it also on the concrete level, as to fully unify the architectural documentation and program code: the program code might be generated from highly detailed diagrams [?], or diagrams may be produced directly from the source code [?].

In this article we focus on the more conventional use of UML to plan and document software architecture on a general level. UML can function as a common language for communication within a

Table 1: Comparing design documentation to Javadoc.

Design documentation	Javadoc
good overall picture	easy to find a given class, easy to check all methods
little detail	detailed
may be slightly outdated at any particular time	methods and classes <i>always</i> up to date (generated from source), doc comments also usually
hard to find explanations for a particular class	no overall picture of classes’ roles
UML diagrams	—
written and organized by humans	written by humans, organized rigidly by package structure

software development team, and for this purpose we prefer human-drawn (non-autogenerated) diagrams that show the semantically meaningful features at the right level of abstraction:

“You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system.” [?, p. 24]

3. THE PROBLEM

In our software engineering project [?], as probably in most projects that use the Java programming language [?], the software development documentation is divided into two major domains: the design documentation and Javadoc [?]. The design documents cover the most important architectural features and are written either before coding (for design) or after (for exposition of the architecture or refactoring design). Javadoc, on the other hand, is a detailed and fully generated documentation from javadoc comments of each class and method in the Java source code. The two types of documentation are complementary, as demonstrated in Table 1:

During coding, the Javadoc documentation is often necessary, but the design documentation is easily left in the dark reaches of the filesystem. It could be argued that the reason for design documents being left unused is that parts related to ongoing work are hard to find.

For example, when referring to the Javadoc on how to use some interface, one often would like to know how to obtain an object which implements said interface, but unless someone has explicitly written the instructions into the doc comments, the Javadoc will only explain how the interface is used. What exasperates this situation is the certainty that in the design document there surely is a diagram and a section which talks about the issue, but finding them will take time.

The distinct pieces (Javadoc and the design documentation) cannot be seen as a whole. The obvious question, then, is: can we increase the overall value by hyperlinking the two distinct pieces of documentation?

When looking at a design document, jumping to the Javadocs to get the details would be useful, and when looking at a Javadoc, it would

Table 2: Linking possibilities from items in different kinds of UML diagrams to Javadoc.

Conceptual	Specification	Implementation
probably no links to javadoc	can link to Java interface and some classes	can link to all classes
the design documentation packages can be linked to the architectural documents discussing those packages	can link to Java packages	can link to Java packages

be most useful to be able to see if any design documents discuss that class or package. We believe that the design documentation would be read more if made easily reachable from relevant parts of Javadoc.

This is the starting point for the Free Software toolchain we developed: a toolchain for bidirectional linking between design documentation and Javadoc, using UML diagrams as multi-ended links.

4. THE SOLUTION

4.1 Usability considerations

The usability of hypertext-based documentation may suffer from user's disorientation: the tendency to lose one's sense of location and direction in a nonlinear document [?, pp. 38-40]. This means that users don't know where they are in the documentation network or how to get to some other place that they know to exist in the network.

Edwards and Hardman [?] argue that the most appropriate types of navigation devices would be based on spatiality. According to their research, individuals appear to be attempting to create cognitive representations of hypertext structures in the form of a survey-type map. They conclude that users should be allowed to develop a cognitive map of one view of the data structure before being given the opinion of navigating through the data some other way [?, p. 123].

In our case, UML diagrams are the obvious candidate for a common navigational metaphor to unify the two distinct pieces. The UML diagrams include graphical objects representing Java classes, UML diagrams not only show the readers a map of documentation but also perform as spatial navigation menus.

4.2 What could be linked

To see which possibilities we have when linking from diagrams, we use Cook and Daniels's classification of UML diagrams into three classes [?]: conceptual diagrams, specification diagrams and implementation diagrams. Looking into the items available in each of these classes, we come to the conclusion shown in Table 2.

The UML diagrams enable navigation in various purposes. Applying Trigg's taxonomy of different link types [?], we could identify several semantic meanings of the links, shown in Fig. 1. Links from the diagram to documentation pages are expressed in HTML

imagemaps, and bi-directionality is ensured by injection of HTML IMG tags to all referred pages. As web browsers generally embed the IMG-linked diagram to the documentation nodes, the diagram creates spatial links between all the referred pages. Further, each documentation node creates spatial links between all the diagrams it refers to, allowing the reader to see which alternate spatial views are available for the current node.

4.3 Reader's point of view

Before reaching its current state, our documentation evolved through several distinct steps, which will be viewed first from the reader's and then from the developer's point of view.

Step 0; The beginning. In the beginning we had a distinct design documentation with UML diagrams and Javadoc generated from the sourcecode — probably the most common case. Both could be comprehensive and well navigable by their own, their information is difficult for the reader to combine because there is no crosslinking between them.

Step 1; UML to Javadoc links. Now there are imagemap links from the UML diagrams in the design documentation to the relevant Javadoc pages. However, after moving to Javadoc the context in design documentation is lost, and there are no links from Javadoc to design documentation.

Step 2; Relevant UML diagrams embedded in Javadoc. The UML diagrams are now embedded into the Javadoc pages they refer to, and they function as imagemaps also on the Javadoc pages. The diagrams function as menus between the objects that appear in diagrams. While this step provides more context for the actual classes, the design documentation relevant to a given javadoc page is still unreachable.

Step 3; From Javadoc through a UML diagram to a design document. Even though the design documents were not included in the original UML diagrams, they are included in the final image on top of the diagram 7. Including **all** the contexts where the diagram appears as links in the graphical image creates a consistent whole - a spatial focus+context menu. The element that represents the current hypertext node is emphasized (colored and also circled) for clarity.

Because the original location on the diagrams is easily reachable from its every implicit occurrence in Javadoc or in design documentation, the size of the implicitly embedded diagram could be reduced by 50%. The name of diagram object is also shown, hovering the pointing device over diagram - also on diminished diagrams.

4.4 Authors's point of view

On the developer side, creating diagrams within the design documentation should be as easy and natural as possible; we want to minimum the barrier of drawing even only a small diagram to make documentation more clear.

For Step 1, we keep it best to demand explicit links in the UML diagrams. Author has to decide for each object in UML diagram whether it should be linked, for example, by giving a fully-qualified java class name or a relative path to design documentation page

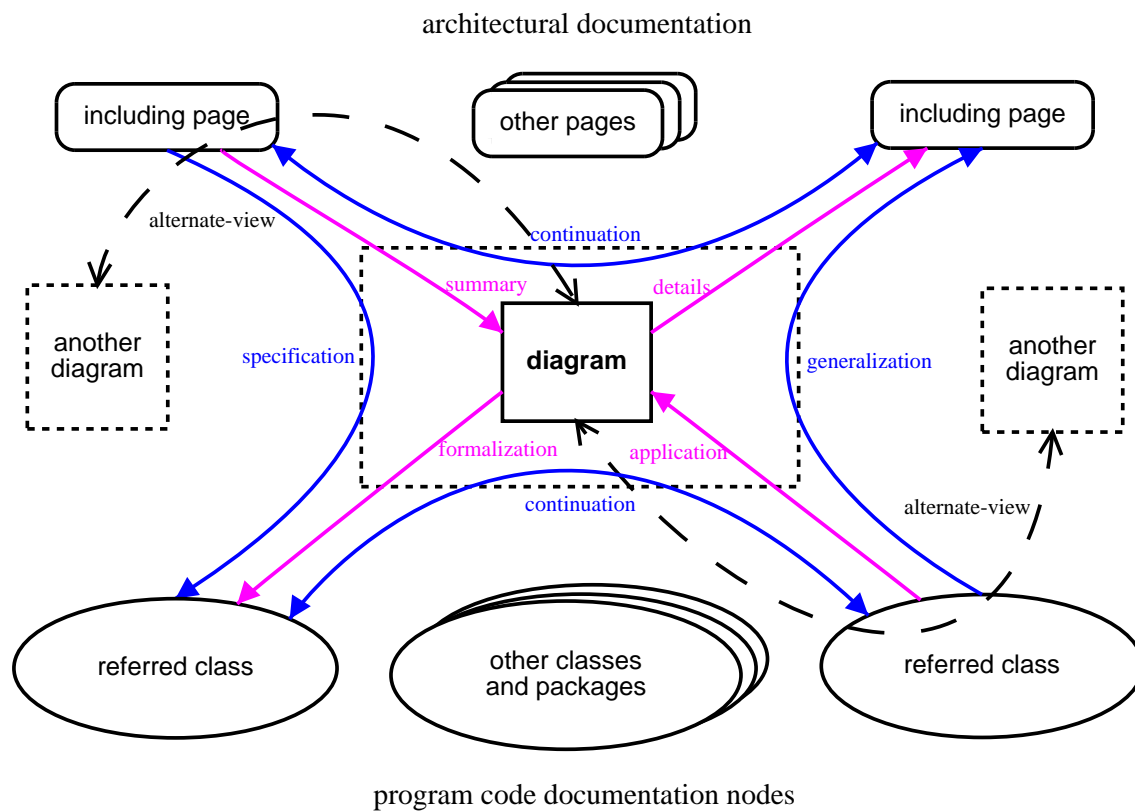


Figure 1: Semantic meanings of traversing the bi-directional links of a diagram. The dashed curves mean linkage by spatial relation.

2. After Step 1 from author's perspective, the computer has *all* the information needed to complete also steps 2 and 3, so *no further changes should be required*. For example javadoc comments should not need changes at all.

During steps 2 and 3 computer simply embeds diagrams into previously linked documentation pages and creates backlinks to the initiating documentation pages.

5. THE IMPLEMENTATION

Eventually, we found no Free Software tools which would fulfill our needs. Therefore, we decided to proceed with our own implementation. In addition to the previously discussed goals of the documentation as whole, there were also several implementation related goals to be achieved. The tool had to:

- be tidy and light-weight
- be built on existing Free Software tools and be a part of Free Software toolchain
- support easy re-editing (from the source)
- provide a plugin interface for different documentation tools

Up to this article, all steps except the last one are implemented. The existing tools we selected as the basis for our documentation tool are: Javadoc, Docutils [?], and our own UML diagram description tool. Further, the UML tool uses several free utilities to

convert each lexical UML diagram description into final *Portable Network Graphics* (PNG) diagram files. Such utilities are *MetaPost* [?] (mpost), which implements a language for picture drawing, and *Netpbm* image file manipulation toolkit. Besides Javadoc, all tools used are Free Software. The current linking tool is implemented to support Javadoc, but after the plugin interface is ready, switching Javadoc to any free alternative or using them as parallel should be possible with only minor plugin programming.

The javadoc format is the standard way to include documentation in Java source code [?]. Generating the WWW pages is not a complicated process, and at the moment GNU Classpath Tools [?] is developing a Free Software implementation called gjdoc. Also many other Javadoc like Free Software tools for exists - many of them supporting multiple programming languages including Java.

5.1 UML tool

Our UML diagram description tool was already implemented before the documentation linking tool, which finally enabled the bi-directional linking between distinct documents. In the following we discuss shortly, how we ended up to use this our own lexical UML language instead of using already existing free direct manipulation diagram drawing tools like Dia [?], or CASE-tool like ArgoUML [?].

There exist also such Javadoc like tools, which generate some embedded diagrams into documentation. Doxygen [?], for example, generates diagrams of class inheritance tree. Also proprietary Ra-

tional Rose could be used to reverse engineer UML diagrams and build up to date documentation (with support of Rational Soda) from source code [?]. Of course, generated documentation may give well detailed information from the current implementation, but the design documentation should also cover the future and be rather well abstracted than well detailed. Therefore, we want to avoid being bloated by a large amount of too detailed diagrams (meaningless for us) and prefer fully human created diagrams in our design documentation.

As we want to remove the barrier of drawing at least small diagrams within the documentation to make it more understandable, the drawing method should be easily integrated into current working customs. Because in small software development group programmers also write the documentation, the documentation tools shouldn't be gap to change the programming tool to document writing tools. Even better would be that the programming tool could be used as documentation tool. Lexical UML diagram description for our UML drawing tool, of course, can be written with any text editor. At least for a programmer, who are used to describe objects lexically, describing also the UML diagrams lexically could be even more efficient than using distinct direct manipulation drawing tool.

“It's as if we have thrown away a million years of evolution, lost our facility with expressive language, and been reduced to pointing at objects in the immediate environment.— We have lost all the power of language, and can no longer talk about objects that are not immediately visible.” [?, p. 74]

Direct manipulation user interface do not necessarily improve performance: users must learn the meaning of the graphical components, graphic presentation could be misleading and graphical presentation could take excessive screen display space [?, p. 64]. Direct manipulation interfaces could be also rather slow to use, since in a such interface user may have to directly manipulate everything. Instead of an executive who gives high-level instructions, the user is reduced to an assembly line worker who must carry out the same task over and over [?, p. 74]. The UML language has a large amount of different symbols and different connection types between them [?]. It could be quite challenging for a direct manipulation interface to make all these alternatives as easily available as they could be just typed when using lexical description.

In our UML tool the description of UML diagram is divided in two: into a description of all the elements in diagram 2 and description their graphical layout 3. The final diagram 5 is compiled

4 from the element and layout descriptions. The description of existing elements is easily done using our UML tools' lexical description language and the description is readable even without compiling the graphical diagram. The graphical layout is more difficult to express lexically without exceptional spatial imagination. Usually the intended results need several compilation trials. We admit that after all elements for a diagram are selected using lexical description, the graphical placing for them could be done much easier by direct manipulation.

5.2 Docutils

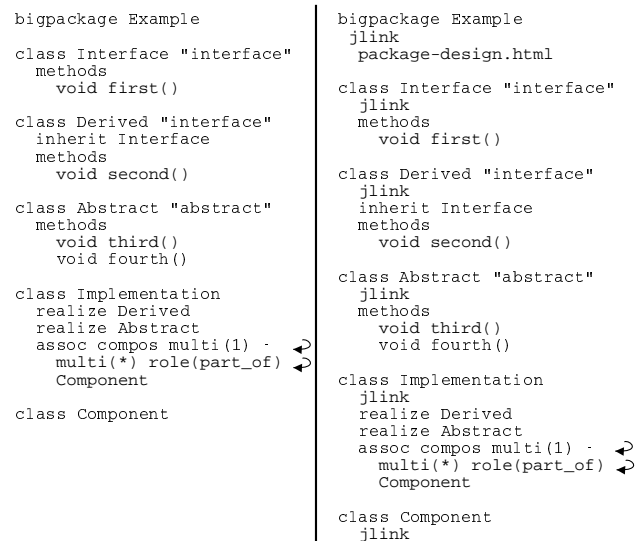


Figure 2: An example of UML tool element description without and with link information for linking tool.

```
# Places "Derived" object into absolute
# coordinates (100, 100). Coordinate (0,0)
# is the lower left corner.
Derived.c = (100, 100)

# Places other objects horizontally or
# vertically relatively to absolutely
# placed "Derived" object.
horizontally(50, interface_h, Interface, Derived, Abstract);
vertically(50, interface_v, Derived, Implementation);

# Places "Component" object into absolute
# coordinates (300, 0).
Component.c = (300, 0);
horizontally(50, component_h, Component);

# Finally the package object is stretched
# around classes.
pad = 30;
Example.nw = Interface.nw + (-pad,pad);
Example.se = Component.se + (pad,-pad);
```

Figure 3: An example of UML tool layout description.

Because Javadoc is generated from special doc comments in the source code, it is almost always up to date. The design documentation, though, is updated manually. Of course, it should be updated regularly during every design cycle, but in practise that won't always happen. To avoid out-of-date design documentation, the threshold of writing the design documentation and explaining the design using UML diagrams should be as low as possible.

A more difficult issue is to select tools for writing the design documentation and drawing diagrams into it. The solution should be cheap, and as a Free Software project we would prefer other Free Software. Also the solution should fit well to our current working customs.

As a natural continuum for UML tool we started to use reStructuredText (reST) plaintext syntax with Docutils parsing system [?] for writing our design documentation. The UML tool already shared some of the main goals of reST syntax [?]: it was readable also raw form, the most common elements had very simple markup, it was writable with any text editor and the UML markup was highly extensible (by enlarging the preprocessor and MetaPost UML macro library 4). ReST syntax itself is extensible easily by inventing new directives and adding parsers for them into Docutils. The extensibility of reST syntax and Docutils parsing system made

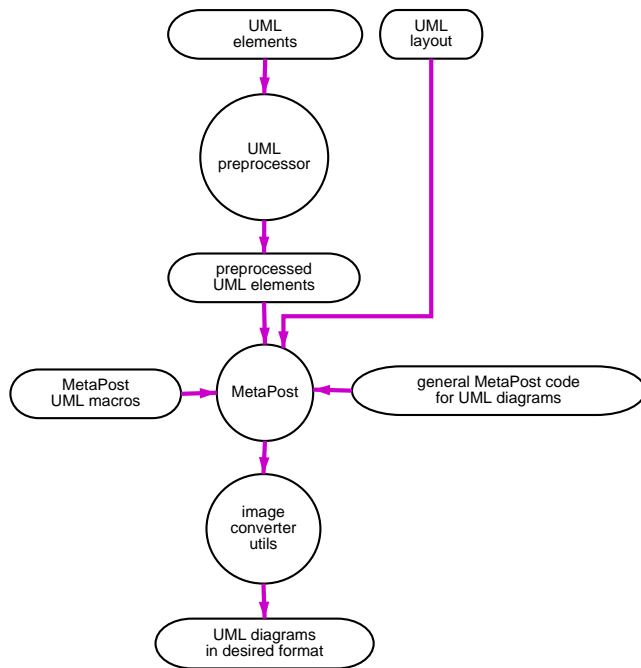


Figure 4: The process of compiling UML diagram using our UML tool.

Note: Because the layout description is passed directly to MetaPost as it is, any additional MetaPost code is allowed to enter within it. This is a small bonus for using MetaPost, since it allows drawing of arbitrary decorations into diagrams.

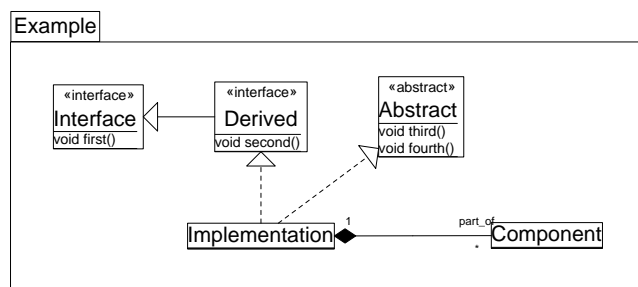


Figure 5: An example of UML tool diagram output.

it possible to write embedded markup of our umltool into design documentation. That way a lexical description of UML diagram is also easily reachable and editable when updating the documentation including the diagram description.

Even reST syntax is somehow WYSIWYG (what-you-see-is-what-you-get). Gentner and Nielsen [?, p. 75] argue that a document has a rich semantic structure that is poorly captured by its appearance on a screen or printed page, which many WYSIWYG tools assumes to be the only one useful representation of the information. WYSIWYG seem to assume that people want always paper-style reports to be read from top to bottom, when we already live within an information flood, where the overloaded reader should be allowed to affect the final representation of document [?, p. 75]. ReST and Docutils parser won't make such assumptions, but they clearly distinguish the semantic structure and rules for converting the text and semantics into printable page. ReST syntax has unambiguous rules to parse document into structural form and after parsing it could be written into multiple output formats [?].

5.3 Linking tool

Using reST syntax for design documentation, lexical description for UML diagrams and extensible Docutils parser for generating the final representation, we were allowed to embed also UML diagram description within the design documentation reST sources. When converting the reST document into final representation, our custom directive for Docutils passed the embedded UML diagram description to our UML tool and added a reference to the compiled diagram into Docutils' document tree structure. This was a promising base to build our linking tool on.

In the current implementation we have two docutils directives to use for embedding UML diagrams into reST based design documentation. The first one is used to describe a new UML diagram, the second is used to refer to an already existing diagram. The correct reference is done simply by diagram name, so it could be spoken about UML diagram namespace, where all diagrams are distinguished by unique naming.

To enable linking elements in UML diagram a "jlink" command after each linkable element is added into lexical UML diagram description 2. If "jlink" has no attributes it is linked to Javadoc page of the package or class determined from the element's name. Otherwise element is linked into file determined by the attribute following "jlink".

The documentation [?] is created in two distinct phases. In the first phase all reST sources are parsed and compiled into HTML and lexical UML diagram descriptions are extracted into temporary storage. Also the paths for reST documents explicitly referring particular diagrams are stored with that diagram source. In the second phase all compiled reST documents are processed again:

- Focused versions of UML diagram with list of all explicitly referring reST documents are created and embedded into compiled documentation pages.
- Image maps targeting all linked elements appearing in UML diagrams are generated and embedded into compiled documentation pages.

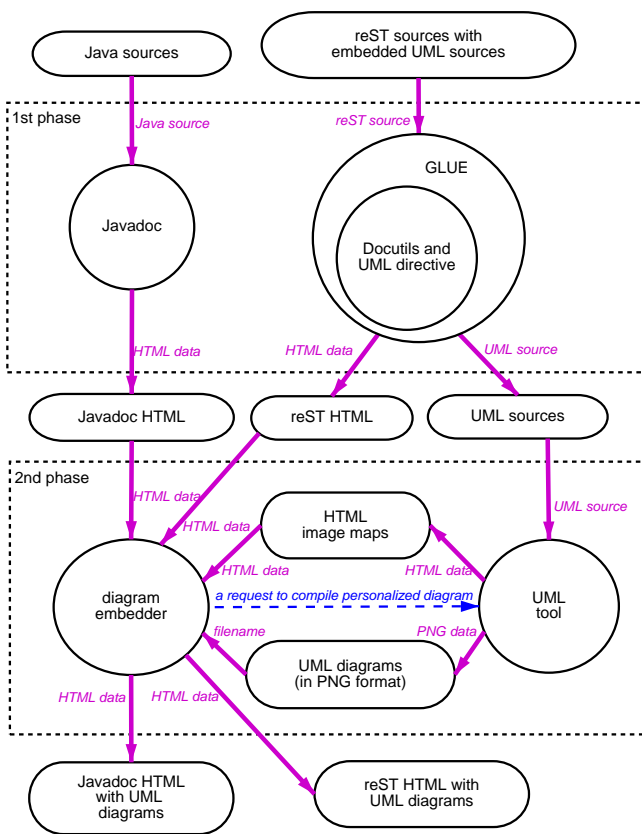


Figure 6: The two distinct phases of generating Javadoc and design documentation bridged via UML diagrams.

- Finally focused versions of UML diagrams are created and implicitly embedded into targets of all linked elements appearing in UML diagrams.

6. EXPERIENCES

The UML and linking tools are currently in everyday use in the Gzz project [?]. For now, the documentation covers about 380 Java classes and over 80 pages of design documentation exist [?]. We have only about 30 human-drawn UML diagrams embedded within the design documentation pages, but the number is continuously growing. For all those diagrams, the linking tool generates more than 150 separate differently focused versions, and by simple calculation the linking tool has embedded each diagram implicitly on average into four different design documentation or Javadoc pages. The UML tool description language covers currently 26 different types of diagram items and is extended regularly as necessary.

7. DISCUSSION

In this article we have discussed the hypertextualization of documentation in our software project. Using existing Free Software tools and a little new glue code we have made UML diagrams into contextual menus for switching between the design documentation and the detailed Javadoc documentation. The current implementation is a part of and in use at our Free Software project [?], where it

generates the publicly web-browsable hypertext [?] of the project's documentation.

The software automatically creates multidirectional links based on simple directives in the UML diagram source code in our documentation. Only web pages generated during the documentation build process are affected, which makes the task of our software easier than that of some other web augmentation tools [?].

Earlier concept-based navigation and map-based navigation have added horizontal links on top of hierarchical hypertext [?]. The structure we discussed in this paper is similar as the diagram becomes a navigational node, but we don't rely on keywords or heuristics in link creation. Further, as the diagram is shown inside a web page, it functions as a multi-target link.

There are a number of areas in which the current prototype needs to be improved. For example, currently the only links are from UML classifiers (i.e. classes, packages, objects, ...) to Java classes and packages and other documents. Methods (and fields) should also be linked, especially in interaction diagrams, as well as associations, whenever there is a suitable object in the code documentation.

The layout of the diagram is also currently more difficult than it should be. While the creation of the diagram structure is easier using written text than direct manipulation, the layout of the resulting structure would benefit from a click-and-drag interface. However, this approach would also lose some expressive power: in MetaPost layout the user is now allowed to draw anything on the diagram. This can possibly be solved by another stage after metapost, where the locations of the nodes could be interactively defined into MetaPost variables. On the other hand, there are also arguments against MetaPost: the error messages received on erroneous input to the UML tool are difficult to decipher due to the translation layer between. Abandoning MetaPost might also speed up the compilation process.

We plan, once some of these issues are resolved, to release the glue code as a standalone Free Software package. A plugin interface would allow linking to documentation in different format than the ones we have used this far. As the UML description language develops, it can support more diagram types, thus the tool would be deployable in new kinds of documentation. Using some other maps in place of UML diagrams, same kind of linking could be used in other hypertext applications.

Finally, we'd like to point out that this tool could benefit from a less limited presentation layer than currently supported by web browsers. Ideally, hypertext browsers would directly support navigation maps in their user interface, stabilizing the map positions during browsing.

8. ACKNOWLEDGMENTS

The authors would like to thank Toni Alatalo for discussions regarding UML software and this manuscript.

9. REFERENCES

- [1] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.
- [2] B. Boehm. A spiral model for software development and

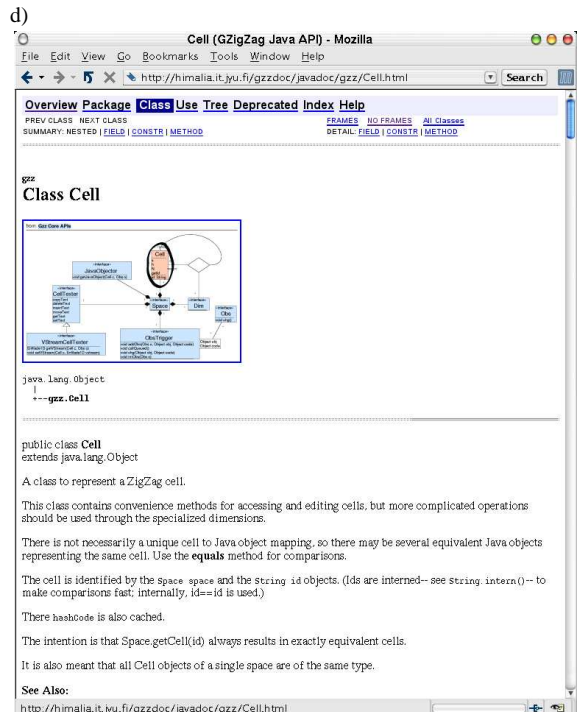
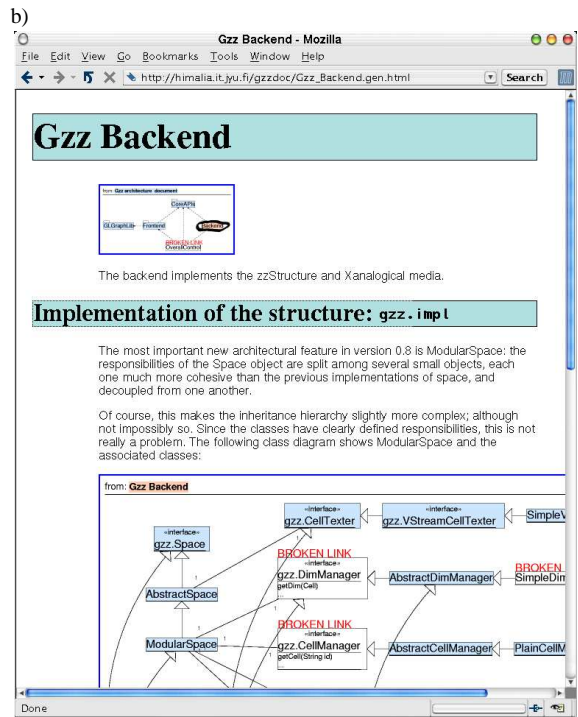


Figure 7: A collage of screenshots from Gzz project documentation.

- The front page of our design documentation. Items with blue background act as links, the item in orange is the active document, and “BROKEN LINK” marks that the item would be a link, but the target was unavailable.
- Moved to “Backend” documentation node. The diagram from the front page is implicitly included at the top of the page in small size. The item representing this node is marked with rough circling in addition to orange background.
- Moved to “Space” Javadoc interface class node. A couple of implicitly embedded diagrams show the various contexts for this package.
- Moved to “Cell” Javadoc class node. All diagrams are split in two by horizontal lines. Below a line is the diagram itself, above is a list of the documentation nodes where the diagram is explicitly included. Most of the diagrams in the current documentation is explicitly included only once. The diagram in this node is originally included in “Gzz Core APIs” documentation node.

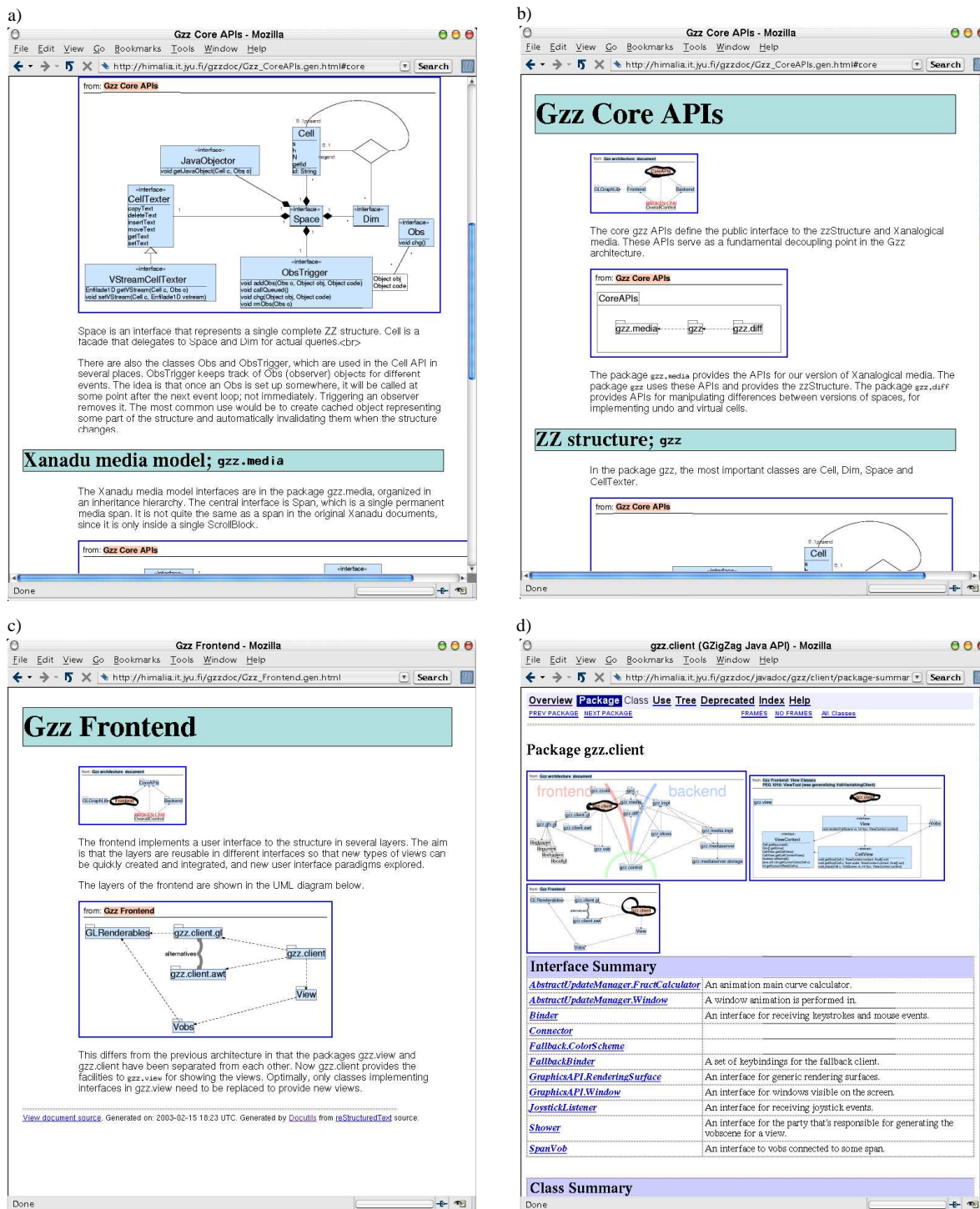


Figure 8: A collage of screenshots from Gzz project documentation.

- Moved to 'Gzz Core APIs' documentation node. The browser's viewport is automatically scrolled to show the diagram. Explicitly included diagrams are always shown full scaled.
- Moved to the beginning of 'Gzz Core APIs' documentation node. Already famous diagram is shown implicitly embedded at the top of the page in small size.
- Moved to 'Frontend' documentation node.
- The journey ends to 'gzz.client' Javadoc package node. The rightmost of the implicitly embedded diagrams can be seen twice included explicitly in documentation. The second explicit reference is done from node labeled 'PEG: ViewTool'. 'PEG' stands for 'Proposals for Enriching Gzz' and it's a crucial part of our design documentation for future improvements.

- enhancement. *IEEE Computer*, 21:61–72, 1988.
- [3] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.
- [4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.
- [5] F. P. Brooks. *Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.
- [6] P. Brusilovsky and R. Rizzo. Map-based horizontal navigation in educational hypertext. In *Proceedings of the thirteenth conference on Hypertext and hypermedia*, pages 1–10. ACM Press, 2002.
- [7] S. J. Cook and J. Daniels. *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice-Hall, unknown 1994.
- [8] D. Edwards and L. Hardman. Lost in hyperspace: Cognitive mapping and navigation in a hypertext environment. In R. McAleese and C. Green, editors, *Hypertext: Theory into Practice*, pages 105–125. Oxford: Intellect Limited, 1989.
- [9] GNU Classpath-tools.
<http://www.gnu.org/software/cp-tools/> Free Software Foundation, Inc, 2002.
- [10] L. Friendly. The design of distributed hyperlinked programming documentation, 1995. Presented at the International Workshop on Hypermedia Design '95.
- [11] D. Gentner and J. Nielsen. The Anti-Mac interface. *Communications of the ACM*, 39(8):70–82, 1996.
- [12] D. Goodger. An Introduction to reStructuredText.
<http://docutils.sourceforge.net/spec/rst/>, 2002.
- [13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JavaTM Language Specification*. Addison-Wesley, second edition, 2000.
- [14] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to Java. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 178–187. ACM Press, 2000.
- [15] J. D. Hobby. A METAFONT-like system with PostScript output. *TUGboat*, 10(2):505–512, 1989.
- [16] A. Larsson, C. Chplov, L. Clausen, and H. Breuer. Dia - a drawing tool.
<http://www.lysator.liu.se/~alla/dia/>, 2003.
- [17] P. D. Lebling, M. S. Blank, and T. A. Anderson. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, 1987.
- [18] T. J. Lukka et al. Gzz. <http://gzz.info/>,
<http://savannah.gnu.org/projects/gzz>.
- [19] T. J. Lukka et al. Gzz documentation.
<http://himalia.it.jyu.fi/>.
- [20] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, 1986.
- [21] R. Pierce and S. Tilley. Automatically connecting documentation to code with rose. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 157–163. ACM Press, 2002.
- [22] B. Shneiderman. Hypertext: An introduction and survey. *IEEE Computer*, 17(8):57–69, 1983.
- [23] ArgoUML - a modelling tool for design using UML.
<http://argouml.tigris.org/> Tigris.org - Open Source Software Engineering, 2002.
- [24] R. H. Trigg. A network-based approach to text handling for the online scientific community.
<http://www.workpractice.com/trigg/thesis-chap4.html>
Department of Computer Science, University of Maryland, 1983.
- [25] D. van Heesch. Doxygen - a documentation system.
<http://www.doxygen.org/>, 2003.
- [26] H. Weinreich and W. Lamersdorf. Concepts for improved visualization of web link attributes. In *Proceedings of International World Wide Web Conference*, 2000.