Université catholique de Louvain
École Polytechnique de Louvain
Département d'ingénierie informatique

Lively documentation: A dynamic annotation system for agile development

Promoteur : Kim Mens

Lecteurs : Sebastián González
Johan Brichau

Mémoire présenté en vue de
l'obtention du grade de
master 120 crédits
en sciences informatiques
option software engineering and programming systems
option networking and security

par
Pierre-Henri Van de Velde

Louvain-la-Neuve
Année academique 2008-2009

Je tiens à remercier Kim Mens qui a donné de son temps lors de l'élaboration de ce mémoire. Il a proposé de nouvelles idées qui ont permis l'élaboration et l'amélioration du programme. Il a aussi répondu a mes questions durant la rédaction du mémoire. Il a également pris le temps de relire et corriger soigneusement mon travail mais également de le structurer, afin que les idées principales ressortent mieux.

Je remercie ensuite Sebastián González qui fut présent tout au long de l'année et qui eu le temps, chaque semaine, de discuter du mémoire. Il a su pertinemment répondre à mes questions et me guider lors de l'implémentation du programme. Il a également corrigé mon travail et je l'en remercie encore. Sans lui, le mémoire ne serait pas ce qu'il est devenu.

Je remercie aussi Johan Brichau qui a accepté de lire ce mémoire. J'espère que la lecture lui plaira.

Je remercie enfin toutes les personnes qui m'ont soutenues tout au long de mes études, et les personnes qui m'ont encouragées afin que je réalise ce mémoire.

Merci à tous.

Abstract

A great problem with Agile development is that the documentation process is usually put aside, yet good documentation is key to usability and maintainability of software. This is why my thesis presents a new methodology that includes the documentation process in the agile development methodology. The key feature to the elaboration of this new methodology is the implementation of a new software that allows developers to annotate code entities such as classes, methods, and so on, with tags, examples, documentation, test-runs. It allows easy searching and structuring of the documentation. It is for example possible to retrieve all entities that have been tagged as "undocumented" or classified in a specific section. Examples and tests can be executed directly from the software to see the results. All these features make it easier to realize a documentation process that blends naturally with Agile programming techniques.

Contents

Chapter 1

Introduction

1.1 Context

The Agile development methodology [9] [70] enables iterative conception of software that respond to the needs of the client. To do so, the methodology recommend, amongst other things, to involve the client in the elaboration of the project rather than to negotiate contracts or to propose a working program rather than a complete documentation [55].

In Agile methodologies, accent is put on clear and self-documenting code and techniques like refactoring to adapt to change. Unit tests are also of great importance as they assure of the validity of source code.

1.2 Problem

In that context, developers that write source code focus on the functionalities to implement, but technical documentation is sometimes put aside. As the program evolves, documentation that was supposed to comment the source code does not reflect the current source code anymore. It might even happen that documentation is inexistent even though technical documentation is the entry point to source code comprehension.

Today, documentation tends to be static in a methodology that needs ever improving source code that changes and evolves over time. Problems like inappropriate documentation or inexistent documentation are then recurring and understanding source code becomes more difficult. The purpose of Agile development is not to promote exhaustive documentation [55] [71]. Precise and useful documentation is not excluded of that process neither. Not enough accent is put in the documentation process. This might be due to the fact that documenting code is time consuming and that documentation systems do not propose to handle documentation in the iterative phases of Agile methodologies. The fact that source code should be self-documenting or that unit tests might be considered as part of the technical documentation might relegate technical documentation into the background.

1.3 Thesis statement

It is possible to improve software documentation and especially technical documentation in a software development methodology that address change like Agile development methodologies by focusing on documentation in the process and by bringing to the developer a mean to accelerate and improve technical documentation. That documentation process should adapt to change in the same way as the source code.

1.4 Solution

To improve technical documentation in Agile development, programmers should rely on a methodology that gives an important role on technical documentation in software development. A documentation system could help the programmer in that process.

That documentation system should improve documentation organization and documentation retrieval. It should also permit to document the programmed system with comments as well as code examples or unit-tests. These possibilities improve documentation by centralizing software documentation in a single system and help the programmer in that process. Tools of the system should propose operations on documentation in the same way IDEs propose operations on the source code. One of those tools, that accelerate the documentation process, is a tagging system that attach defined documentation information to the entities of the system. The documentation system relies on techniques to annotate source code with useful information that is already present, like examples or with new information that is easy to add, like tags. It is also possible to add classical comments to the entities. These features are to help software documentation in a changing environment.

1.5 Outline

Chapter 2 presents the different types of documentation and the existing documentation tools. Documentation methodologies that center on software documentation are also presented. These existing tools and methodologies are then analyzed to identify their advantages and limitations, which will help us identify the needs of a documentation system.

Chapter 3 talks about the requirements for a documentation system and shows a mockup that presents how to respond to those requirements.

Chapter 4 presents the implemented documentation system and the tasks that can be executed with that system. This chapter also presents the methodology that guides the programmer on software documentation in the software development process.

Chapter 5 shows how the system was implemented and explains the technologies involved in the process.

Chapter 6 tests the documentation system on a real example to asses of the qualities of the implemented software. A discussion explains the results, the advantages and the limitations of the implemented documentation system.

Chapter 7 summaries the different steps of the elaboration of a documentation system and discuss about the encountered difficulties. It presents the further work that could be realized to improve the documentation system. Finally it presents the results of the thesis and explains how the implemented system can help in software documentation.

Chapter 2

Background

To improve software documentation it is important to understand the existing documentation systems. Comments can be used in different situations and might address distinct persons. But commenting source code is not the only way to provide technical documentation that helps understanding source code. Does a programmer consider that examples are documentation? Even if it is not clear, people tend to use examples to grasp the source code. Are there other ways to document code? If it is the case, these other techniques are important to understand software documentation. Several tools and methodologies deal with documentation. This chapter will present how they help the programmer but also what are the limitations of such documentation systems.

2.1 The need for code documentation

It is not sufficient to develop well, it is also important to document produced code. Later on, it will often be another developer that will face the code that someone writes. It is therefore a crucial step. That step is however often neglected by developers that think their code is very clear.

The problem with this behavior is that code evolves and becomes more difficult to understant. This is why it is important to comment code all along the programming process. Documenting source code is an exercise in which the programmer takes a step back at what he is coding. At that moment, it is possible to see if an implemented feature is correct and addresses the problem to resolve. If the feature is difficult to explain in natural language, it might be difficult to implement as well and should be worth reconsidering.

Documenting code has to be considered not only for the programmer's own purpose but also for the others. There are distinct situation in which documentation has to be provided. These situations will now be presented.

2.1.1 Documenting alone

When programming alone, there are often no constraints and the only limitations are the programming language limitations and what's in the programmer mind. While coding, people forget to put comments on a feature and postpone the documentation process to later in time. Finally, the programs ends up with some notes or even no documentation at all. When returning to the program after having let it rest for a while, it might be quiet difficult to understand what has been done, letting the programmer no choice than startup from scratch again instead of using strong bases.

Documentation is a process that should be taken care of way before actually coding. A good practice is to document a feature before coding it. This allows the programmer to understand what he really has to do and if it takes sense. "*Ce qui se concoit bien s'ennonce clairement...*" is an important principle that takes all it sense when one has to explain what he is about to do. If it's not clear, it's maybe not the point to code it and one should reconsider the feature.

2.1.2 Documenting in teams

Most of today's software are made by a team of programmers coordinated by a project leader. This type of projects involve multiple people working on different parts of the whole program. Programmers needing to use another's source code might encounter difficulties to understand the code alone if no documentation is available. That makes documentation a necessity. It might also happen that another programmer must take over another's source code. Clear documentation then accelerates the work of that person, helping him to better understand the code. As we can see, documentation is unavoidable in big projects.

2.1.3 Documenting for others

People involved in the development of an application programming interface (API) are not only dealing with programmers from their team. They must provide documentation if they want external people to actually use their API. If there is no documentation, no one will understand the features and the need to use the API. Would you ever imagine using a API for your project where you would have to dive into the API's code to actually understand it? Some programmers find it normal to do so. For programmers that are accustomed to the API and the programming language in which it is developed, this do not seem to be a problem. It is obvious that this is not the case for people discovering the API. The first steps to understand source code need a big assistance provided by the documentation.

For a user to actually understand an API, how-to's and tutorials must also be written. This is a part of the documentation process that can partially be done with existing tools. With Doxygen [51] for instance, it is possible to add examples in the technical documentation. This illustrates the fact that software documentation is not limited to comments in the source code. Users rely on help to start with the use of APIs. To do so, the documentation must be useful and well structured.

If people can easily find their information, the documentation will become an important tool of the everyday life of the programmer. On the other hand, useless information will be dissmissed. Useful documentation need structure, cohesion, fast information retrieval to be of any use for the programmer especially at the learning phase. Because everything is new for him, the user of a technical documentation will need most help in that learning phase.

We saw that the process of documenting is important in a project. Software documenting is not only about running a tool to generate an API documentation automatically. Other content can also help understanding source code, like diagrams, code examples and so on. There are however several types of documentation. There are also many tools to help the programmer all along the documenting process. In the next section we will see the different types of documentation.

2.2 Types of documentation

Until now we essentially discussed about documenting source code, which, with the help of a tool, can generate what is called **technical documentation**. Documentation is not only about generating technical papers like any Javadoc-like [50] program could do. There are several kinds of documentation that serve several purposes. It is important to distinguish these different types of documentation.

2.2.1 Architectural documentation

The first type of documentation, essential for the existence of big softwares is **architectural documentation**. This documentation describes diagrammatically how the distinct components of a system should work. To represent how this system should work, several views are necessary. A logical view describes the static an dynamic components of a system and helps to identify the different elements that will be part of the system. A process view describes the interaction of the different components or an interaction of the users with the system. Several models exists to represent those informations. Among them, UML [48] is a modeling language that defines several types of diagrams to illustrate the views of a program. UML however, does not only target the architecture of a program, but also the design. The information that can be retrieved both from the architectural and design documentation of a program, produced with UML for instance, is a great information for anyone needing documentation about a system. Architectural documentation is the big picture of how software is built and how it works [57]. The problem with architectural documentation is that it is sometimes to technical and often to vague to give precise information like "how to use the system?" or "what does this class do?". This is especially true for people that do not need to know how the system work but want to know how the system can be used. Other types of documentation addresses these questions.

2.2.2 End-user documentation

End-user documentation or **manuals** are important for users that have no specific knowledge about programming. Manuals are of great importance as they explain the features of a program and help the user understanding a program. If we take a look at several libraries for Web 2.0 [41] programs, we can distinguish two types of projects, the documented ones and the others. It is often the manual, together with examples and beginners guides that make the difference between adopting a API or another. That's why end-user documentation should not be underestimated but considered altogether in the achievement of a project. End-user documentation however, is used for the early stage of the learning step of a program. Once the user has a feeling of what he can do with software and how he can use the basic features, he will need more precise documentation of how to do more technical stuff that cannot be addressed in the end-user documentation.

2.2.3 Technical documentation

Technical documentation explains what the code does and how it does it. It is not targeted at people that do not understand programming. Several programs extract the comments in source code to generate technical documentation but this does not exempt the programmer to write correct documentation. It is however a tool that can bring some ease of use by generating more readable papers like HTML pages with colors and hyperlinks that helps navigating the documentation. As we can see, technical documentation can go from documented source code to generated files like HTML to even more advanced techniques that will provide information to the programmer. These documentations can be separated in roughly two kinds: static documentation and dynamic documentation.

Static documentation

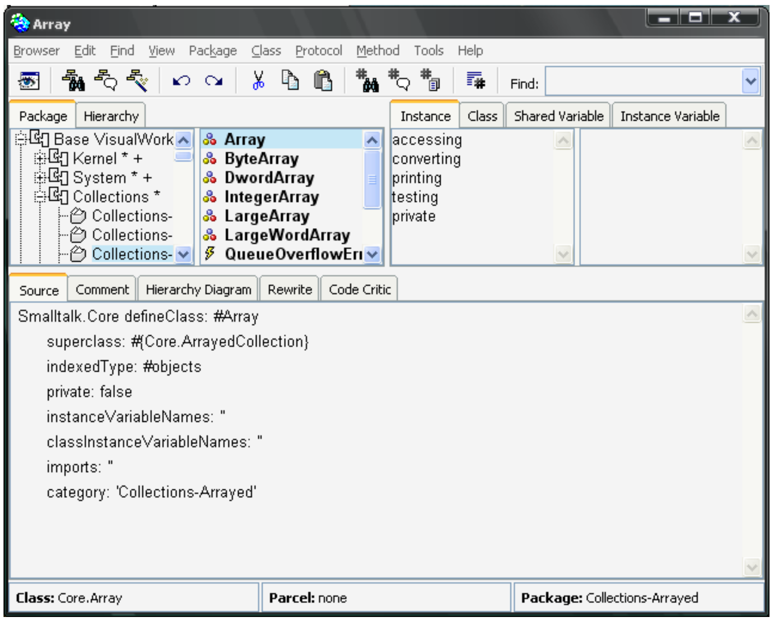
Lots of common programming languages rely on comments in the source code. These comments are usually discarded at compile time and are not part of the executable. To generate documentation for these types of languages, a tool will read the source code and extract all the comments to generate technical documentation. Some advanced documentation tools allow the programmer to add information in the comments like specific tags that will be used by the tool. This will provide more information in the technical documents like parameters types and return values, code examples, etc. Some of these advanced tools can also generate diagrams from the source code. This way of documenting source code is considered as static as it is parsed from the source code. The technical documentation is only a snapshot of the program at any given time. When the code changes, a new snapshot need to be taken. Javadoc [50] is a classical example of technical documentation tool. It generates documentation for Java [26]. The code fragment 2.1 shows the documentation of a method `foo` that takes three inputs, the first is number of times to print "foo", the two other are summed, which is the return value. As we can see, the tags `@param` indicate the parameters of the documented method and `@return` indicate the return value of the method. These two tags are recognized by the Javadoc tool and receive a specific treatment.

Listing 2.1: Example of Java documentation

```java
/* The method foo prints ``foo'' on the standard
 * output x times and returns the summation of
 * the parameters a and b.
 * @param x - The number of times to print ``foo''
 * @param a - The first number to sum
 * @param b - The second number to sum
 * @result Returns the summation of parameters
 * a and b
 */
public int foo(int x, int a, int b) {
    for (int i = 0; i < x; i++) {
        System.out.println(''foo'');
    }
    return a+b;
}
```

Dynamic documentation

Lots of widely used programming languages uses external tool to generate a form of technical documentation. Some are provided with the language, like Javadoc. Others are external applications, like Doxygen [51]. Some languages have yet another approach to documentation. The idea behind it is



```
Smalltalk.Core defineClass: #Array
    superclass: #{Core.ArrayedCollection}
    indexedType: #objects
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'Collections-Arrayed'
```

Class: Core.Array | **Parcel:** none | **Package:** Collections-Arrayed

Figure 2.1: Smalltalk IDE: VisualWorks documentation tab

to make the documentation part of the program. That way of doing allows the programming language to interact directly with its documentation. A language that uses that approach is Python [29] [49].

In Python, it is possible to put a docstring just after the declaration of a class or a method. This docstring becomes an attribute of the method or the object which makes it accessible at runtime. The code fragment 2.2 shows the docstring of the `int` object. The `help` function allow to retrieve that information along with other useful information a programmer might need to use an object of Python.

Listing 2.2: Example of Python docstring

```
int.__doc__
'int(x[, base]) -> integer
Convert a string or number to an integer, if
possible. A floating point argument will be
truncated towards zero (this does not include
a string representation of a floating point
number!) [...]'
```

If we take a look at Smalltalk [27], which is a highly reflective language, it isn't possible to interact much with the documentation. Documentation is however strongly coupled to the integrated development environment (IDE) and might encourage the developer to use it for documenting code. As we can see in a screenshot of the IDE of Smalltalk (2.1) [53], a tab is available for commenting the code. Another information that can serve as documentation for the Smalltalk programmer is the section in which a method is classified. This helps the programmer to know what the purpose of the method is.

Dynamic documentation and IDE supported documentation offer new possibilities that are not supported in static documentation. These approaches offers the programmer a new way to make documentation with tools helping him in the process. Python for instance offer the possibility to the programmer to get the documentation of an entity at any given moment. If the documentation is updated and the user asks for it, the new documentation will be presented. The user does not even need to worry about compiling a new technical paper before reading it. It could be interesting to use these kind of features to provide the programmer with tools allowing him to get the best of documentation.

2.3 Beyond generated documentation

It is not sufficient to write some pieces of comments and run the tool to make it accessible to someone out of the domain of the program. Java programmers know the language and how to use it. They share the same vocabulary that allow them to communicate with structures specific to Java but someone new to the Java world cannot start programming in that language with the help of the API documentation only. What a beginner needs are examples, how-to's, guides. If you look at the Python web site, you can see the documentation of the language is subdivided in several sections like: Tutorials, Using Python, Language Reference, Library Reference etc [30]. It clearly points out that an API documentation generated from one or the other tool isn't sufficient for someone to actually use Python.

2.4 State of the art

We saw that documentation can be classified in different types and that each type of documentation do not necessarily target the same persons. One common fact is that programmers need technical documentation. Some other models of documentation have also been presented. These models focus on documentation and propose a way to integrate it in the implementation of a program. We will analyze some of these means of documentation and see what they bring to documentation.

2.4.1 Comment parsers

If we take a look at the most used programming languages today, but also in the past, we can observe that a large number of languages provide a way to add comments directly in the code. Comments are specified as being escape characters allowing the developer to put phrases or annotations in the source code. In the past, most of the time, those annotation where not treated by the compiler. As a result, there is no way to access the source code in the compiled program. Typical languages that are designed that way are C, C++, Java, Fortran, Assembly, Python, etc.

The Javadoc tool is certainly one of the most known and most used tools for generating technical API documentation. The principle of the tool is to run it with the source code as input. The output will be a full documentation of the program generated in HTML format. As we saw in the source code example 2.1, there are reserved words that allow to annotate the documentation with useful information like references to other classes, parameters and return values for methods and many others. It is even possible to add HTML markup directly in the source code. The advantage of using such a tool is that reading the API in a web browser is much more convenient than in the source code of Java.

Doxygen is a tool that transposes the world of Javadoc to the other languages, like C, C++, C#, Python, with some interesting extensions. The output is not limited to HTML. The documentation parses the source code and add it to the documentation with syntax coloring. From any point in the documentation, it is then possible to see the corresponding source code fragment. Some dependency graphs, inheritance diagrams, and collaboration diagrams can be generated automatically. Doxygen is therefore certainly the most advanced tool when talking about technical documentation generation.

Advantages

The advantages of code parsers in general is that they can be run after the program is completed into several outputs. The number of existing tools today provide the user several distinct features and the possibility to tune the final result of the documentation. Moreover, those tools are most of the time straightforward to use. The output is often similar from one tool to the other, which helps the reader in browsing the documentation.

Limitations

On the other hand, for a code parser to generate useful documentation, the programmers have to actually write well documented code. This is something that programmers do not always do. The generated documentation is a static view of the program. When a new version comes out, the documentation must also be regenerated. After some time, code changes and evolve, but programmers tend to forget to update the documentation as well and it can happen that the documentation does not reflect the source code anymore. Comment parsers cannot see these types of mistakes as there is no version indication. Another bad point is that the programmer needs to know about the syntax of the comment parser. These are different from one tool to the other. Fortunately, this is not really a big problem in practice as syntax stays simple an takes a lot of well known representation languages (like HTML for instance).

Looking into the source code of a program to generate the API documentation isn't the only way to create documentation. It is however the most common for most of the programming languages, but other techniques exists. Unix systems for instance provide a way to access documentation without browsing the source code.

2.4.2 Man pages

People that think about documentation usually think about comments in the source code and Javadoc. These are rather language-specific. Their output is cross-platform but is not strongly coupled with the operating system. If we take a look at Unix systems in general, it is possible to view the manual of a program by typing `man` followed by the desired program in a terminal. It has nothing to do with technical documentation at first sight. For most of the programs, it is a user-manual, like the name of the functionality suggests, but if you want information about a specific function of the **C** API, you can use man pages as well. This is very useful for C programmers. Whenever they want to know about a C function, the man pages are the first place to find the information.

Man pages are divided in nine sections: (i) Executable programs or shell commands, (ii) System calls, (iii) Library calls, (iv) Special files, (v) File formats and conventions, (vi) Games, (vii) Miscellaneous, (viii) System administration commands and (ix) Kernel routines.

The most useful for a simple user is certainly the first and the seventh section. For programers, the second and the third are used a lot. Each man page is made of several paragraphs. Depending on the program listed, all or some of the paragraph will be available. They will however always be displayed in the same order.

Advantages

An interesting advantage of the man pages is that they are accessible to all users on a Unix system. It is also possible to make man pages [12]. The difference with the other system that we have seen so far is that man is part of the system and makes it a standard. That is certainly why man pages are so often used in the Unix world. This is a huge advantge. Every manual, be it for a program or a function of a programming language, is represented the same way, with the same sections. Once a programmer is used to man pages, retrieving the needed information becomes easy. Another advantage of it is that it is multi-purpose, which is a second reason for its widely use. System calls and library calls for the C programming language are included in the system.

Limitations

Of course, there are some limitations to man pages. It is not possible to directly show man pages for other programming languages. On could imagine a section *programming* with subsections for each language installed on the system. This would extend the purpose of the man pages to every programming language, making it a big standard for manuals and technical documentation. Perldoc, a documentation tool for Perl provides this functionality. If the programmer has written a manual for his Perl program, a user can type `perldoc program_name` to see the manual formatted as classical Unix man pages.

Man pages also suffer from the same limitations as generated documentation from the source code. If changes are made to a function, the man pages must be re-generated. As the name also states, man pages are for manuals. They are great to explain how to use a program and what are the arguments to a command line, but the use of it for programming languages is less intuitive. Moreover, navigating man pages from one function to another is not that easy compared to technical documentation.

Comment parsers and man pages rely on the fact that the programmer includes documentation in his program and generates the technical documentation or the manual. These two approaches do not help programmers to organize their code and to display information the way they would like it to be displayed. The presented tools only provide a solution for presenting documentation if it is available. How can the programmer be helped in the actual fact of writing documentation?

2.4.3 Integrated Development Environments

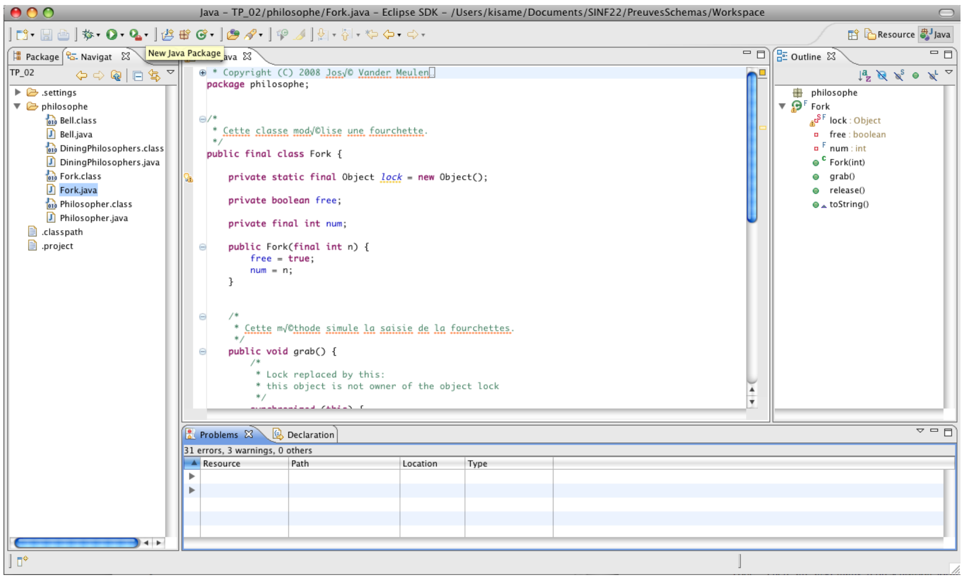
Integrated development environments are tools that become the every-day friend of a programmer. These integrate not only a possibility to edit code and have nice color syntax, but a lot of other features. As the IDE grows, more and more plugins are added to it. Every programmer has his own way of programming and wants to add something to the tool. An IDE also need lots of customization. IDE's can also help in the documentation process.

If we take a look at the Smalltalk IDE for instance we can observe that there is a dedicated tab to add comments for the code. This is already of help for the programmer. When he sees the tab, he knows code must be documented. Smalltalk methods must also be classified in distinct sections called "protocols" to explain the purpose of the method (like accessors, printing, testing or private). This is another form of documentation. Classifying methods according to their purpose gives the ability to a programmer to structure his code but also the possibility to browse and understand code faster. The drawback with Smalltalk is maybe that it is mandatory to classify the method and a method can only be put in one section. When a method has crosscutting concerns, it becomes difficult to classify it correctly. On the other hand it is also an advantage as it will force the programmers to write methods that focus on a single task which is an important OO design principle.

The Java IDE Eclipse [52], represented in figure 2.2, has also some interesting documentation tools. The possibility to annotate code with a "todo" tag is a very powerful feature that allows programmers to easily retrieve parts of the code that need modifications or that seem problematic. A window shows all the todo-tags with their annotation and allow the programmer to easily browse to the concerned file.

Automatically generated documentation also helps the programmer. Once the generic documentation strings are automatically added, the programmer can then easily comment the code without effort an let the IDE do the repetitive work. Behind that, Eclipse offers many other advantages that do not concern documentation. It shows multiple views of the system that help the programmer to structure and retrieve code efficiently.

- On the left, a window displays the content of the projects structured like a classical file browser. The programmer can navigate files of a project efficiently.
- On the right, the outline window displays the content of a single file with classes and their attributes and methods. Views can also be



```
 * Copyright (C) 2008 Josée Vander Meulen
package philosophe;


/*
 * Cette classe modélise une fourchette.
 */
public final class Fork {

    private static final Object lock = new Object();

    private boolean free;

    private final int num;

    public Fork(final int n) {
        free = true;
        num = n;
    }


    /*
     * Cette méthode simule la saisie de la fourchettes.
     */
    public void grab() {
        /*
         * Lock replaced by this:
         * this object is not owner of the object lock
         */
```

31 errors, 3 warnings, 0 others

Figure 2.2: Java IDE: Eclipse

configured depending on the programming language that is used with Eclipse.

There are many other views and each of these can be fully customized by the user of the IDE. This give lots of freedom to the programmer to arrange his environment the way it fits him the most. Even if these features do not seem to concern documentation at first sight, they help the programmer to retrieve information. Information retrieval is also a key component for documentation.

Advantages

IDEs goes far beyond documentation but as we can see, some interesting features help to document code effectively. The multi-view that an IDE gives from a program is also an important feature that help the programmer, not only to document his code, but also to organize it. When things are organized they are somehow self-documented as the protocols in Smalltalk are for methods.

Limitations

An IDE can bring a lot of conveniences to a programmer but when the programmer forget to document his code, the only solution to produce one is to read all the code and add documentation manually. The problem is that programmers focus on code and not on the documentation. But source code is sometimes difficult to understand as the name suggest.

Another way to provide documentation is to reconsider all the way programming is presented and give code a human understandable semantic or to give documentation a central place in the development process. Some programming paradigms took that step.

2.4.4 Literate programming

Usually, documentation is needed in programs because the source code is difficult to understand. This is certainly a reason why documentation is still needed today. Donald Knuth [54] proposed another approach, literate programming [10] [72], where a program is explained logically like a natural language. In this paradigm the programmer uses macros to describe what he is doing with phrases. These macros hide chunks of code and other macros. Code is typically written in C but it is also possible in C++ or Java. The root macro is defined by `<<*>>`. The code fragment 2.3 shows the main file of the program that describe what it does and that defines the root macro and sub-macros to include files and to define the main program.

Listing 2.3: Root CWEB program file

```
The purpose of this program is to show how literate
programming works. The main program propose a simple
factorial function

This illustration program is defined by the
following files
<<*>>=
<<include files>>
<<program>>
@
```

The main program is then documented in code fragment 2.4. The file contains the implementation of the program. As we can see the program is written in C augmented by the macro definitions and a lot of sophisticated documentation.

Listing 2.4: main of CWEB program

```
This chunk is the main program and provides the actual
implementation. What is provided here is a function that
counts the factorial of a number. If the number is one,
the function returns one, otherwise, the function
recursively counts the product of the value
 and the value-1.
<<program>>=
int  fact(int n) {
    if (n <= 1) return 1;
```

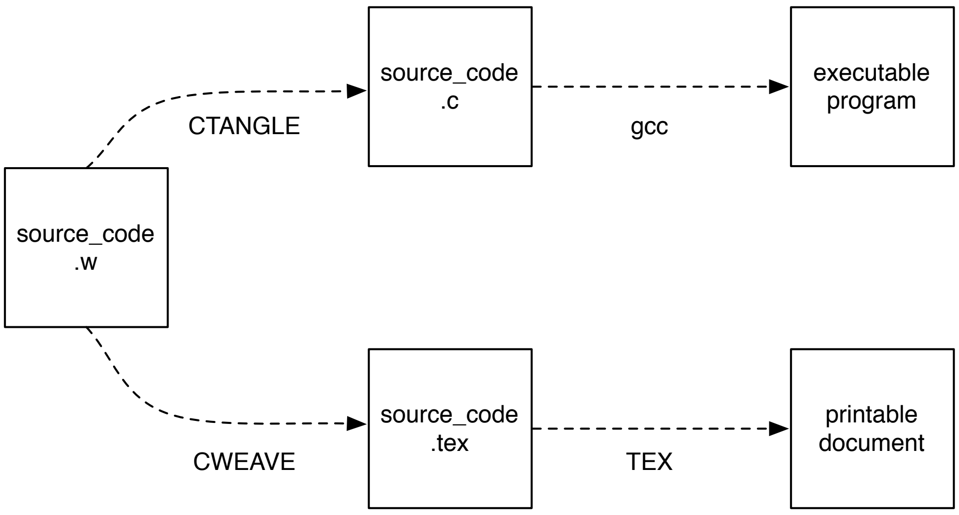


Figure 2.3: From a CWEB file to executable and documentation

```
        << do the recursive call>>
        else return n*fact(n-1);
}
@
```

To produce the output of a literate programming file, the program parses the source code. From the source code, one program (CWEAVE) will extract the documentation code an generate a .tex file, the other program (CTANGLE) will generate the .c source code file. Finally the TEX compiler will produce the documentation and a standard C compiler, like gcc, will produce the executable file. The process is represented in figure 2.3.

Literate programming offers an approach to programming that focus on comprehension of the code. It is achieved with lots of comments and a logical way to structure the program.

Advantages

Advantages of this paradigm is that the programmer focus on expressing the thoughts of the program, providing better documentation and also better designed code. The source code is then extracted on one side and documentation is assembled on the other side. Whenever source code is produced, documentation is produced as well.

Limitations

As we saw in the example, literate programming uses a normal programming language for the source code. This means that even if the programmer does explain his code with words, the programmer still need to switch from natural language to code. Sometimes, code examples can also explain what code does and how to use it, which make the code easier to use. Literate programming also gives the impression that source code is only present as illustrations for a report.

2.4.5 Example-centric programming

Lots of programming languages are learned with examples. Whenever a programmer tries to understand software, he does it by mentally running examples. Lots of books also propose to learn a programming language by examples. This common approach has seen a new programming methodology called example centric programming [11]. The motivation of example-centric programming is to give IDE support for examples throughout the programming process.

In example-centric programming, the programmer writes examples that will be interpreted automatically by the IDE. The programmer writes his code normally and then add examples that illustrate the code. On the other hand, the IDE will interpret the examples according to the code and show an execution trace of the examples. With this methodology, abstract code becomes concrete which in turn, makes it easier to understand.

Advantages

The advantages of example-centric programming is that concrete examples are written. These examples are not only useful to explain and understand the code. They also have the advantage that an execution trace is produced. This execution trace shows the behavior of the program but also explain what the program does. All this process is automatic and the programmer can directly benefit from these advantages.

Limitations

Example-centric programming uses a natural programming language, like Java, and rely on the power of the IDE to produce the trace execution and the duality code/example. Sometimes, the problem in understanding code comes from the programing language itself. Programing languages are still an abstraction which sometimes makes it difficult to understand. This is why some methodologies focus on the programming language directly by making it more understandable for human beings.

2.4.6 Intentional programming

Another way to improve documentation could be to improve the programming language itself. One can compare programming to write a set of instructions to operate a computer. It is however possible to have a better approach where programming is comparable to communication between human beings. This is what intentional programming [73] proposes. The purpose of intentional programming is to explain what to do in a humanly understandable way instead of in a machine language. It is really seeing the process of elaborating programs from another point of view. The programer explains what he wants to do, not how the computer should do it. The nice advantage of proceeding that way is that the process of documentation is actually the process of elaborating the program. If we take the the example of printing all numbers from one to then, a programmer could write the following code in Java:

```
for (int i = 1; i <= 10; i ++) {
    System.out.println( ‘‘number ’’ + i );
}
```

The previous example is quite standard and easy to understand for every programmer. However, if you think at it, the programmer never says he is printing the number from 1 to 10. It seems straightforward if you look at the source code, but this might not always be the case. Loops can be much bigger, containing a lot more code and on top of it, programmers sometimes do not use explicit variable names. A good programmer could indicate what he does using comments. But usually, it is not the case, making the program yet more difficult to understand. If some other methods with fancy names are called in the process, it becomes very difficult to understand what the initial behavior of the program was.

```
print number for 1 to 10
```

The idea is thus to represent the actual behavior of the program like a human should explain to another human, leaving the process of actually generating the source code to a machine. Indeed, who else than a machine can better talk to a machine? It is also true for humans. This gives us a programing language close to human representation, making it both easy to use and to understand.

Advantages

The purpose of intentional programming is to deliver programs that are easier to maintain and to browse. This is also a purpose of documentation but instead of explaining what some mysterious code does, intentional programming goes to the root of the problem and remove the notion of code to make programming a way for programmers to explain what to do and letting the machine do the translation.

Limitations

Of course, literate programming cannot be applied to existing languages. This means that a new language is needed to support intentional programming. Changing the programming language is not always possible and lot of programmers used to a language develop in a way that is self documenting. Smalltalk programmers will tell that methods with the right protocol and the right naming conventions already tells everything about the method before even reading the code.

2.4.7 Summary

Documentation can take multiple forms. The most obvious is comments in source code but it can also be code examples for instance. Some tools helps the programmer to read the documentation of a program more easily. This is the case of comment parsers like Javadoc or Doxygen. An IDE can bring information that helps the programmer browsing and understanding source code. Some methodologies focus on new aspects to make code more understandable, with more comments in literate programing. By making examples part of the program with example-centric programming or even by changing the language in intentional programming. These advantages are summarized in table 2.4.

2.5 Limitations of existing approaches

There are many ways to document a program, from architectural documentation to technical documentation. Some methodologies also give documentation a central place in the implementation process. Tools that can help in the documentation process, like IDEs have also been presented. We will now see why these solutions are not perfect.

2.5.1 Generate documentation automatically

The problem with code parsers is that the programmer need to generate its documentation explicitly. When Eclipse compile the Java code of a program, no documentation is generated and the programmer uses his tool to browse the code and understand it. Why should the Java code be compiled and not the documentation if documentation is that important? Why bother generate documentation if it isn't to read it? The programmer that has its documentation at hand will certainly use it and thus maintain it. This imply that documentation is generated automatically. This can be done

source code doc.	On same place as source code Simple to use Easy to edit
comment parsers	Multiple outputs formats Support specific notations Easy navigation with HTML Direct access to source code Syntax highlighting
man pages	Integrated to OS Standardized representation
IDE	Fully customizable Multiple views of the system Organize code Retrieve information easily Integrated documentation features
literate prog.	Programmers explicitly state what they do Documentation is inherent to the process Exhaustive use of phrases to document code
example-centric prog.	Examples illustrate the code Execution trace is generated from examples Programmer can browse execution trace Execution trace + examples make code clear
intentional prog.	Focus on code comprehension Code is auto-documented Source code is extracted to be compiled Easy navigation from trace to code

Figure 2.4: Benefits of current documentation

externally like Eclipse does when compiling Java source code in background. Another way is to rely on docstrings that are kept durning the runtime of the program. This can greatly enforce the coupling between documentation and programming.

2.5.2 Edit documentation

The problem with static documentation is that it is not modifiable directly. When a programmer wants to change something, he has to return in the code, which is finally postponed to a later moment in time. Editing the documentation directly and annotating it with any useful information the programmer might need would increase the accuracy of the documentation. A program evolves all the time and so must the documentation if we want better documented programs.

2.5.3 Different activities, different needs

Documentation is not only about comments and documentation strings in a program. Documentation does also target distinct persons durning the evolution of a program.

- A programmer uses its own documentation to understand what he programmed (if he wrote a documentation). This often happens in the source code.
- The tester must verify that the program works correctly. Those tests shows how the program works and can serve as examples much like example-centric programming proposes.
- End-users need documentation to understand a program. This is can be a manual for end-user applications but also other programmers if the program is an API. These people will need examples, tutorials and technical documentation.

Today, documentation tools only give one vision of documentation that is not always useful for different persons.

Documentation is often considered as something independent of programming but all the examples presented before could be integrated and provide a documentation system that is more complete, that also represent documentation differently according to the needs and retrieve the best information needed depending on the user.

2.5.4 Make the most of dynamic languages

Some languages allow to access the documentation at runtime with a docstring. This give some flexibility that was not possible with languages that does not parse the comments into the program. At any given time in a Python program, it is possible to get the documentation of any entity. It is also possible to put tests in the docstring. These tests can then be run to check if the program is correct. This is a feature that deserves more interests. The danger is of course that documentation becomes more difficult to read because multiple information is saved in one place.

2.5.5 Organize documentation

We saw that protocols are an important feature of Smalltalk that helps programmers keep their code organized. It is a convenient way of classifying code and to retrieve something effectively. When a method is classified under "accessor", the programmer directly knows it is to access fields of an object and when the programmer wants to find a method that prints something, searching in the "printing" section is a good way to start with. This feature also reduce the need to write comments in the code, leading in a more self documented program. The problem with Smalltalk protocols is that methods can only be classified in one section. On the other hand, programmers try to code functionalities that addresses only one purpose, which is a better practice. But the code could also be marked with more than only behaviors of an entity. Entities could be marked according to their type and to the design pattern to which they belong if there is one. Marking could be useful for tests too. This functionality could be extended to annotate any entity with any useful information. When the programmer is looking for something specific, he can search for the entities that are marked with that information.

2.5.6 Separate concerns

Until now, one big assumption of documentation is that it is part of the source code of a program. This is maybe a limitation that has no reason to be. The problem whit adding information in a docstring is that it makes it unreadable as the information added into it grows. Adding comments, examples and test-runs in the same docstring would lead to more comments than the actual code. It might even be that the user do not need to see tests but wants illustrative code examples. The programmer that document his own program would not need to show the tutorials of his documentation but once he need to write a tutorial, a feature could help him indicating different pieces of codes that belong to the same example. When dealing with crosscuting concerns the problem also is of where to put the documentation. Repeating it unnecessarily increases human errors. Documentation for a

source code doc.	Extra work for the programmer
comment parsers	Need to explicitly generate documentation Modification only in source code Does not reflect program changes
man pages	Not convenient for browsing between entities Do not really fit to OO programming Suits more to manuals than to programming
IDE	Does not focus on documentation Not suitable for end-users
literate prog.	Lots of babble Suits more articles than real coding
example-centric prog.	Need an IDE
intentional prog.	Need a new programming language Project abandoned

Figure 2.5: Limitations of current documentation

crosscutting concern can be stored at one place. Whenever a feature of the documented program deals with that concern, the appropriate documentation information is shown. The ability of a documentation system to show the appropriate information according to the needs therefore important.

2.6 Conclusion

There are always two sides to a coin and the presented existing approaches to documentation do not break this rule. The limitations of these approaches are presented in table 2.5. As we can see, lots of features can be added to make documentation a part of the programming process. Software documentation could benefit from multiple improvements with integration of the documentation as central point. Integration of the documentation in the program, integration of the programmer in the process of documenting, integration of documentation features into documentation to make it a tool and a friend of the programmer but also the tester and the end-user. This can be done in several ways. Making documentation dynamic is one part. Adding classification to documentation is another. Integrating it with other tools that help understanding a program is a third. Allowing the programmer to directly interact with its documentation and see it live like its programs is certainly a key to the adoption of such a documentation system. There is clearly a need to see how this process can be achieved, leading to better documented programs.

Chapter 3

Research problem

We saw how documentation is handled today. There are tools like Javadoc or Doxygen that generate technical documentation with more or less customization options. The generated HTML files make it easy to navigate through the entities of the documented program. IDEs also provide features that help in the documentation process like Smalltalk's protocols or Eclipse's *todo* tags. On the other hand, methodologies focus on documentation like literate programming or by making code more understandable like example-centric programming and intentional programming.

In the previous chapter, we saw that all these concepts have some limitations. To improve documentation, a methodology should take care of the documentation in the development process of software. This methodology could be alleviated with a documentation system that helps the programmer in the phases of that methodology. To design that documentation system, it is important to identify the requirements of the system. We will see what a documentation system should ideally do. We will finally present a mockup that illustrates how the system could work in practice.

3.1 Software requirements

When we studied the existing documentation approaches, we identified several limitations that have been presented in the previous chapter. This consolidated the idea to improve documentation by proposing a methodology and a documentation system. We analyzed the advantages and limitations of each of the documentation approaches and identified several requirements:

- Support for any kind of documentation
- Support documentation classification
- One documentation entry - multiple entities

- Handle special tags
- Manipulate data uniformly
- Support documentation persistency
- Run code from examples
- Support multiple views
- Have a dynamic system

These requirements will now be explained in more detail an will help us understand what a documentation system should do. This will lead us to a proposal for a documentation system with illustrative mockups.

3.1.1 Support for any kind of documentation

A lively documentation system should not rely on comments only but rather support any kind of documentation, like images, links, test-runs and so on. This means that a piece of code could have one or several documentation informations attached to it. The system must be able to support these different types of documentation.

3.1.2 Support documentation classification

If the documentation system supports multiple kinds of documentation it should also be possible to classify documentation in relevant sections, like Smalltalk protocols. For example, a documentation string might reference methods of several classes that share the same concern. It should be possible to tag those classes to be part of a whole that works together. If the user of the system browses the code of one class, he should see that it belongs to a system of classes that collaborates together. This is for instance interesting when implementing a design pattern. Knowing which classes are involved in the pattern is very convenient to understand and maintain the system.

3.1.3 Map documentation and entities

It is possible that documentation refer to multiple entities at the same time. This might be the case when multiple classes work together and need one documentation entry to illustrate that fact. Whenever the user consult documentation for one entity, he should be advised that there is a global documentation that link that entity to others. This gives a more global view of relations between entities and also reduce documentation duplication.

3.1.4 Handle special tags

The documentation system aims to provide a full set of tools to improve the documentation process of a program. Some programmers find it convenient to mark their source code with special tags. With Eclipse, it is possible to add a “todo” comment. This comment will automatically be added to a list of tasks. Such documentation information is of great help for the programmer. The system should support such types of documentation. It should be easy to define new special comments and add views to manage or display those comments with a special treatment.

3.1.5 Manipulate data uniformly

Information can come from the program to document as well as from the documentation system. It should thus be possible to get information about the program in an abstract way. That information could be associated to the information of the documentation system in an uniform way. The goal is to ease manipulation of data coming from the program to document as well as from the documentation system for customizers that want to enhance the documentation system.

3.1.6 Support documentation persistency

Programmers tend to devote more time on their code than on their documentation and as times goes by, documentation might not correspond the actual code anymore. This problem is accentuated by the fact that programmers need to run another tool to generate the documentation.

A way to overcome these problems is to do versioning of the code and the corresponding documentation. If code changes, the system could warn the programmer that documentation might need an update after a code modification. This is a difficult problem as lots of possible situations can arise. Code can be added, modified, replaced, renamed, or deleted and the documentation that correspond to it need some treatment. The purpose of the system is not to automate all the process but help the programmer maintain consistency in his documentation. This is why the programmer should be warned when these changes occurs.

3.1.7 Run code from examples

When programming, it is often the case that tests are run to see the immediate result of a programmed feature. This is especially true for programs that are not well defined at start. These tests are intended to verify that the code is correct. Test-driven development [74] is a methodology where tests are written even before code is written to verify that they effectively fail. Sometimes, the programmers writes some tests only to convince himself that his code is correct but he does not save these tests. Storing these tests in the documentation tool could add some interest to the documentation.

- To begin with, these tests could serve as test-runs for the program. If the input and output of the tests are defined it is possible to check the results and see if the code succeeds the tests. This process could be automated, warning the programmer that some code does not pass the tests anymore. Extending this feature would allow the programmer to embed real unit tests for each added feature. The tool would then generate the appropriate code for a full unit test and run it.
- The other point is that test-runs usually show how to use the code with small representative examples. Examples could thus serve as documentation as well. The idea of using examples joins the ideas of example-centric programming 2.4.5.

3.1.8 Support multiple views

To be flexible for several documentation purposes and to merge them all under a unique tool, it is important to provide several views of the system. A programmer that is writing code usually will need a tool where he can easily see the parts that still need documentation. A tester might need to see only the code he need to test or even only the results, not even bothering about the code.

3.1.9 Have a dynamic system

Introspection gives the ability to a program to get information about itself. This is a very powerful feature of the programming language that the lively documentation system could take advantage of. With introspection, it is possible to get information about the classes and functions in a program or even the documentation string of entities of the program at runtime. This makes it possible to build a dynamic documentation system on top of these features. Relying on features that are already present in a programing language will allow us to build a documentation system prototype in less time.

The different requirements have been presented in detail and help to understand what a documentation system should do. To put images on words, a mockup will now be presented to see how the documentation system could handle the presented requirements.

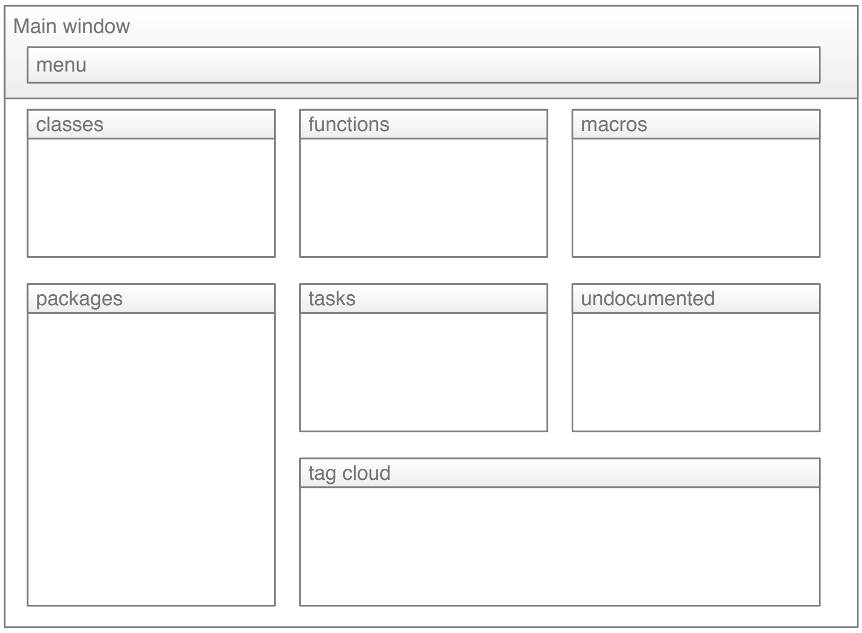


Figure 3.1: Main window of the documentation system

3.2 A first mockup

A first mockup illustrate how tasks could be realized by the documentation system. It is from that mockup that screens of the documentation system are finally built.

3.2.1 Main window

The main window displays general information about the documented program 3.1.

- A menu.
- Listings that show information of the important entities of the main package of the user (classes, functions, etc.).
- A complete listing of the available packages in the system.
- A listing with the elements the user tagged as “todo”.
- The last undocumented entities of the system.
- A tag-cloud to give a visual feedback of the system to the user.

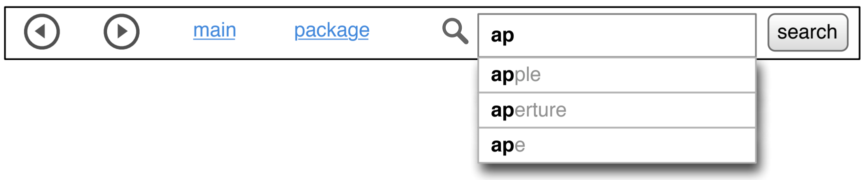


Figure 3.2: Menu of the documentation system

The window is separated in distinct boxes so that the user can choose where to place the boxes or add his own boxes with the information desired. Every box is a listing of elements that lead to other windows showing more specific information.

3.2.2 Menu

The menu 3.2 gives several fast access to information the user might need directly:

- Back and forward actions to see the next or the previous window.
- A button to return to the main window.
- A button that shows the content of the main package of the user.
- A search function for retrieving of information relative to a tagged entity. Ideally, a suggestion box could help the user.

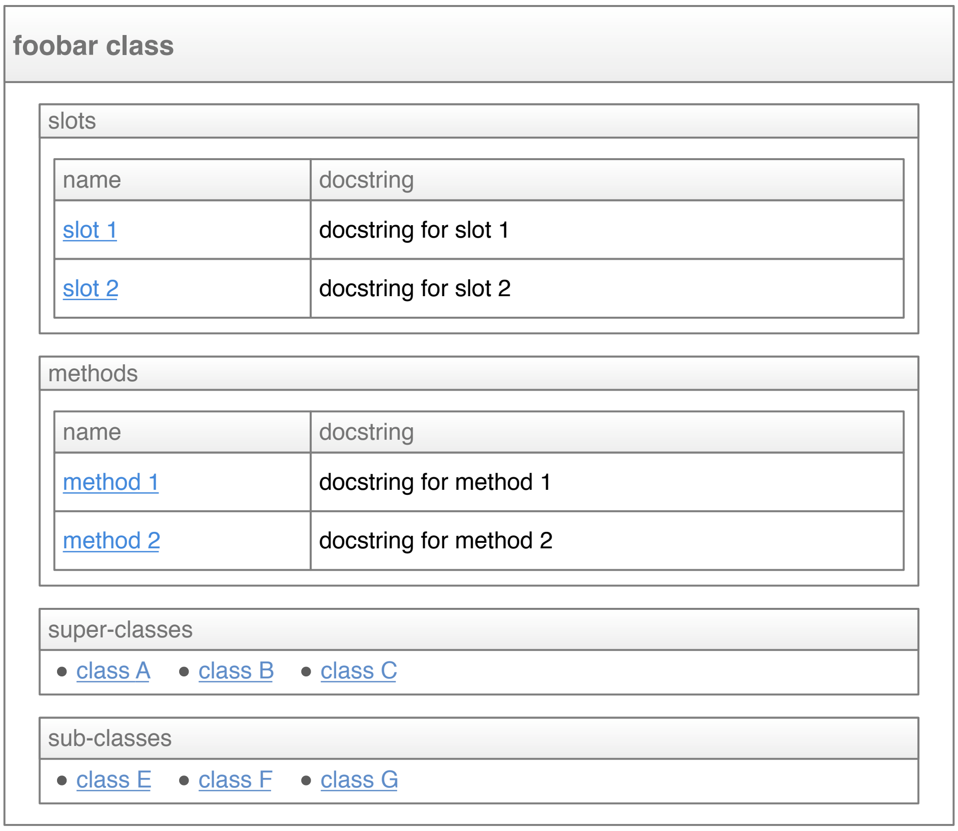
Other elements might be added in the menu later on, like user customization and so on.

3.2.3 Documentation browsing

All the entities of the system have almost similar information to display. The only exception is for classes that have fields and methods, superclasses and subclasses. An entity representation is separated in two parts.

General documentation view

The first part represented in figure 3.3 is the general view of an entity. It represent a table with the entities. A class has slots and methods, so there are two tables. Each table shows the name and the docstring of the entity. The name links to the documentation detail for that entity. Other useful information can also be displayed like super-classes and sub-classes in the case of a class.



foobar class

slots

name	docstring
slot 1	docstring for slot 1
slot 2	docstring for slot 2

methods

name	docstring
method 1	docstring for method 1
method 2	docstring for method 2

super-classes

- class A
- class B
- class C

sub-classes

- class E
- class F
- class G

Figure 3.3: Browsing documentation of a class (general view)

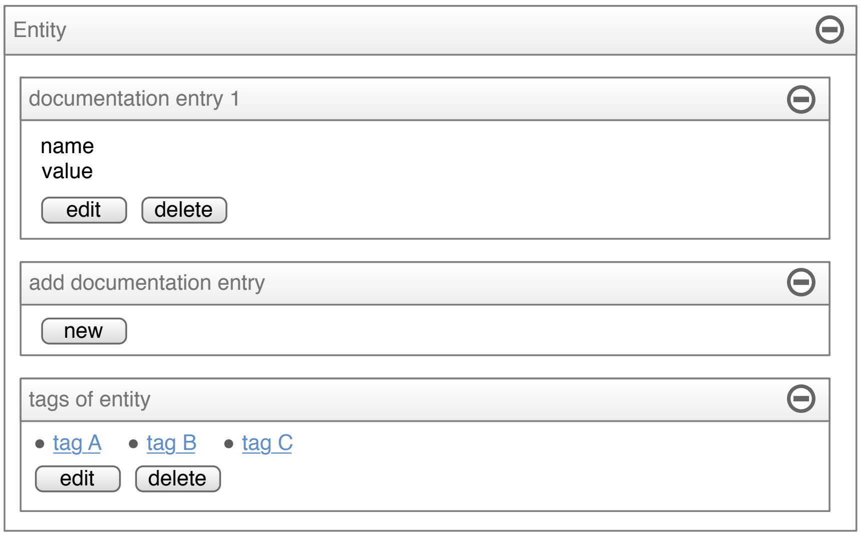


Figure 3.4: Browsing documentation of a class (documentation view)

Detailed documentation view

The second part 3.4 represent the detailed documentation view (i) where the user can see each documentation entry. Forach documentation entry, (ii) it is possible to edit it or delete it. If the user clicks on "delete" the corresponding entry is immediately deleted. If the user clicks on "edit", a simple documentation editor opens to allow editing of the documentation. (iii) A user can also add a new entry for the documentation of that entity by clicking on the "new" button. (iv) All the tags that where annotated are listed at the bottom of each documentation entity.

3.2.4 Documentation editing

When the user clicks on "edit" or "new" to update a documentation entry 3.4, (i) a simple editor opens to modify the entry 3.5. (ii) A name and a type can be chosen for the documentation entry. (iii) The user can then save his changes.

Advanced documentation editing

In the case documentation must be added to target multiple entities, (i) the user can check the global checkbox and he can then choose entities to which this documentation is related by clicking on "attach another entity". The attached entity can also be deleted by unchecking the associated checkbox. To attach media to the documentation the same principle is used. The user clicks "attach another file", he can then browse the system to chose the file.

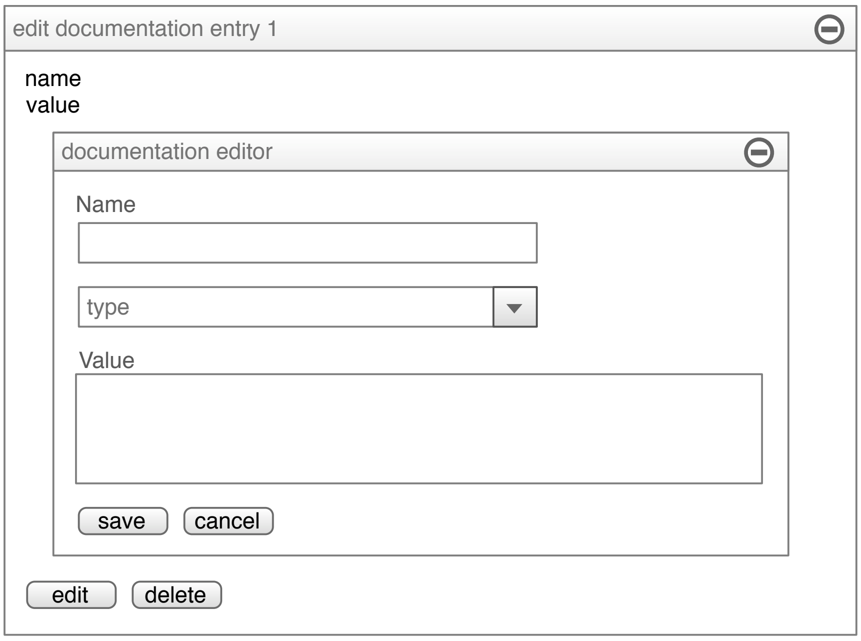


Figure 3.5: Documentation editing

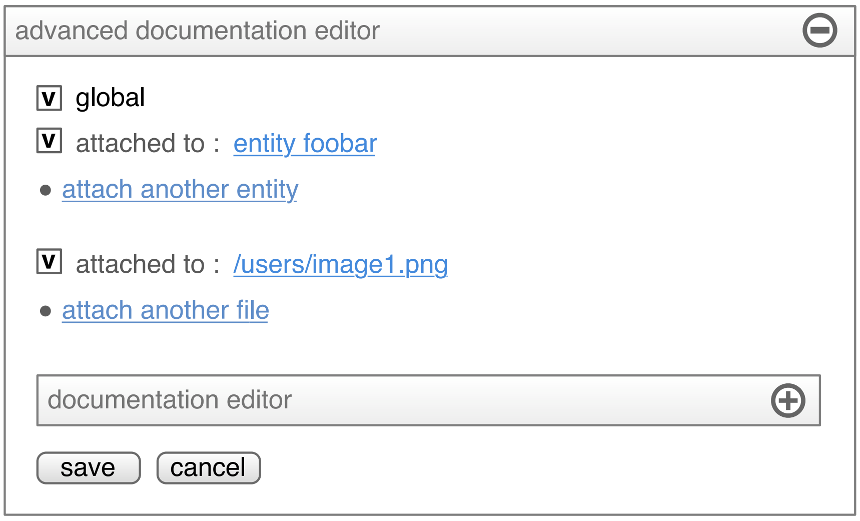

advanced documentation editor
V global
V attached to : entity foobar
attach another entity
V attached to : /users/image1.png
attach another file
documentation editor
save
cancel

Figure 3.6: Advanced documentation editing

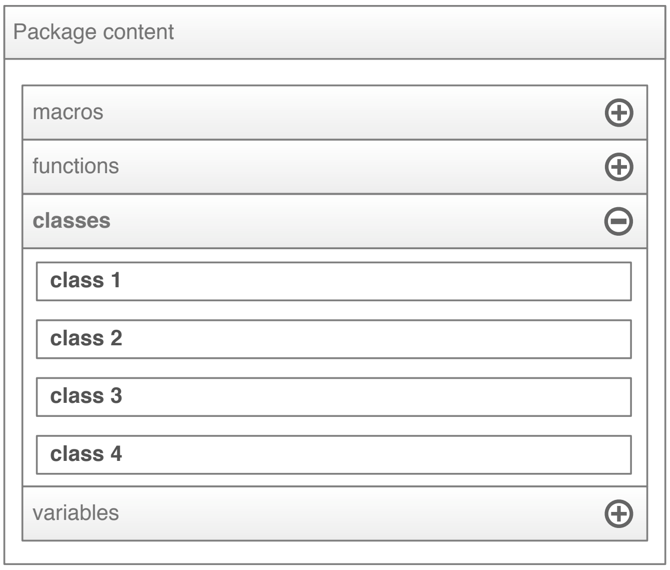


Figure 3.7: Accordion menu to show entities of the system

Once the file is selected, it can be removed by unchecking the associated checkbox.

3.2.5 Organized listings

As the screenshots 3.3 and 3.4 show, displaying all the information of a class takes some space. To display listing information in a more structured manner, it is for instance possible to use an accordion menu 3.7 to show the interesting entities of a package. When the user clicks on the “classes” menu, it opens to show all the classes of the package and the other menus close. These type of organized listings can be presented on the left or right side of the documentation of the entities.

3.3 Conclusion

We saw that current documentation systems have limitations and we proposed ideas to make a new documentation system that can overcome these limitations. The software requirements where summarized in a listing on the beginning of the chapter. Each requirement has then be presented. This led to the realization of a mockup of the system to show how important tasks of the system can be addressed. Of course, this mockup does not cover all the possibilities that will be presented. However, they give a first intuition on what must be done and guide to the next steps in the realization of the documentation system.

Chapter 4

System Use

Before entering in much detail on how to implement a documentation software. A presentation of the implemented software gives a precise vision of the goal of the system. This will introduce the notions used in the system and help to understand how the documentation system works. After having presented the implemented software, a methodology is proposed to include documentation in the process of software development. This will advance the reasons to implement a documentation software that helps programmers in the use of the methodology that will be proposed.

4.1 Documentation system

This section will present the general use of the system and how each use case presented is finally achieved using the implemented documentation system.

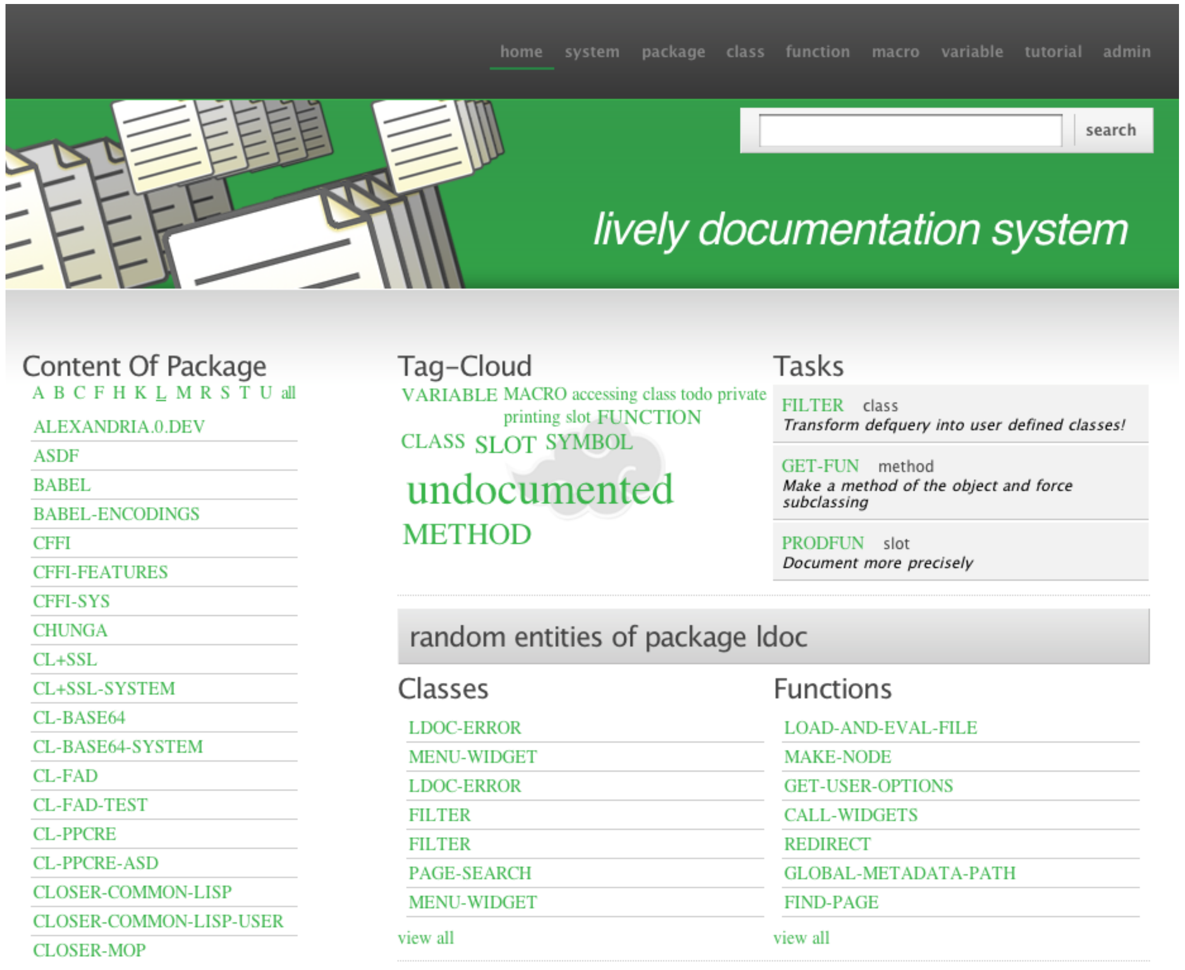
4.1.1 Browse documentation

The first and most important task the system has to execute is the display of documentation such that a user can navigate and easily retrieve information. Several features have been implemented to allow the retrieval of information in a natural way.

Home page

The screenshot 4.1 shows a part of the home page of the system. Each element will be presented with more detail.

- The menu is on the top of the page.
- Under the menu there is a search bar.
- On the left, the user can browse for the packages of the system.



Content Of Package

A B C F H K L M R S T U all

ALEXANDRIA.0.DEV
ASDF
BABEL
BABEL-ENCODINGS
CFFI
CFFI-FEATURES
CFFI-SYS
CHUNGA
CL+SSL
CL+SSL-SYSTEM
CL-BASE64
CL-BASE64-SYSTEM
CL-FAD
CL-FAD-TEST
CL-PPCRE
CL-PPCRE-ASD
CLOSER-COMMON-LISP
CLOSER-COMMON-LISP-USER
CLOSER-MOP

Tag–Cloud

VARIABLE MACRO accessing class todo private printing slot FUNCTION CLASS SLOT SYMBOL undocumented METHOD

Tasks

FILTER class
Transform defquery into user defined classes!

GET-FUN method
Make a method of the object and force subclassing

PRODFUN slot
Document more precisely

random entities of package ldoc

Classes

LDOC-ERROR
MENU-WIDGET
LDOC-ERROR
FILTER
FILTER
PAGE-SEARCH
MENU-WIDGET
view all

Functions

LOAD-AND-EVAL-FILE
MAKE-NODE
GET-USER-OPTIONS
CALL-WIDGETS
REDIRECT
GLOBAL-METADATA-PATH
FIND-PAGE
view all

Figure 4.1: Home page of the documentation system

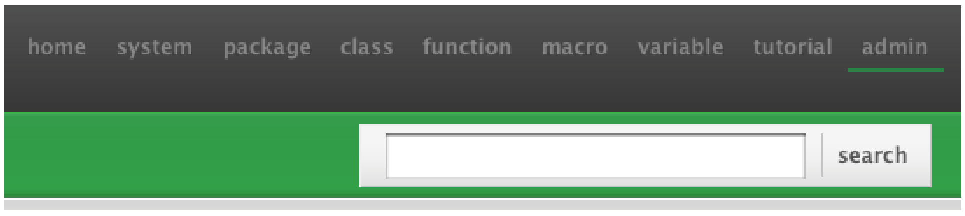


Figure 4.2: Menu of the documentation system

- In the middle, a tag-cloud present the tags of the system.
- On the right of the tag-cloud, the tasks of the user are presented.
- The rest of the page shows some random entities of the system.

As we can see, the home page does not only show information relative to documentation like entities listings but also other elements like a tag-cloud and a task list. These entities among with others can be inserted on any page to give further information to the user of the system. Each of this features is a component that plugs itself into the system. We will now present the elements of the home page individually.

Menu

The menu is made of two parts as shown in screenshot 4.2.

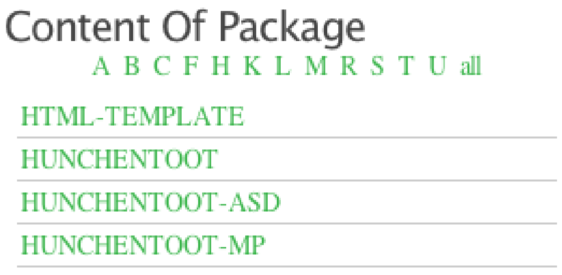
- A navigation bar. Each section of the system is directly accessible from the menu. The user can for instance browse entities of the system like packages, classes, functions, but also tutorials and go to the options page. The page navigated by the user is underlined in the menu.
- A search bar. The user can enter a tag to search for it in the search-bar. The results are presented on a new page (see. 4.7).

List packages

On the left side of the home page the user can choose among all the packages of the system. On the top of the menu, an alphabetical sorter shows only elements corresponding to the selected letter. In the screenshot 4.3, only the packages starting with letter H are displayed.

Tag-cloud

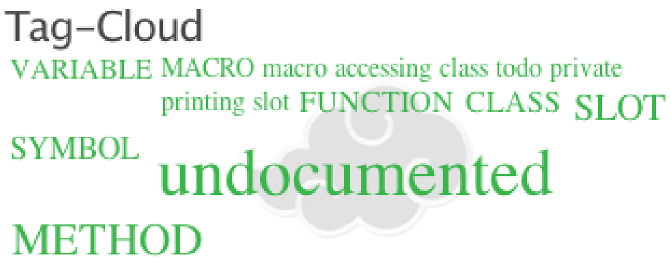
The tag cloud gives a visual feedback of the tags used in the system. The bigger the tag, the more it is used. In this screenshot 4.4, the tags that are


Content Of Package

A B C F H K L M R S T U all

HTML-TEMPLATE
HUNCHENTOOT
HUNCHENTOOT-ASD
HUNCHENTOOT-MP

Figure 4.3: Home page's package browser


Tag-Cloud

VARIABLE MACRO macro accessing class todo private printing slot FUNCTION CLASS SLOT SYMBOL undocumented METHOD

Figure 4.4: Home page's tag-cloud widget

used the most are "undocumented" and the existing types of the entities of the system. These are set automatically by the system.

Tasks

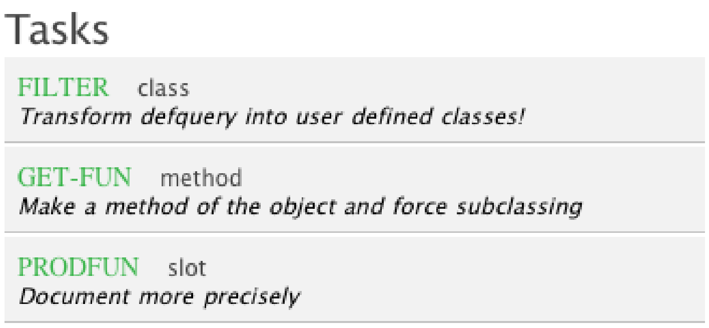
The tasks widget on screenshot 4.5 shows the elements that have been commented with a "todo" documentation entry. The entities are automatically tagged by the system as "todo". Each entity with that tag is then presented in the task widget together with its comment. This allows the user to annotate code and easily retrieve it.

Other information

The last element on the home page is a set of listings with some random entities of the documented program. The screenshot 4.6 shows seven classes and seven functions. The user can browse each of the listed entities or view all the elements for the type of that listing.

Search results

When the user searches for an entity in the search bar, the results are displayed on the search page presented in screenshot 4.7. In this example the

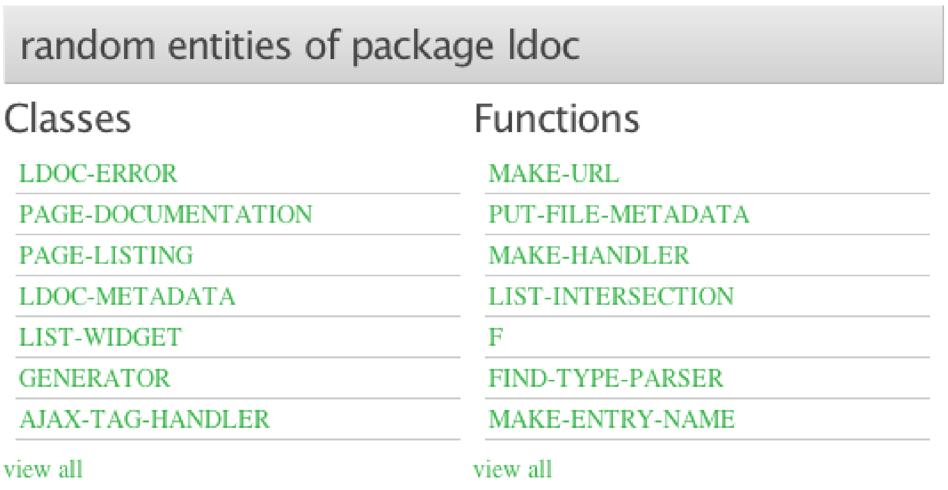

Tasks

FILTER class
Transform defquery into user defined classes!

GET-FUN method
Make a method of the object and force subclassing

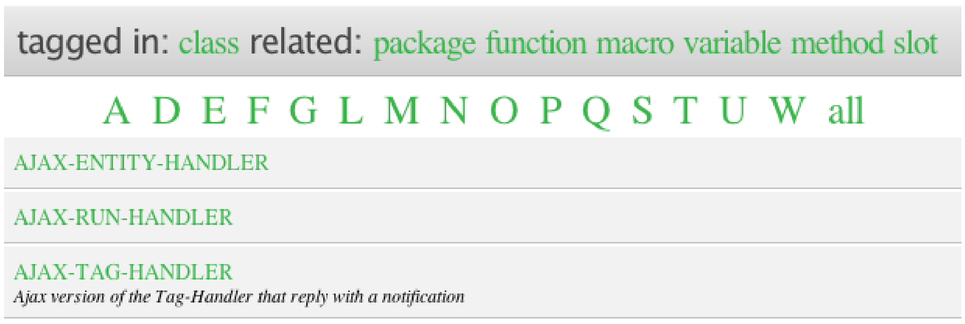
PRODFUN slot
Document more precisely

Figure 4.5: Home page's tasks widget


random entities of package ldoc

Classes	Functions
LDOC-ERROR	MAKE-URL
PAGE-DOCUMENTATION	PUT-FILE-METADATA
PAGE-LISTING	MAKE-HANDLER
LDOC-METADATA	LIST-INTERSECTION
LIST-WIDGET	F
GENERATOR	FIND-TYPE-PARSER
AJAX-TAG-HANDLER	MAKE-ENTRY-NAME
view all	view all

Figure 4.6: Other information displayed on the home page


tagged in: class related: package function macro variable method slot

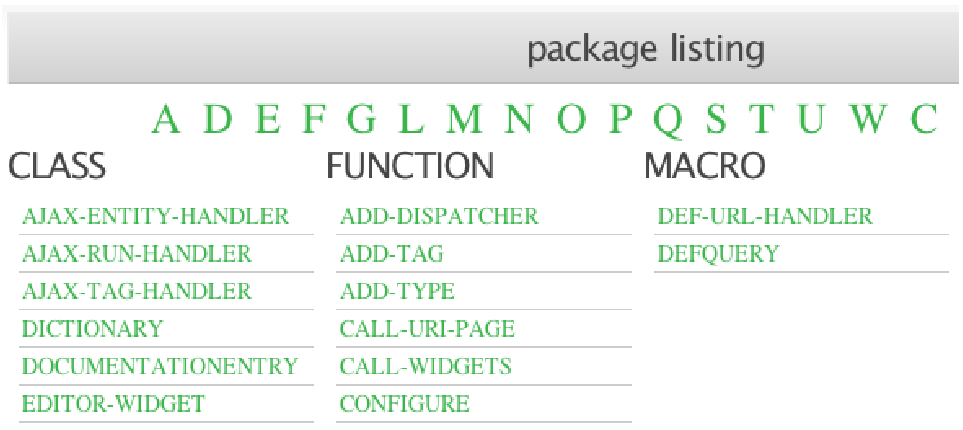
A D E F G L M N O P Q S T U W all

AJAX-ENTITY-HANDLER

AJAX-RUN-HANDLER

AJAX-TAG-HANDLER
Ajax version of the Tag-Handler that reply with a notification

Figure 4.7: Search results page of the system


package listing

A D E F G L M N O P Q S T U W C

CLASS	FUNCTION	MACRO
AJAX-ENTITY-HANDLER	ADD-DISPATCHER	DEF-URL-HANDLER
AJAX-RUN-HANDLER	ADD-TAG	DEFQUERY
AJAX-TAG-HANDLER	ADD-TYPE	
DICTIONARY	CALL-URI-PAGE	
DOCUMENTATIONENTRY	CALL-WIDGETS	
EDITOR-WIDGET	CONFIGURE	

Figure 4.8: Package content page of the system

user searched for all entities tagged as "class". The results are alphabetically sorted and the screenshot shows entities sorted on letter A only. For each entity, the docstring is displayed as well if there is one. In the title, the user sees the tag he searched for but also the related tags. In this examples, tags related to "class" are all the other types of the documentation system, like package, function, etc.

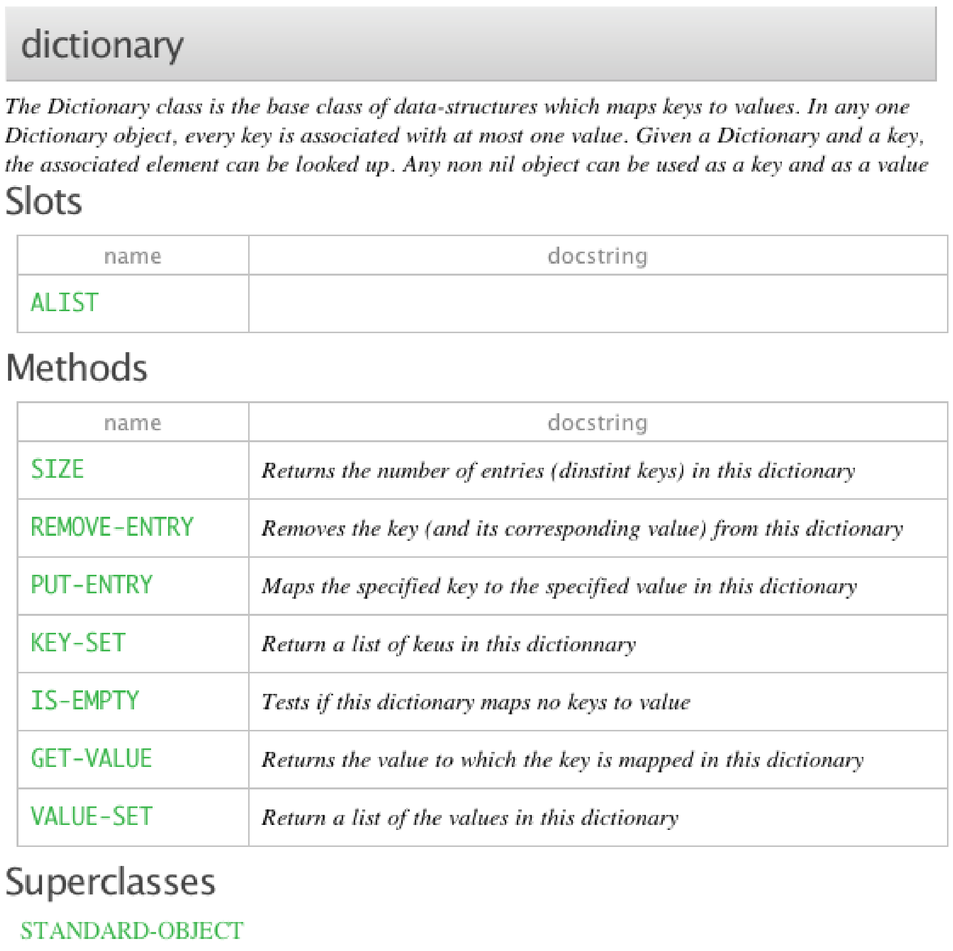
Browse package content

The package content page basically represents each type of entity in a listing. As we can see in screenshot 4.8, an alphabetical sorter allows the list to be filtered in real time thanks to Javascipt. It is thus possible to retrieve all the entities that starts with letter "C" by clicking on the sorter. No new page need to be loaded. The entities are classified in distinct columns depending on their type. The screenshot only shows 3 columns of the system for readability.

Browse class content

A class entity is represented on a page that is subdivided in two parts. In the first part, tables shows the slots and the methods of the class together with their docstring as shown in figure 4.9. The docstring of the class is also represented. Beneath that, other information relative to a class is also available, like the subclasses and superclasses of the class. This part gives a more general view of the entities that compose a class and the general information that can be retrieved from the system.

In the second part, each entity is represented in a box with every documentation entry. Each box has the name of the entity followed by the tags



dictionary

The Dictionary class is the base class of data-structures which maps keys to values. In any one Dictionary object, every key is associated with at most one value. Given a Dictionary and a key, the associated element can be looked up. Any non nil object can be used as a key and as a value

Slots

name	docstring
ALIST	

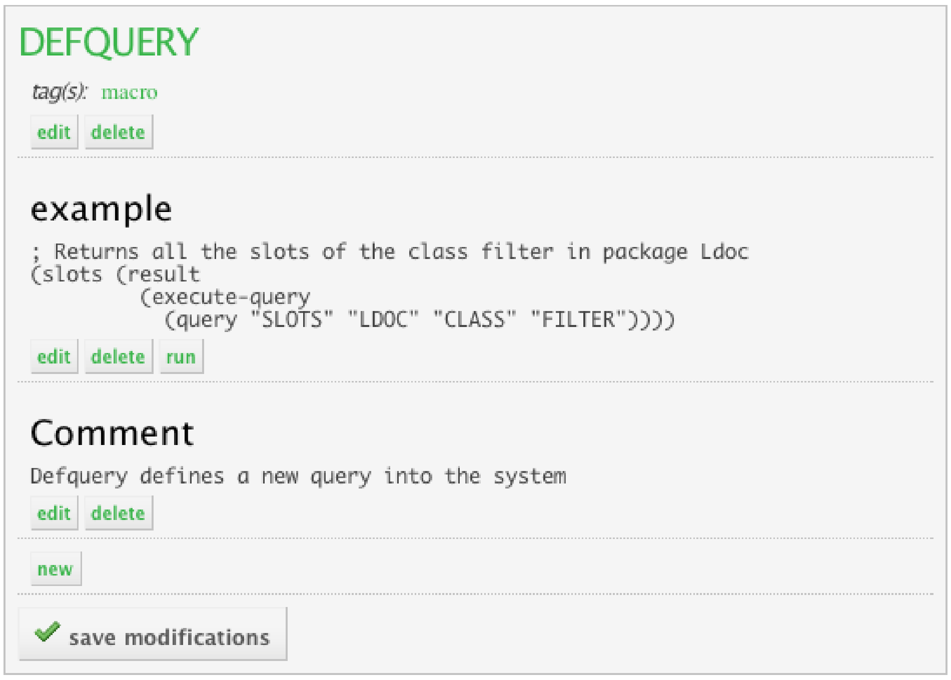
Methods

name	docstring
SIZE	*Returns the number of entries (dinstint keys) in this dictionary*
REMOVE-ENTRY	*Removes the key (and its corresponding value) from this dictionary*
PUT-ENTRY	*Maps the specified key to the specified value in this dictionary*
KEY-SET	*Return a list of keus in this dictionnary*
IS-EMPTY	*Tests if this dictionary maps no keys to value*
GET-VALUE	*Returns the value to which the key is mapped in this dictionary*
VALUE-SET	*Return a list of the values in this dictionary*

Superclasses

STANDARD-OBJECT

Figure 4.9: General view of a class of the system



example

```
; Returns all the slots of the class filter in package Ldoc
(slots (result
         (execute-query
           (query "SLOTS" "LDOC" "CLASS" "FILTER"))))
```

Comment

Defquery defines a new query into the system

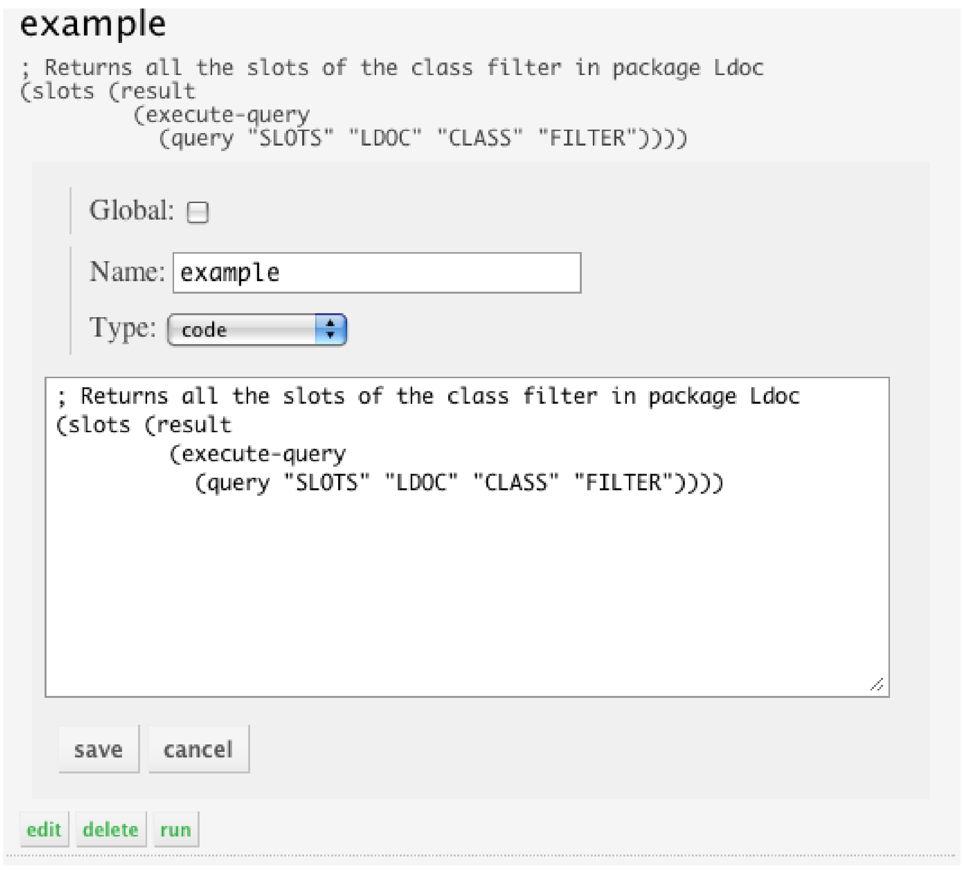
Figure 4.10: Documentation view of an entity

of the entity. Then, for each documentation entry, the name of the entry and its data is represented.

Other entities like functions and macros are represented the same way as classes. The only difference is that all the entities are represented on the same page and that they are not subdivided with slots and methods like it is the case for classes.

4.1.2 Edit documentation

From the documentation entry of an entity represented in the screenshot 4.10 it is possible to directly edit the documentation as presented in screenshot 4.11. When the documenter clicks on the edit link, an editor shows up to edit documentation of an entity. There is a text-field for the name of the documentation entry, a drop-down menu for the type and a text-area for the data. Changes can be saved or canceled with the appropriate buttons. The *edit* button toggles the editor and the *delete* button removes the documentation entry. The changes can be submitted to the server by clicking on *save modifications*. A notification warns the documenter of the data submission.


example

```
; Returns all the slots of the class filter in package Ldoc
(slots (result
         (execute-query
           (query "SLOTS" "LDOC" "CLASS" "FILTER"))))
```

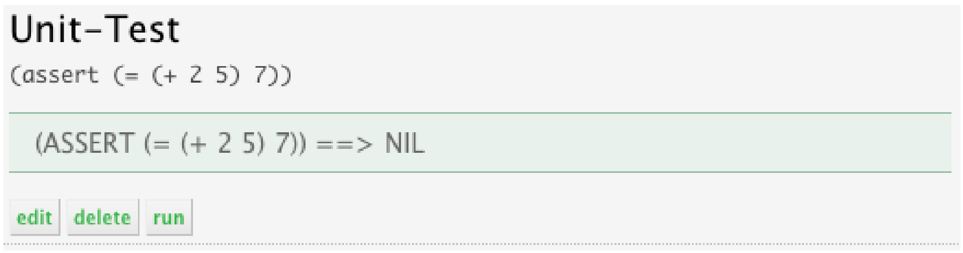
Global: ☐

Name: example

Type: code

```
; Returns all the slots of the class filter in package Ldoc
(slots (result
         (execute-query
           (query "SLOTS" "LDOC" "CLASS" "FILTER"))))
```

Figure 4.11: Edition of a documentation entry


Unit-Test

```
(assert (= (+ 2 5) 7))
```

```
(ASSERT (= (+ 2 5) 7)) ==> NIL
```

Figure 4.12: Run code from examples

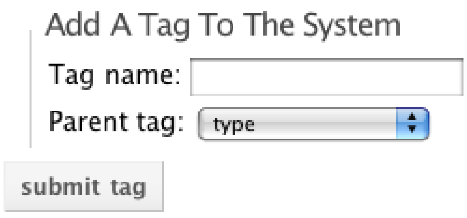


Figure 4.13: Adding a new tag to the system

4.1.3 Run code from examples

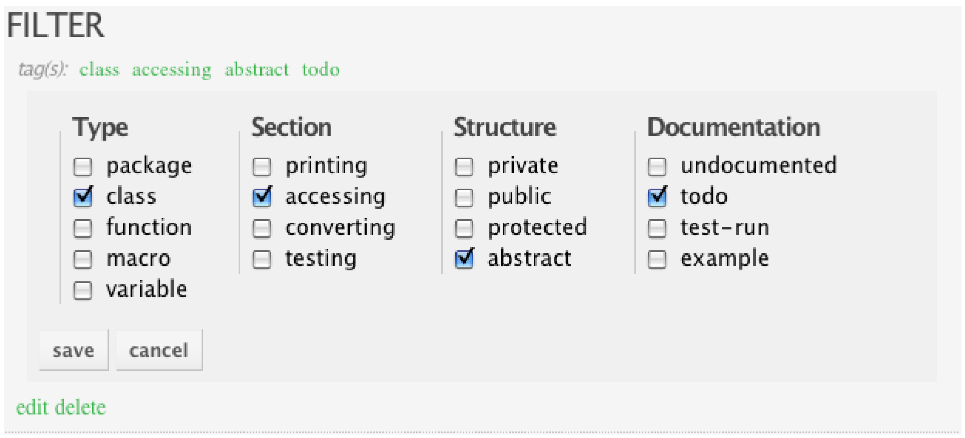
Screenshot 4.12 shows how to run code from examples. The type of the documentation entry is set to *code*. A new button appears to run the code. If the button is activated, the source code is sent to the server that parses it and evaluates it. The result is returned to the user with a notification. In this example the assertion returns `nil` to indicate the code is correct. If the code is incorrect, an error is returned. It is possible to execute examples as well as unit tests.

4.1.4 Handle tags

To mark an entity with a tag, the tag must first be added to the system. Then it is available in the tag editor.

Add tag

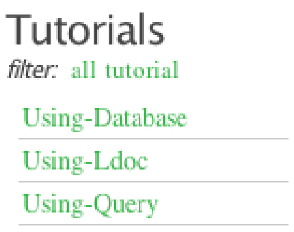
The customizer (see 5.2) can add new tags to the system 4.13. This is done on the option page in the section reserved to the tags. To add a tag, a drop-down menu indicates the parent tag of the new tag and a text-field indicates the name for the tag to create.


FILTER

tag(s): class accessing abstract todo

Type	Section	Structure	Documentation
☐ package	☐ printing	☐ private	☐ undocumented
✓ class	✓ accessing	☐ public	✓ todo
☐ function	☐ converting	☐ protected	☐ test-run
☐ macro	☐ testing	✓ abstract	☐ example
☐ variable			

Figure 4.14: Choose which tags to apply for an entity


Tutorials

filter: all tutorial

Using-Database
Using-Ldoc
Using-Query

Figure 4.15: Tutorial listing

Edit tags

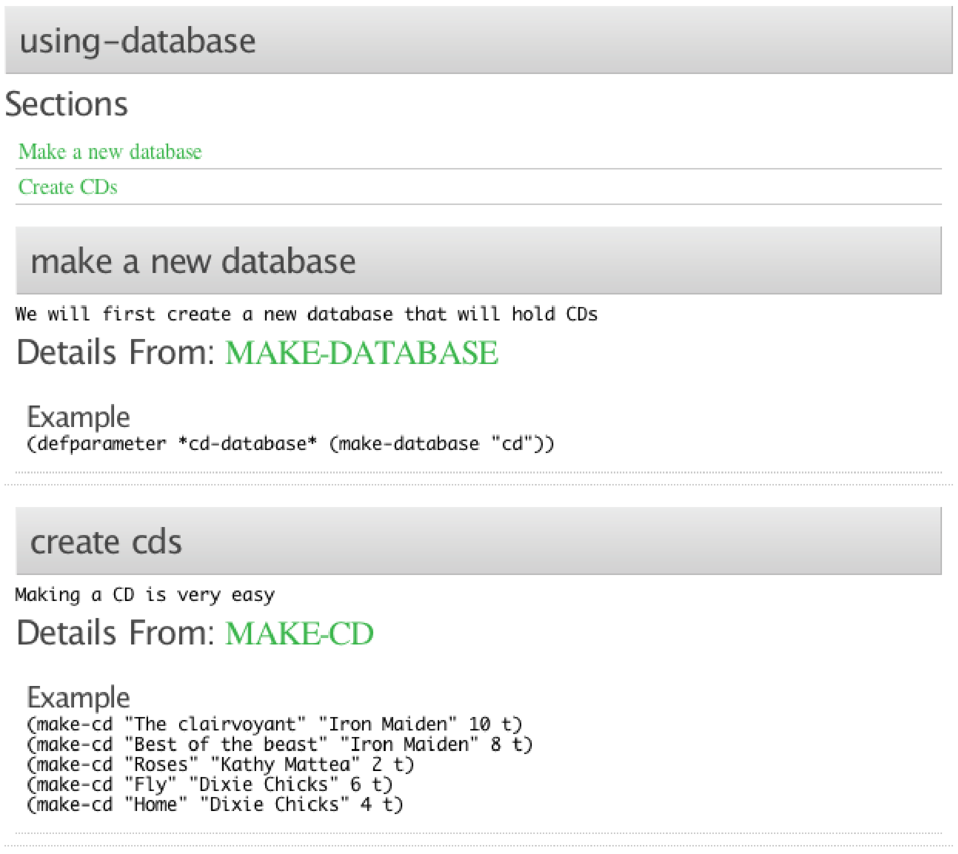
Screenshot 4.14 shows the opened tag editor. All the tags of the system are represented in distinct sections and the user can choose to tag or untag the entity by checking each desired tag. The example shows that the entity is tagged in `class`, `accessing`, `abstract` and `todo`. Tags `class` and `todo` have been added automatically by the system.

4.1.5 Handle tutorials

To read tutorials, it must be possible to add them to the system. We will first show how they are presented and then how to add a new tutorial in the system.

Read tutorial

The tutorial page is divided in two parts. On the left, represented in screenshot 4.15, all the available tutorials are represented. Clicking on one of them


using-database

Sections

Make a new database

Create CDs

make a new database

We will first create a new database that will hold CDs

Details From: MAKE-DATABASE

Example

```
(defparameter *cd-database* (make-database "cd"))
```

create cds

Making a CD is very easy

Details From: MAKE-CD

Example

```
(make-cd "The clairvoyant" "Iron Maiden" 10 t)
(make-cd "Best of the beast" "Iron Maiden" 8 t)
(make-cd "Roses" "Kathy Mattea" 2 t)
(make-cd "Fly" "Dixie Chicks" 6 t)
(make-cd "Home" "Dixie Chicks" 4 t)
```

Figure 4.16: Tutorial

shows the full tutorial on the right.

The screenshot 4.16 represents a full tutorial. On the top, the different sections are summarized in a listing. Each section is then presented with a title, the comment, a reference to the original entity from which the documentation comes and finally the documentation of that entity.

Add tutorial

The screenshot 4.17 shows how it is possible to add new tutorials to the system. A tutorial is made of a name and of several sections. The name identifies the tutorial. Each section has a name too, then a reference to the desired entity is required (It must be added manually). Finally, a comment can be added to explain something more on the documentation entry that is referenced. The button *new* adds a form to add one more section to the tutorial. When edition is finished, button *save tutorial* submits the data to

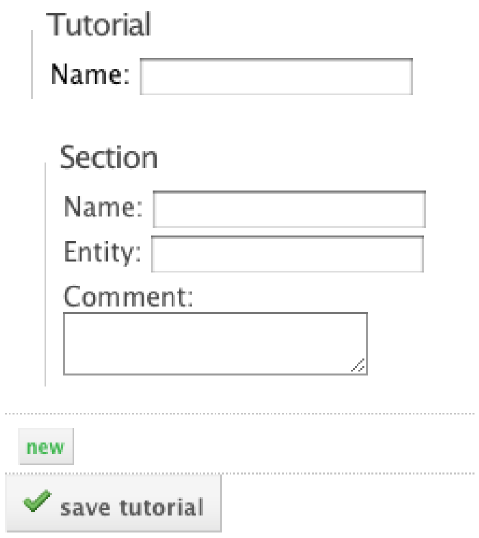


Figure 4.17: Add a tutorial to the system

the server to be saved. It is then possible to navigate to the tutorial page to consult the tutorial.

4.1.6 Web browser operations

Because the documentation system is a web application, some features are provided by the web browser and do not need to be implemented by the system.

- It is possible to navigate back and forward through the different pages of the system.
- It is possible to bookmark any page to return to it later on.
- It is possible to navigate to any entity by specifying its name in the address bar of the web browser. Names have been taken into consideration to use the possibilities of the address bar of the web browser.

To keep these operations, it has been decided to add asynchronous requests only to improve the edition of the documentation. Classical navigation is preserved and every page has a navigable address. If all the HTML documents are loaded with Ajax in the same page, it is not possible to bookmark a given page or to use the buttons to navigate back and forward. Asynchronous requests are thus used in an unobtrusive way to support the web browser operations.

4.2 Documentation framework

We saw the operations that users can do with the documentation system, but the system is also conceived to support extensions and to support addition of new documentation features when they are discovered from the use of the documentation system. To illustrate this possibility, we will now see how to extend the documentation system with a new feature.

4.2.1 Define queries

Defining a query is the main tool of the customizer to describe the data that has to be represented and to manipulate it.

Listing 4.1: Defining a new query

```
(defquery :tag-cloud :full-package-content (list)
  (let ( (counts (make-dictionary)) )
    (loop for entity in list
      do (loop for tag in (tags (metadata entity))
           do (if (has-key counts tag)
                  (add-entry counts tag
                    (+ (get-value counts tag) 1))
                  (add-entry counts tag 1))))
    counts))
```

New queries are added in the system with the macro `defquery`. The query will be named `tag-cloud` and will receive its input as a list from a parent `full-package-content`. The last argument of the macro is the definition of our query. This is the part where the code that will actually count the tags is implemented. To count tag occurrences, we will use a dictionary in which each key will be a tag name and the value is the occurrences of that tag. To populate that dictionary we need to loop over all the entities received as argument, then loop over each tag of the entity. If a tag already exist with that name in our dictionary, we can increment the tag counter, otherwise, a new entry must be added into the dictionary. The entry has the tag name as key and 1 as value because the tag appeared for the first time. Once every entry is has been treated, the dictionary can be returned.

4.2.2 Define views

With the query defined before, we will build a reusable widget that displays a tag-cloud.

Define a template

Widgets uses HTML code to be displayed and need a template file presented in the next example:

Listing 4.2: The html-template for the tag-cloud

```
<!-- TMPL_IF title -->
<h3><!-- TMPL_VAR title --></h3>
<!-- /TMPL_IF -->
<div class="tag">
    <ul><!-- TMPL_LOOP tags -->
        <li>
            <a><!-- TMPL_VAR tag --></a>
            <span><!-- TMPL_VAR count --></span>
        </li><!-- /TMPL_LOOP -->
    </ul>
</div>
```

A tag-cloud has a title, and an unordered list of tags with their occurrences. In this example we use the HTML-TEMPLATE [23] syntax to build a template .

Implement the widget

In the system, a widget needs a path to the template file and a name that will be used by the system to retrieve it. Two slots will hold the title of the tag-cloud template and all the tags. A widget always inherit from the base class `Widget`. Here is the definition of the class:

```
(defclass Tag-Count-Widget(Widget)
  ((template
      :initform #P"widget/my-tag-cloud.html")
   (name
      :initform "cloud")
   (title
      :initform nil
      :initarg :title
      :accessor title)
   (tags
      :initform nil)))
```

Some methods are also needed to add items to the widget and to update the widget. The method to add an item will receive the tag name and the count for that tag. The two elements will then be added to the `tags` slot of the widget in a list of form `(list :tag tag :count count)`. The keywords of the list maps with the template definition and are used by HTML-TEMPLATE to fill the template:

```
(defmethod add-item ((widget Tag-Count-Widget) name count)
  (with-slots (tags) widget
    (setf tags (add tags (list :tag name :count count))))
  (update-widget widget))
```

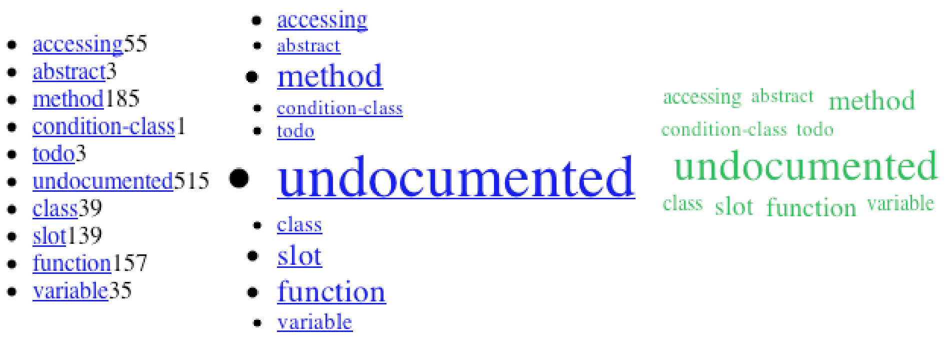


Figure 4.18: Evolution of a tag-cloud widget. (a) simple html. (b) with Javascript. (c) with Javascript and CSS

Then the widget updates itself by setting the `model` slot inherited by `Widget`. Here the keywords for the list are also important because they must correspond to the template defined earlier:

```
(defmethod update-widget ((widget Tag-Count-Widget))
  (with-slots (model title tags) widget
    (setf model (list :title title :tags tags))))
```

This gives an object that is reusable in the system. Once the widget is defined, the programmer do not need to bother with HTML and can rely on the features of the widget to update it and display it.

Create a page

To show the widget, a `page` must also be created. The page name is `tag-cloud` and the template is defined in a file `count.html`.

Listing 4.3: Page that displays the tag-count widget

```
(defclass Page-Tag-Count(Page)())

(defmethod update-model ((page Page-Tag-Count) uri)
  (let* ( (data (result
                (execute-query
                 (query :tag-cloud "LDOC"))))
          (widget (make-instance 'Tag-Cloud-Widget
                                :title "Tag Count")) )
    (loop for tag in (key-set data)
       do (add-item widget tag (get-value data tag)))
    (with-slots (model) page
     (setf model (call-widget widget :count)))))
```

The only method to implement is `update-model`. This method will use the previously presented query and widget. Then, for each tag and count in the result of the query, an item is added to the widget. In the previous example, the query is executed for the package “LDOC”.

Listing 4.4: The tag-count page is added to the system

```
(make-page 'Page-Tag-Count :tag-cloud #P"doc/count.html")
```

Finally, the programmer needs to call the function `make-page` to indicate to the system the name of the page and where to find the template for that page.

All the pieces are available to provide the system with a new feature that gives information about the occurrences of tags in the system for a given package. The programmer can now navigate to the `tag-cloud` page in his browser to see the result represented in figure 4.18(a).

The next steps will illustrate advantages of Javascript and CSS to improve the design of the tag-cloud.

Make a jQuery plugin

To illustrate how to make a reusable jQuery plugin, we will now implement a plugin for the tag-cloud widget.

```
$.fn.tagcloud = function(options) {
    return this.each(function() {
        // Plugin code here
    });
};
```

To start with the implementation of a plugin, a function `tagcloud` must be added to extend jQuery. This code is put in a `tagcloud.jquery.js` file. The next step is to implement the tag cloud algorithm.

```
// (i) Loop over each item
items.each( function(i) {
    // (ii) Get the count occurence from the span element
    count = parseInt( $(this).find('span').text() );
    // (iii) Count minimal and maximal
    max = (count > max ? count : max);
    min = (min > count ? count : min);
});
```

The function operates on HTML unordered list items (`ul li`). What we want is to express the size of each item in a percentage in a range defined by `minPercent` and `maxPercent`. To do this (i) we loop a first time over all items of the list, (ii) and count the value in the `span` element of the list item. Then (iii) The maximal and minimal value for count are updated.

```
// (i) Count the multiplier
var multiplier = (maxPercent-minPercent)/(max-min);
```

```
// (ii) Loop over each item again
items.each( function(i) {
    span = $(this).find('span');
    count = parseInt( span.text() );
    // (iii) count the size to apply for each element
    size = minPercent + ((max-(max-(count-min)))*multiplier)+'%';
    // (iv) apply the CSS rule
    $(this).css('font-size', size);
    // (v) remove the unnecessary span element
    span.remove();
});
```

Now (i) it is possible to count the multiplier that will give a size that depends on the number of occurrences of each tag. Finally, (ii) a second loop over all the elements will count the size (iii), (iv) apply it as a CSS rule to the item and (v) remove the span as it is not necessary anymore.

```
$(document).ready( function () {
    $("div.tag"). tagcloud();
});
```

In the main HTML file, the plugin must be called on the `div.tag` element. Each elements that match the `div.tag` selector will be applied the `tagcloud` function. The result is presented in figure 4.18 (b).

Improve with some CSS

This example does not resemble to a tag cloud yet. A final touch can be brought with some CSS.

```
/* (i) remove bullets of the unordered list */
ul { list-style-type: none; }

/* (ii) let the list items float */
li { float:left; }

/* (iii) add some decoration to the anchors */
a {
    padding-left: 0.5em;
    text-decoration: none;
    font-family: serif;
    color: #39B54A;
}
```

(i) The unordered list must not have bullets, (ii) The list items must float instead of being displayed one after the other. If there is not enough place for the item to fit in the container box, the item will be displayed on the next line. Because the sizes of the items are not uniform, this will look like the elements are put in a cloud. (iii) Finally enhance the look of the

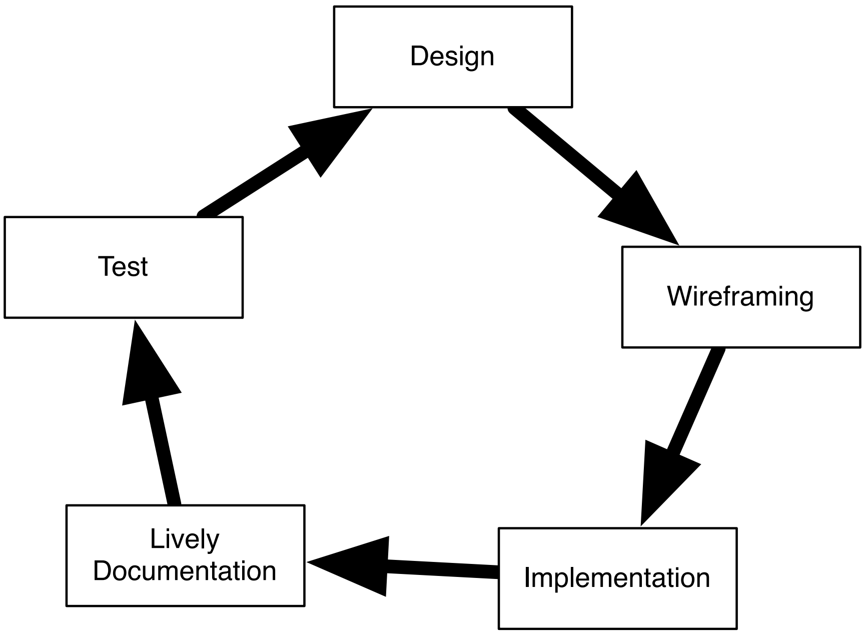


Figure 4.19: Methodology workflow

links with a color, a special font and put some padding so that each element has some space to be displayed. The final result is represented in figure 4.18 (c).

4.3 Methodology

The implementation of a tool alone will not provide better documentation. It is also important to provide a methodology that describes when and how to use the system. This methodology will help the programmer to integrate the documentation system in the development process.

The methodology is an Agile programming methodology [9] and suits well programs that need fast development with functionalities that can be implemented in separate modules. The base principle relies on five phases in which each module passes, possibly more than once.

- The **design** phase is the phase in which the global constraints of the module as well as the detailed design like design patterns or programming idioms are analyzed.
- The **wireframing** phase happens before code is written. In this phase, the programmer documents the main behaviors of the system he will

implement. For this, the programmer describe the functions, classes and methods that will be involved in the system as well as their documentation. The entity does not need to be implemented yet, only the declaration of the behavior of the entity is done. A typical example is the writing of the skeleton of a class for a data structure. In Java, this would be done using interfaces. The main tasks that the data structure will provide are described with a declaration and a documentation of the behavior of each task together with the input and output parameters or preconditions and postconditions. In Java, an interface will typically been written for this purpose. The behavior is defined and documentation is already written but no implementation is done at this step. To guarantee that the implementation will work correctly, tests must already been written in this phase. They will serve as documentation for the implementation and show how it should behave.

- The **implementation** phase is the moment in which the programmer will implement the features described in the previous phase. In this phase helper methods will be added next to the described methods of the interface. Choices will also be done and might need an explanation for better understanding. This is also the phase where the tester writes test-units. This step is strongly coupled with the next one.

- The **lively documentation** phase happens together with the implementation phase. Whenever helpers methods are added or some behavior need illustration or explanation, the programmer add documentation to those pieces of code. Annotations and anything else that helps organizing or understanding the code is added in this phase.

- Once code has been written and documented, it is possible to finally test it. In the **test** phase, the tests elaborated in the wireframing phase are run to see if the code actually does what is expected. This phase will assure that the implementation is correct and that the program behaves correctly. If the program does not satisfy the tests, the program will need a second step of implementation and eventually documentation. Once all the tests are correct, a new iteration of the process can be started.

These five phases explain a general approach of the steps to follow when a program must be realized. The methodology does not state at which level these phases must be applied. The level of granularity of the methodology depends on the people that uses the methodology. It is for instance possible to realize a whole program in one iteration. This is a very high level of abstraction and need a lot of insight and experience for the system to work well after only those five phases. Experience shows that for bigger projects, it is usually not possible or not a good idea. However, the programmer at home might easily accommodate with this level of abstraction for a personal project. On the other hand, a programmer can use the methodology and apply it to any class he is adding in the system. This need a lot more work but will guarantee that any piece of the system is fully designed, programmed, documented and tested. It is possible that not all classes in a system need that level of precision and someone might choose to use the methodology at an intermediate level, where a feature of the system is designed and finally tested. A feature being a plugin of the whole system or a group of elements executing one specific task. The methodology is thus flexible enough to let the programmer choose at which level to use it.

The purpose of the lively documentation system is to help the programmer in the three last phases of the methodology. In general, design phases will need some analysis of the desired program discussion with the client to respond to his requirements. For the documentation phase, it is recommended to use any IDE that the programmer is normally using. The programmer can then write the documentation and the interfaces in his language. These interfaces can then be submitted to the system. This is where the lively documentation system comes into play. Once the base entities are documented, the system can access the documentation information and display the entities which can then be browsed and edited by the users of the system. In the next phases switching between the IDE and the documentation system will allow the users to fully use the dynamic documentation system. This is where the methodology can benefit the more of the use of such a system. As we can see, the purpose of the documentation system is not to suppress the need for an IDE nor to make it an IDE but to use it in combination with the IDE to help the programmers in the application of the methodology.

4.4 Conclusion

The documentation system has been presented at work. The documentation features of the system demonstrate how to document entities of a program but also how to extend the system with new features. Finally, we saw how a methodology to deal with documentation and presented how to use the documentation system in that methodology.

Chapter 5

Implementation

The documentation system has been presented but the way it works and the technologies that it uses have not. It was necessary to show the system at work to clearly understand what it was all about before to dive into more technical concepts. Now that the documentation system is introduced, we will see how it is implemented. We will first focus on the technologies that where used to implement the system as they drove the system to a specific architecture. The architecture of the system and the tasks that where analyzed are then presented. Finally, some more technical details explains how the system works and some important implementation details that are worth mentioning to fully grasp all the concepts of the documentation system.

5.1 Technologies

The first step to explain the implementation of the system is to explain the technologies that have been used to implement it. These choices have advantages and limitations as well as alternative solutions that will be discussed.

5.1.1 Lisp

Lisp [17] is one of the earliest programming language that was invented and that is still used today. It supports both imperative and functional programming. Lisp became the choice of predilection for artificial intelligence but today, it is used in other domains as well, like web development, games, banking and so on.

Example

Lisp programs are easily recognizable with their parenthesize representation. This representation is used both for data and source code. The main data structure of Lisp is a linked-list and Lisp takes it name from “list processing”.

```
CL-USER > (+ 2 3 4)
9
```

The first example shows a Lisp expression. Each expression is delimited by parentheses and takes a number of elements. The first element is the function to apply to the other elements. Each element is separated by a space. In the examples the function "+" is applied to three numbers and the result is 9.

```
CL-USER > (list 2 3 4)
(1 2 3)
```

Te second expression shows a list with 3 elements. As we can see, the structure is similar to the first expression we saw just before.

```
CL-USER > (defun factorial (n &optional (acc 1))
              (if (<= n 1)
                  acc
                  (factorial (- n 1) (* acc n))))
FACTORIAL
```

A recursive implementation of the factorial function is represented in the next example. The function can take an optional accumulator value that is set to 1 if no value is provided. Lisp can take advantage of terminal recursion. It is now possible to call the function as follows:

```
CL-USER > (factorial 6)
720
```

Motivation

Common Lisp has been chosen for its simple notation. We used SBCL [18], an open source compiler and runtime system for ANSI Common Lisp. Lisp is dynamically typed and has a meta-object protocol [20] [21] and reflection facilities. The docstring is accessible dynamically, which was an interesting desired feature for the system. These properties altogether make it a good choice to implement a lively documentation system. These are however not the main reasons why Lisp has been chosen. Another reason is to have a dynamic documentation for context oriented programming language AmOS [5] [6] programmed in Lisp.

Alternative choices

Other languages also propose dynamic typing, a good meta-object protocol and reflection facilities. It is for instance the case of Python [29] and Ruby [28]. On the other hand, AmOS has been programmed in Lisp. This is why Lisp has finally been chosen among the possible programming languages.

5.1.2 A web-based approach

The choice for Lisp drove the lively documentation toward a web-based application for several reasons. Hunchentooth [22], a web server programmed in Lisp enables it to have a strong support for the bases of our documentation system. Lots of libraries also help to generate HTML syntax. HTML-TEMPLATE [23] is one of them and allows to build templates of the pages that will serve to the application.

Because of the choice for a web-application, it became obvious to develop an Ajax application. The next section will introduce Ajax and the other technological choices.

5.1.3 Ajax

Ajax for Asynchronous JavaScript and XML [66] is the combination of multiple technologies used together to develop web applications. In general, Ajax is a combination of:

- HTML or XHTML for the structure
- CSS for the presentation
- DOM and Javascript to dynamically manipulate represented information
- A method to execute asynchronous data transfers with the web server (XMLHttpRequest, iFrame, etc.)
- A format for the data exchanged between the server and the client durning the asynchronous transfer mode.

Motivation

It was necessary to add some dynamics in the web pages to have more usable operations. Because Javascript has some compatibility problems between Internet Explorer and Netscape-like web-browsers, it has been decided to use a javascript framework that also gave the possibility to use the full power of Ajax, and especially asynchronous transfers. It is easy to use asynchronous messages to send modifications of one entity of the system directly. Advantages are that only necessary information is send to the server and that modifications are reflected to the user without reloading the page.

5.1.4 HTML

HTML for Hypertext Markup Language [42] [62] is a markup language commonly used to represent web pages. It links the pages through an automatic navigation system of hyperlinks. HTML structures the content of the page semantically. It is also possible to include images and input forms, etc.

Example

A markup tag is delimited by "<" and ">", has a name and an optional number of arguments. Each tag must be closed either by the respective closing tag or a "/" at the end of the tag.

```
<TITLE>The title of my page</TITLE>
The phrase uses an <A HREF="target.html">hyperlink</A>.
<P>A paragraph that do not use other markup tags.</P>
<INPUT TYPE="text" NAME="firstName" VALUE="John" />
```

In the example we can see a simple HTML document that is composed of one title delimited by the tag `<TITLE></TITLE>`. A first sentence uses one hyperlink to a target document with the markup tag `<A></A>`, the referenced document is identified with the attribute `HREF`. A second sentence is delimited by a paragraph tag `<P></P>`. The last tag is an input tag where the user can put his first name to illustrate a "/" to close a tag. The `<INPUT>` tag has an attribute `type` that says it is a text field, a `name` for the input and the `value` for that name.

Motivation

HTML is a simple language that is widely used today. An HTML wireframe [15] of each window can be build very fast to have an immediate feedback of the user interface. Once the wireframe is accepted, it can be used within the Lisp application with Hunchentooth to generate dynamic web pages in minimal time. It also separates concern by representing the document content only and letting CSS deal with the document representation.

Alternative choices

Because of its simplicity, HTML has some limitation. This is especially true when implementing web applications. User interaction is difficult to simulate with web-pages. A combination of HTML and Javascript is necessary. Alternatives lies in XML based user-interfaces like XUL [46] [68] or XAML [47] [69].

5.1.5 CSS

CSS for Cascading Style Sheets [43] [63] is a language that is used for the presentation of HTML and XML data. The purpose of CSS is to separate the structure of the document and the presentation. Advantages of this separation is that accessibility is improved, presentation can be changed more easily and the complexity of a document is decreased. CSS can be described inside the document but also in a separate file. It is then possible to apply the same CSS to multiple documents. Because the CSS file remains in the cache of the web browser, interfaces are loaded faster.

Example

Each CSS entry is identified by a CSS selector [44] and a block of entires with the desired property and its value. The selector identifies the element of the document to which the values must be applied.

```
p {
    font-size: 110%;
    font-family: Helvetica, sans-serif;
}
```

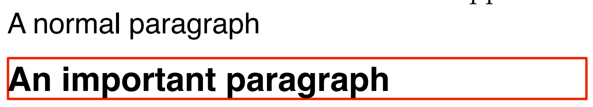
In this example, each element of the document identified by a paragraph tag will have a 110% font size and a specific font, Helvetica if it is available on the system or the default sans-serif font of the browser if Helvetica is not available. The "cascade" is the combination of multiple styles applied to the same document. To organize the style to apply, priority rules exists. It is for instance possible to describe the following property for document elements with the attribute "important".

```
.important {
    font-size: 120%;
    font-weight: bold;
    border: 1px solid #DD0000;
}
```

In the CSS example, the class "important" has a font size of 120% size and is bold, a red border is also added. The result will be applied to the next HTML example:

```
<p> A normal paragraph</p>
<p class="important">An important paragraph</p>
```

This is the result of the two CSS rules applied to the HTML example:


A normal paragraph

An important paragraph

As we can see, the first paragraph has the first CSS rule applied and the second paragraph has the two rules applied. Because `.important` is the last of the two rules, it is applied to the second paragraph.

Motivation

With CSS it is possible to focus on the structure of the document and add visual improvements later on. This leaves the design choices for the look and feel of the application to the very last moment and the programmer can focus on the features of the program. Multiple CSS sheets allow to use the cascade property to design general behaviors of the application and add specific style if new components are added to the application later on.

Alternative choices

It is not mandatory to use CSS to style a document and the style can directly be embedded in the HTML tags of a document. Doing so is however discouraged because it clutters the style information with the structure of the document.

5.1.6 jQuery

jQuery [31] is an open source Javascript framework that simplifies Javascript programming. It supports traversal of DOM [45] [65] elements, events, animation effects, CSS manipulation and plugins. The next example illustrates how jQuery works [34].

Example

This example will present how to change the default behavior of an anchor if that anchor has a class attribute "special". The example will be applied to the following HTML document:

```
<html>
<head>
    <style type="text/css">
        a.clicked { font-weight: bold; }
    </style>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
    /* Javascript code here */
    </script>
</head>
<body>
    <a class="special" href="http://jquery.com/">jQuery</a>
    <a href="http://jquery.com/">jQuery</a>
</body>
</html>
```

All jQuery code must be added inside the function that checks the document and wait until it is ready. This code is added in the HTML page between the tags `<script></script>`:

```
$(document).ready(function(){
    // jQuery code here
});
```

Now we will add a a click handler to the link with class attribute "special". The script will change the default behavior of the anchor to show a popup window that displays a message to the user and that change the class attribute of the anchor.

```
$(document).ready(function(){
    $("a.special").click( function(event){
        alert("You clicked on an anchor");
        $(this).addClass("clicked");
        $(this).hide("slow").show("slow");
        return false;
    });
});
```

- The anchor is selected in the document with the syntax `$("a.special")`. With this syntax all the anchors with an attribute class "special" are selected but in our example there is only one element.
- Each selected anchor will have a function associated to the event `click`.
- The function displays a dialog message with `alert`. It is a standard Javascript function to display a dialog with a message.
- `$(this)` is the current element inside the function. In our example, we add a new class "clicked" to the element. Because of the CSS, the anchor will become bold.
- Then we make an effect by making the element disappear and reappear again.
- And finally we prevent the default behavior of the anchor by returning false.

We saw the basic operations of jQuery but an important feature that is used extensively in the program is the use of plugins. An example of how to implement a plugin to improve a view of the system is presented in section 4.2.2.

Motivation

To have dynamic web pages with HTML, Javascript code must be added to the document. This is for example the case when actions must be added to a button without having to submit the page to the server. Javascript however, is not supported identically on Microsoft web browsers like Internet Explorer and on other web browsers like Safari, Opera or Mozilla. To solve this problem, several frameworks add an abstract layer above the base Javascript operations. With the rise of “Web2.0”, lots of Javascript frameworks emerged. They support different functionality like effects and Ajax.

jQuery has been chosen for its good documentation with embedded examples, the possibility to use CSS selectors to navigate the DOM elements on the HTML page and a system of plugins. Other less important reasons are the lightweight footprint of the script (19kb when compressed), support of Ajax and graphic effects. The huge community that propose many tutorials also makes it an easy to learn framework with many plugins available for the user.

Alternative choices

Many other Javascript frameworks exist today. Some of the most known are Scriptaculous [38], Dojo Toolkit [37]. Yahoo and Google also propose a framework for Javascript (respectively “User Interface Library” [39] and “Google Web Toolkit” [40]). It is very difficult to choose the best framework today and an extensive study should be required to precisely identify the advantages and limitations of each framework. These considerations are not important for the implementation of the documentation system. Choosing one that fits the programmer seems a good choice for a first model of the documentation system.

We saw the different technologies used by the system and motivated their choices. In that process, other technologies have been presented as well. We can now present how the system is implemented to provide lively documentation. Different levels of details will explain how the program works and how the technologies are used to lead to the implementation of the documentation system presented in chapter 4.

5.2 System organization

We will start by giving a structure that illustrates the external properties of the system.

The documentation system is a web application [14]. It is therefore made of several distinct components represented if figure 5.1.

- The **web-browser** is the interface to access the documentation system functionalities.
- The **server** component handles the requests of the user and respond

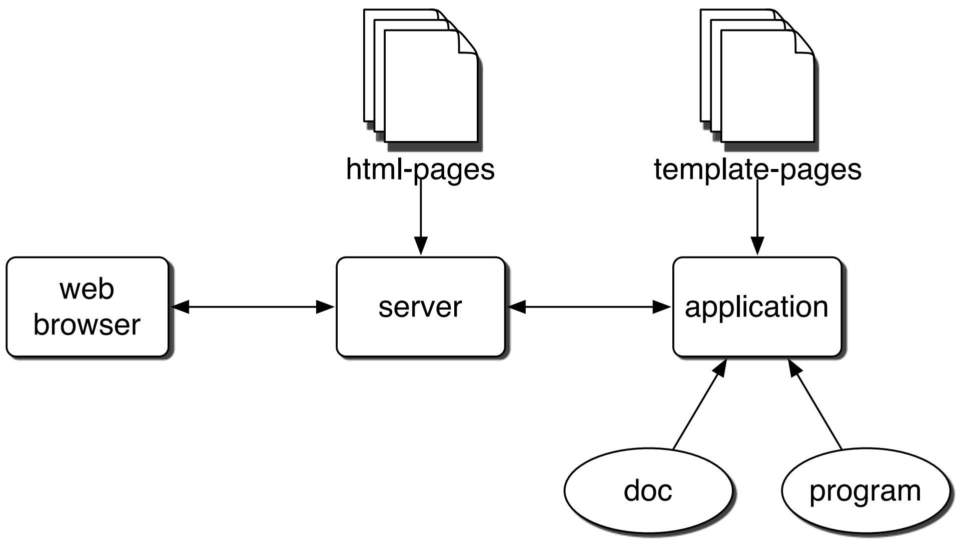


Figure 5.1: Architecture of the system

with **html-pages**. To respond to the web-browser, the server communicate to the application component.

- The **application** component does all the internal treatment to retrieve information from the program and from the documentation. **Template pages** are used to convert the information into html pages before they are sent back to the server.
- The **documentation** and the **program** are the entities from which information is extracted. As a result, no database is used here contrary to what is usually done.

5.3 User tasks

A use-case diagram 5.2 shows the different persons that use the system and the tasks they have to realize. Each person has a precise role:

- The **documenter** is the person who writes documentation for his program. He must be able to read and edit the documentation of the program he is coding. It is also the person who is in charge of making tutorials.
- The **tester** writes tests for the program. Usually, the tester and the programmer should be two separated persons, but this is not always the case. Anyway, the tasks a tester must be able to do with the

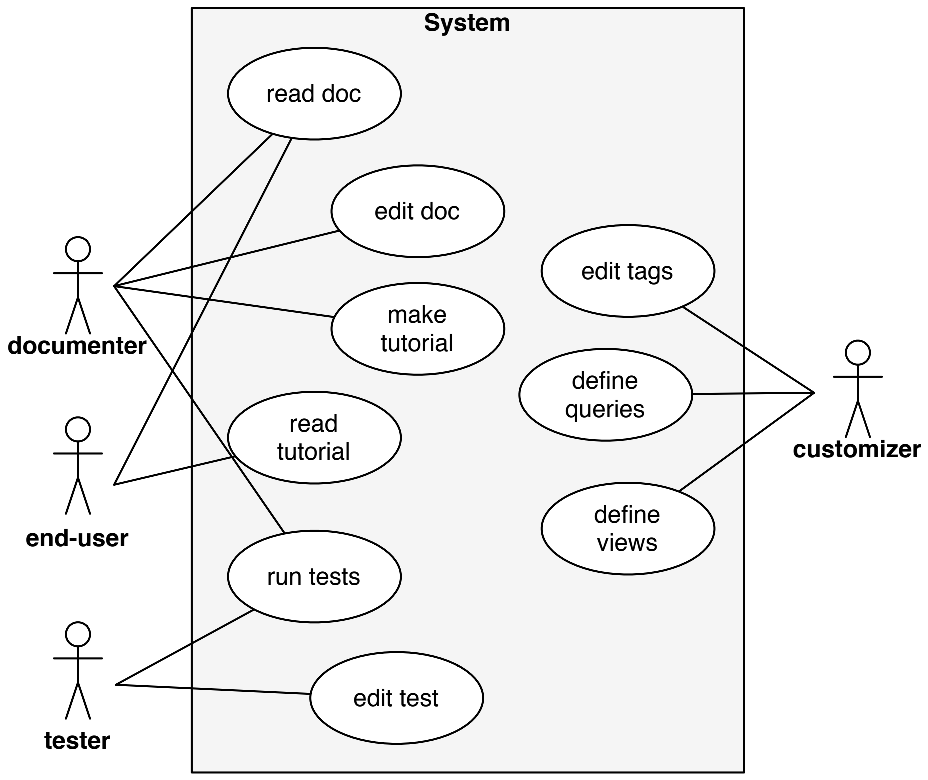


Figure 5.2: Use case diagram of the system

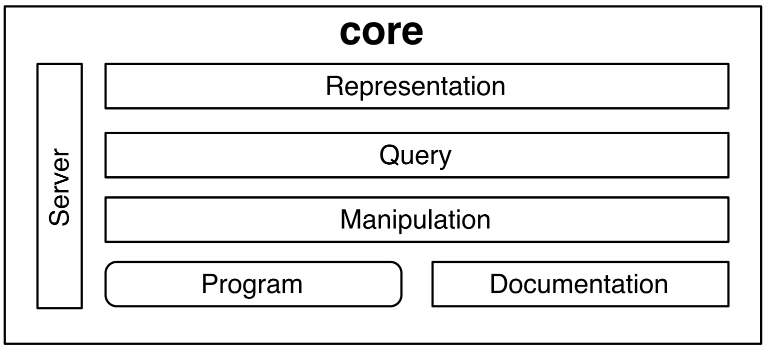


Figure 5.3: Core components of the system

system is to read the documentation of the program and also edit new tests and run them to verify that the program to test passes them.

- The **end-user** is the person that need documentation to use a program like an API or a Framework for instance. He will usually read the documentation and the tutorials and try out the program.
- The **customizer** is the person that shows the documentation features in a certain way. He must be able to define the queries to retrieve information, define the views to show the results and also decide which tags exists in the system.

5.4 System modules

The system is made up of two parts. The **core** system is a framework to support documentation. The **ldoc** system defines a concrete implementation of the framework to build a lively documentation system.

5.4.1 Core

The **core** system is made up of several components 5.3:

- The **program** is the component that needs documentation. It will serve as input for the documentation system. We assume that the programing language or the environment in which the program is expressed has some intercession capabilities.
- The **documentation** component is responsible for storing and retrieving documentation of the associated program. It also abstract away the concrete implementation of persistency. Because the type of documentation might be text, images, links, etc., an abstraction represent

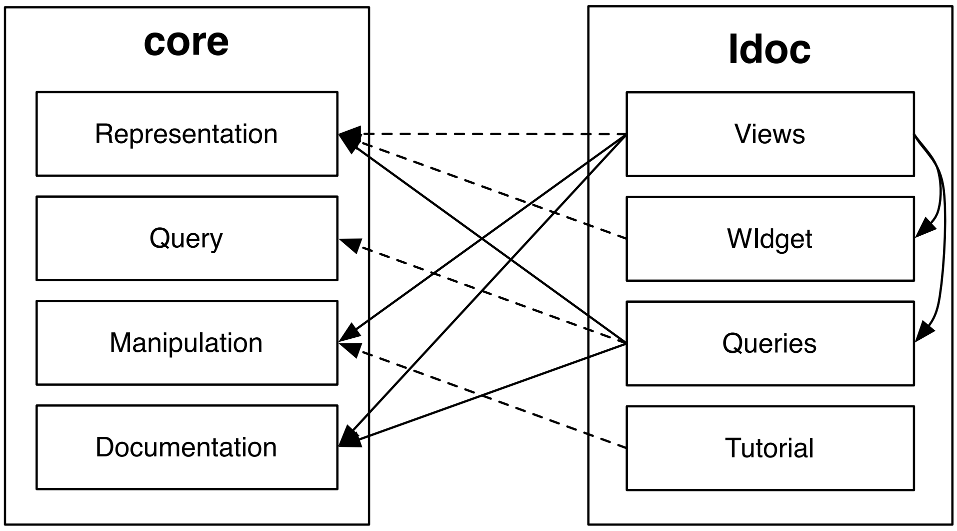


Figure 5.4: Components of the documentation system

all the possibilities toward the other components and deals with the real data internally.

- The **manipulation** component handle the data coming from the documentation and from the program to merge them together.
- The **server** component deals with the elements that are necessary to run a web server and handle requests and responses. It defines the variables of the server of the user and the operations to load plugins of the system.
- The **query** interface represent an abstraction to retrieve information uniformly over the system, might it be documentation entries or program entities.
- The **representation** component deals with the representation of the information and the interactions with the user. This component define operations to make new views and a hierarchy of abstract components to handle display.

5.4.2 Ldoc

The **ldoc** 5.4 system implement the lively documentation system on top of the **core** features:

- The **views** component define all the concrete views and handlers of the system. It defines abstract views for documentation and concrete views like a home page, a package listing, a class listing, an administration page, etc. The handlers manage adding new documentation entries, tags or option modifications.

- The **widget** component define all the reusable widgets of the documentation system.
- The **queries** component define all the concrete queries to retrieve information like all the entities of a package or all entities with a given name.
- The **tutorial** component define new entities to represent tutorials and manage creation of tutorials in the documentation system.

The dotted arrows represent extensions from the core component in the ldoc component. As we can see, these extensions are limited from one core component into one ldoc component. The full arrows represent the elements of the core component that the ldoc component uses:

- The **queries** component manipulates data from the meta-object protocol and from the documentation.
- The **views** component manipulates these two data as well. The **views** component also uses the queries defined in the ldoc system as well as the widgets.

To stay clear, the interaction between the tutorial component and the other components are not represented but they can be seen as an extension to the representation and therefore have the same interactions as observed between the representation component and the other components.

5.5 Internal representation

Now that we have a global view of the system, we will see how entities are represented internally. We will focus on how to manipulate data coming from the documented program, handle views to generate HTML pages and present the notion of tags in the documentation system. We will firstly present how entities of the program to document are represented internally. It is however possible to improve the system with another solution that will serve as example in the case study of the system 6.1 (The new solution is not implemented in the system for time reasons).

5.5.1 Representation of program entities

Entities of the program to document need an internal representation to be manipulated. It has been decided to define internal objects that define the

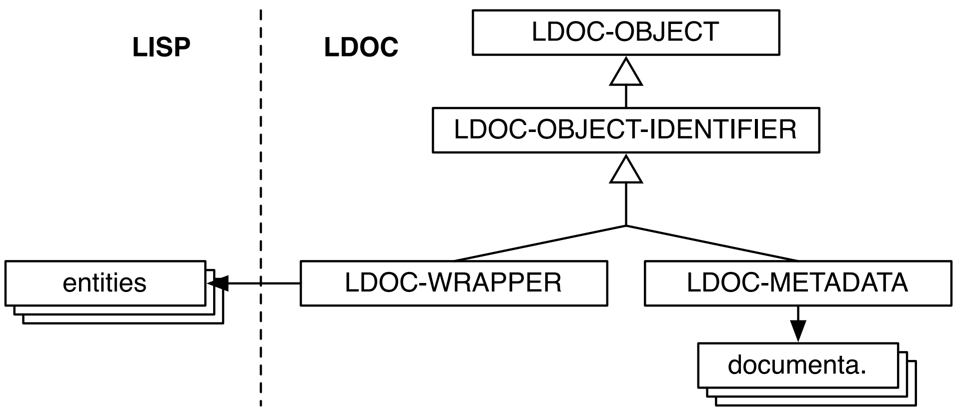


Figure 5.5: Internal representation of entities

possible operations and to wrap the entities of the program in these internal objects. The figure 5.5 shows the chosen hierarchy for the documentation system. The advantages of wrapping the program entities is to leave the program intact without overloading the entities with code necessary for the documentation system. It follows the general concept of encapsulation [3].

- **Ldoc-Object** is the base object for the entities in the documentation system.
- **Ldoc-Object-Identifier** defines objects that have unique identifiers. With this concept it is for instance possible to retrieve a class named *bar* with the following string: `package/class/bar`. This notion is interesting for showing entities with a human readable name in the address-bar of a web-browser.
- **Ldoc-Wrapper** is the class that wraps the entities of the program to document and provides functionality to the documentation system.
- **Ldoc-Metadata** describe any data that can be associated to entities of the documented program. This is for instance the case of the documentation entries.

5.5.2 Uniform data manipulation

To manipulate data uniformly in the system, a *pipes and filters* structure is used as represented in figure 5.6. List are easy to manipulate and makes them the perfect data structure to cary data. A generator will produce initial data like entities of the program and wrap them in an `ldoc-wrapper`. The sieves only filter the list according to the input parameter.

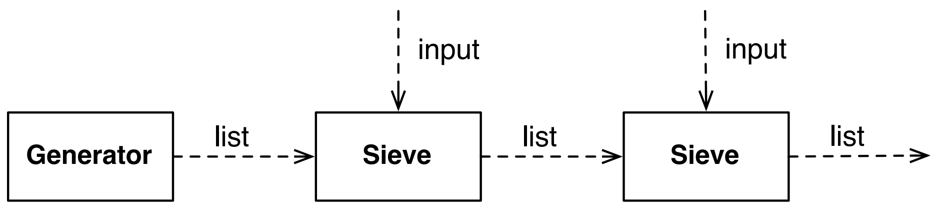


Figure 5.6: Pipes and filters used to manipulate data

This solution abstracts away the fact that a program is composed of classes or functions and that the documentation system uses wrappers, documentation entries, etc. It is thus possible to get data about the system with a common syntax as represented in the next example:

```
(query "packages")
(query "name" "ldoc" "class" "ajax-entity-handler")
(query "tagged" "abstract")
```

These three queries use the same syntax and will return a list of `ldoc-wrapper`. The first query return all the packages of the documented program. The second query return a precise class in the package `ldoc`. The third query return all the entities tagged as "abstract". As we can see, it becomes easy to manipulate any kind of data coming from the program or from the documentation system.

There is a performance problem associated to the approach. A lot of elements are created before they are filtered. This is aggravated by the fact that multiple queries will not use the previously generated lists to filter the data. For each new query, all elements are created again. This might become a bottleneck in the system. The query also lacks some expressivity like joining lists, etc.

Performance issues are not a problem for the model and the final documentation system handles the creation with a minimal time overhead and still providing relative easy data manipulation as was required.

5.5.3 Representation of views

In the documentation system, the chosen representation of data is HTML. HTML templates are used to describe components and are then filled in with the appropriate data. To abstract away these manipulations, a hierarchy of classes has been defined.

The hierarchy is represented in figure 5.7. The base class that handles display of data in the system is the **widget**. The system also defines a

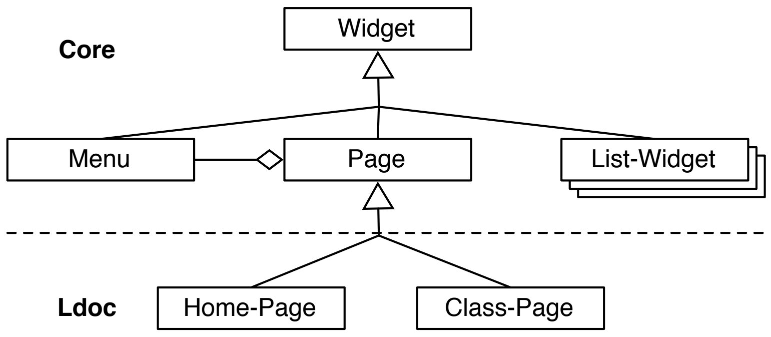


Figure 5.7: Internal representation of views

page class that is responsible to display a whole page on the web-browser. Finally, a set of other classes are implemented to show basic representations like lists or tables. The user can extend these base classes to make concrete pages like a home page or a page to show the entities of a program.

The advantage brought by the solution is a separation between HTML code that requires a specific syntax to fill the HTML templates and Lisp code. The classes are defined once with their corresponding template and the programmer must not deal with templates later on. If HTML is to be replaced for another representation then modifications are contained in widget classes only.

On the other hand, for each new element to display, new classes must be created. Every HTML tag should have its Lisp counterpart. In practice, for time reasons, not all HTML tags have been implemented. As a result, a combination of widget classes and template representations is used within the system.

5.5.4 Representation of tags

An important concept in the system is the notion of **tags**. They are used to add information to documented entities. To keep a certain organization in those tags and to allow relations between them, tags are represented in a tree structure.

The figure 5.8 represent a hierarchy of tags as they are used in the system. The advantage of such organization is that manipulation of tags is relatively easy. Interesting information can also be extracted from entities that are tagged.

- It is for instance possible to define related entities as entities that have

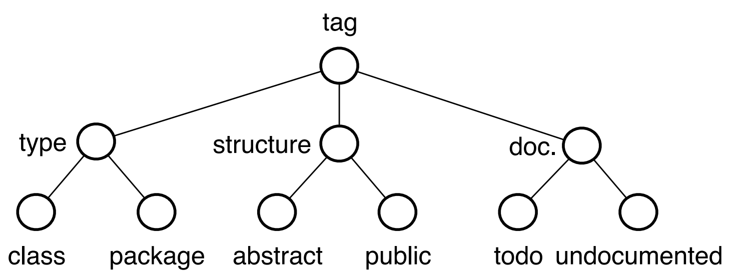


Figure 5.8: Internal representation of tags

sibling tags. (Get the parent tag, then all the children tags, finally get all the entities tagged with those children tags)

- If an entity is tagged somewhere in the tag tree, all the information relative to the parents of the tag is automatically available.

The problem is to choose a tree structure that has a correct semantic. These examples are only a glimpse of what can be accomplished with such structures and it was not possible to investigate the idea more in detail.

5.6 Conclusion

The chosen programming language drove the documentation system toward a web-based application. We then presented the web technologies that where used to implement the system. We presented the architecture of such applications and the design choices to actually implement a working documentation system presented in chapter 4. We also disscussed of advantages and the disadvantages of the proposed solutions.

Chapter 6

Validation

We presented how the system work. We saw illustrations of a mockup followed by screenshots of the documentation system. We saw that several tasks where implemented and that the system seems to work well. After that, we presented the implementation so that the reader understands how the system was implemented and the technologies that where involved in the process. Now that the documentation system is presented, we would like to validate it and see if it does respond to the requirements and if it does improve documentation. We would like to know the features that work as well as the features that are not working. A case study will set the documentation system at work by applying the methodology to an example, then the quality of the documentation system will be evaluated. Finally, those evaluations will be discussed to assess the realization of the documentation system.

6.1 Case study

To validate our concept, we would like to apply the presented methodology to a concrete example and show how the documentation system can be of use in the process. The example (coming from [1], chapter 3: "Practical: A Simple Database") will be kept simple to focus on the methodology and not on Lisp programming. The methodology is made of five phases: design, wireframing, implementation, lively documentation and testing. Implementation and lively documentation are presented together because those phases are tightly coupled to each other. We will not go to much in the detail of the design phase.

6.1.1 Design

The user of the database application should be able to add entries in a database and to query the database. We do not want to rely on existing databases but on an internal LISP representation that do not need object relational mapping [16]. We then fill it with some CDs to illustrate how the database works and how to query it.

6.1.2 Wireframing

The interface proposed for the database application is written in LISP so that it becomes navigable in the lively documentation system:

```
(defclass DataBase(standard-object)
  (:documentation "Represent an abstract database in LISP"))

(defmethod add-record ((db DataBase) record)
  "Add a record to the database")

(defmethod dump-db ((db DataBase))
  "Return a string representation of the records of the
database in human readable format" )

(defmethod load-db ((db DataBase) filename)
  "Load the database from the file given in parameter")

(defmethod select ((db DataBase) where-fn)
 "Select takes a function as input that filter the records
of the database and that retain only those that match with
the where function given in parameter")
```

A class represent the **database** and several operations are possible.

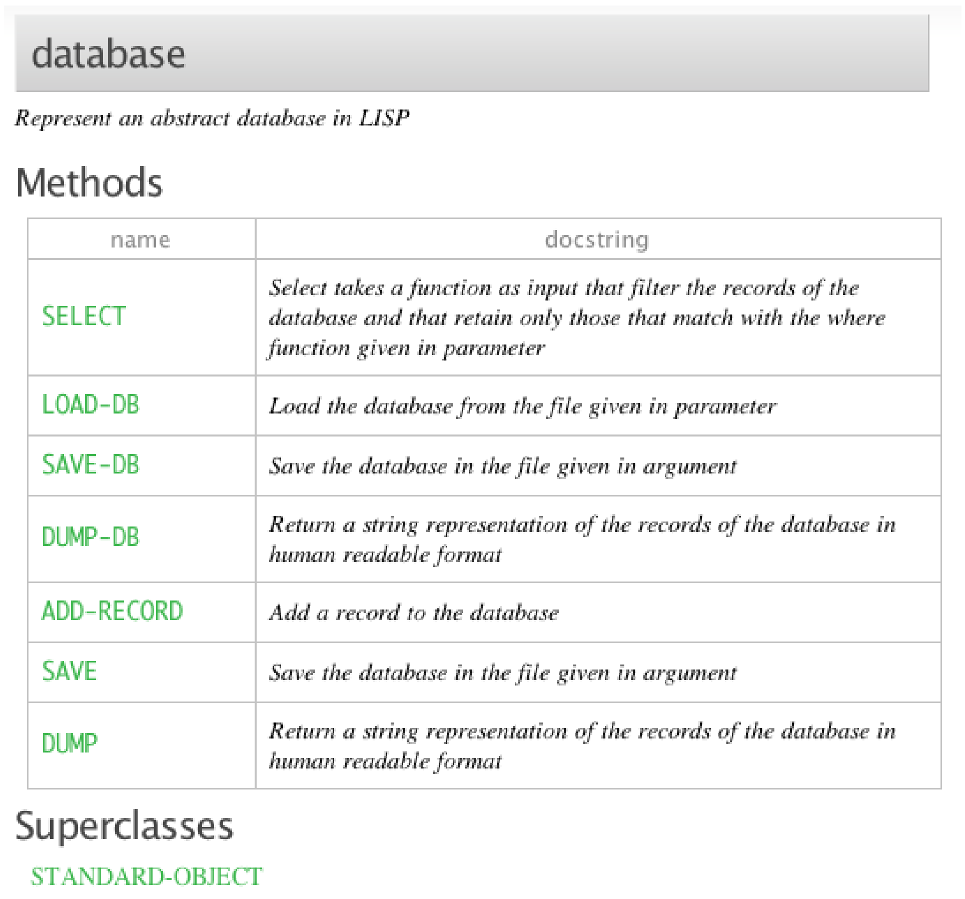
- Add records to the database.
- Save the content of the database to a file and to load it from a file.
- Display the content of the database in human readable format.
- Query the database with selects

The user can also create a new database with a given name and a CD with a title, an artist and a rating:

```
(defun make-database (name)
  "Create a new database with a given name")

(defun make-cd (title artist rating)
  "Make a new cd with the given arguments. Title: the title
of the CD, Artist: the name of the persons who made the CD,
Rating: A score between 0 and 10")
```

In the lively documentation system, it is now possible to browse for the database and see the documentation of the different methods as represented in screenshot 6.1.



database

Represent an abstract database in LISP

Methods

name	docstring
SELECT	*Select takes a function as input that filter the records of the database and that retain only those that match with the where function given in parameter*
LOAD-DB	*Load the database from the file given in parameter*
SAVE-DB	*Save the database in the file given in argument*
DUMP-DB	*Return a string representation of the records of the database in human readable format*
ADD-RECORD	*Add a record to the database*
SAVE	*Save the database in the file given in argument*
DUMP	*Return a string representation of the records of the database in human readable format*

Superclasses

STANDARD-OBJECT

Figure 6.1: Documentation of class Database and its methods

6.1.3 Testing

While the programmer writes code, the tester writes test-units to assess the correctness of the program. He can then add them to the documentation system so that they can be run later on, when navigating the documentation. The tests won't be exhaustive here:

```
(define-test test-querying
  (let ( (l (list "Fly" "Home" "Roses"
                  "The clairvoyant" "Best of the beast")) )
    (loop for elt in l
       do (assert-true (select *cd-database*
                               (where :title elt)))))
  (assert-false (select *cd-database*
                        (where :title nil)))
  (assert-false (select *cd-database*
                        (where :title "Doudou"))))

(run-tests test-querying)
```

A test is defined to see if querying works according to expectations. Four titles should be found in the database and `nil` and `Doudou` should not. These tests are then entered in the documentation and will be tested afterwards.

6.1.4 Implementation and lively documentation

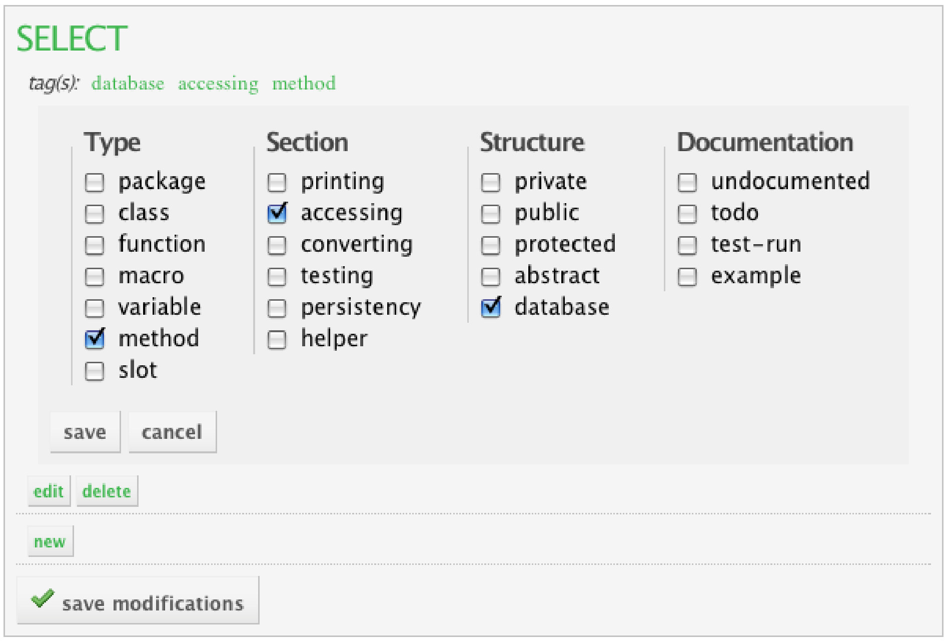
Now that the wireframe seems fine, the programmer implements his solution. He adds fields to the `DataBase` class. He defines one new macro for the creation of `where` functions (that are passed to the `select` method). He also needs two helper functions for the implementation of that function: `make-comparison-expr` and `make-comparisons-list`.

```
(defclass DataBase(standard-object)
  ((name    :initarg :name
            :initform (error "A database should have a name")
            :reader get-name)
   (data    :initform nil)))

(defmethod add-record ((db DataBase) record)
  (with-slots (data) db
    (push record data)))

(defun make-database (name)
  (make-instance 'DataBase :name name))

(defmethod dump-db ((db DataBase))
  (with-slots (data) db
    (dolist (record data)
      (format t "~{~a:~10t~a~%~}~%" record))))
```


SELECT

tag(s): database accessing method

Type	Section	Structure	Documentation
☐ package	☐ printing	☐ private	☐ undocumented
☐ class	☑ accessing	☐ public	☐ todo
☐ function	☐ converting	☐ protected	☐ test-run
☐ macro	☐ testing	☐ abstract	☐ example
☐ variable	☐ persistency	☑ database	
☑ method	☐ helper		
☐ slot			

Figure 6.2: Tagging the select method of object Database

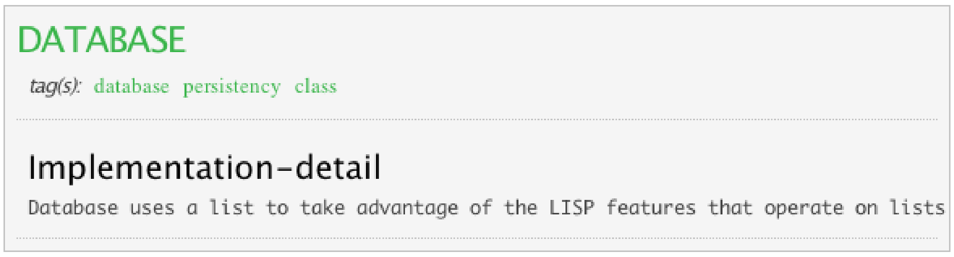
- The database is implemented using slot `data` that is a list. This is to take advantage of the LISP features that operate on lists. We also give a `name` to the database.
- `add-record` push the new record in the `data` slot of the database.
- `make-db` create a new instance of the database with a given name.
- `dump-db` loop over each record of the list and print it with the name of the record and its value, one per line.
- `save-db` and `load-db` use file operations to write or read the `data` slot in a file, they are not presented here.

Once code is written, the programmer tags all the entities presented before, like methods, classes and helper functions in their respective sections to keep them organized (screenshot 6.2). He also uses the tag **database** to retrieve all his entities later on. The entities are then classified as follows:

accessing add-record, get-name

querying select, where

persistency load-db, save-db

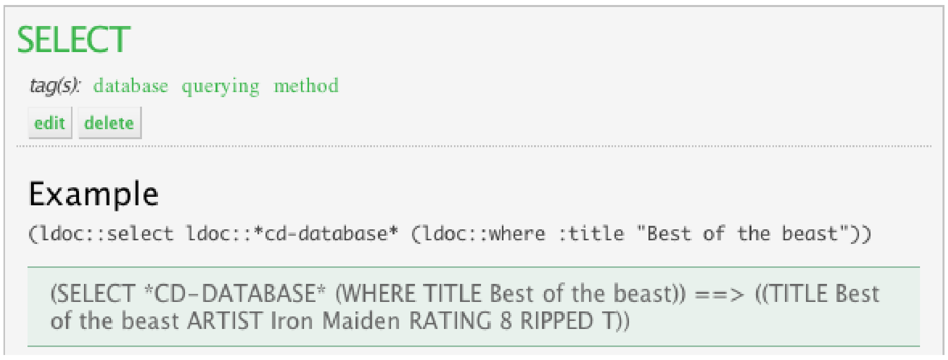

DATABASE

tag(s): database persistency class

Implementation-detail

```
Database uses a list to take advantage of the LISP features that operate on lists
```

Figure 6.3: Adding an implementation detail in the documentation


SELECT

tag(s): database querying method

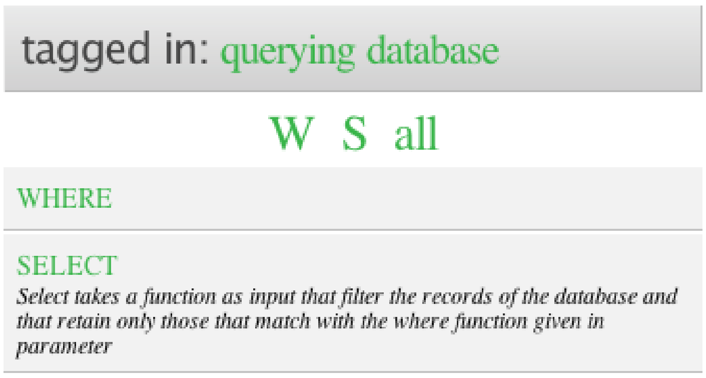
edit delete

Example

```
(ldoc::select ldoc::*cd-database* (ldoc::where :title "Best of the beast"))
```

```
(SELECT *CD-DATABASE* (WHERE TITLE Best of the beast)) ==> ((TITLE Best
of the beast ARTIST Iron Maiden RATING 8 RIPPED T))
```

Figure 6.4: Running code for an example of query


tagged in: querying database

W S all

WHERE

SELECT

Select takes a function as input that filter the records of the database and that retain only those that match with the where function given in parameter

Figure 6.5: Results for entities tagged in `querying` and `database`

Figure 6.6: Execution of the tests

Figure 6.7: A tutorial that show how to use the created database

printing dump-db

helper make-comparison-expr, make-comparisons-list

The programmer also adds documentation, like the fact that database uses lists (screenshot 6.3). A runnable example is added to show how to use `select` and `where` (screenshot 6.4). To verify that the programmer did not forget an entity, he searches for entities tagged in `querying` and `database`. He effectively finds the two defined querying methods `select` and `where` (screenshot 6.5).

The tester finally browses to the documentation of `database` to see if the tests pass correctly. The documentation system shows that all the tests are correct (screenshot 6.6).

Now that the program is working, the programmer wants to create a tutorial on how to use the database. With the existing documentation that has already been added in the system, less effort is necessary.
The tutorial is composed of four sections and is named `using-database`.

- Make a new database
- Create CDs
- Add records
- Query database

For each section, a comment is added to explain the step. Because examples have been added throughout the implementation, it is possible to get more information from the defined entities by linking to their documentation (screenshot 6.7).

It would be difficult to show multiple different variants of the same example to show the possibilities of the system. This example, however, shows that it is possible to add and retrieve documentation with less effort. The programmer do not need to document all his code but he can rely on tags and add examples. These examples can then be aggregated to show a new information made of distinct informations that where already present with less effort for the programmer. The work of the tester is also put in the system and bring some more documentation. We will know proceed to a more methodical assessment of the system.

6.2 Quality assessment

Now that the methodology has been demonstrated, we would like to asses the qualities of the implemented system to see if it effectively improve software documentation. The qualities to be assessed concern different aspects of the system. We would like to asses the **code**, the **user interface** and the **tasks**. The tasks will be evaluated according to the following qualities:

correctness	The aspect is correct with respect to the specifications
completeness	The aspect is totally or partially implemented
usability	The aspect is easy to use

6.2.1 Browse documentation

The user can browse entities by multiple ways in the system. If he knows what he is looking for, then he can browse the package and finally the entity in the package. If he knows tags of the entity, he can search for those tags and then browse the result to find his entity.

Show documentation

correctness It is possible to browse for an entity starting from any page in the system and the entities are displayed correctly. All the documentation of each entity is displayed as the screenshots in chapter 4 shows.

completeness It is not possible to browse for all types of entities directly. Packages, classes, functions, macros and variables are the only types supported for now. There are also some problems with entities of packages that do not belong to the LDOC package of the system.

usability A user can (i) browse all the packages of the system. (ii) See all the entities of a package. (iii) Browse the content of a class or a function. The user can also browse for entities in his current package directly from the menu (figure 4.8). Only supported entities are put in the menu. The user can also filter the entities of a package alphabetically (figure 4.8).

Search documentation

correctness The user can search for tagged elements. He can also search for multiple tags by separating them with a colon. The result is always the entities that have all the tags the user is looking for. If there is no entity with that tag, the systems displays no error.

completeness The user cannot search for entities by another mean than providing the tag of an entity. This means that he cannot lookup entities with a given name or according to word matching in the content of the entity's documentation.

usability To search an entity in the system, the user has to: (i) Type the desired research in the search field (figure 4.8). (ii) Browse the entity from the result page.

Filter entities

Whenever there are listings, the user should be able to retrieve information more easily.

correctness The filters allow to sort entities on their type or alphabetically depending on the pages. When an element of the filter is clicked, the corresponding entities are displayed correctly. Some alphabetical listings are not ordered correctly.

completeness Some pages allow to filter entities based on their type, other pages allow to filter entities alphabetically. It is not possible to switch between both. Results of a search are not filterable

usability When filter items are clicked, entities are filtered instantaneously.

6.2.2 Edit documentation

correctness The documentation that is showed match with the entity of the system. The modifications submitted by a user are saved for persistency and the user can retrieve that information when he restart the system.

completeness The user can document entities that are browsable. The user can set documentation for one entity. The input can be simple text, Lisp code or markdown syntax. It is not possible to document multiple entities with one documentation entry. The user cannot add media to an entity. If an entity is modified in the code, the documentation is not updated to point to the new entity. It is not possible to document packages.

usability To document an entity, the entity must exist in the system. Then the user has to: (i) Retrieve the entity. (ii) Open the documentation editor. (iii) Add a new documentation entry or modify an existing documentation entry. (iv) Save the modifications.

One comment for multiple entities

It is not possible to add one comment for multiple entities. This means that that documentation like design patterns that relates to multiple entities of the system is not possible. Tutorials use multiple entities but the link to the referencing entities must be added manually.

Support multiple formatting

The system support simple text and Lisp code as primary formatting. A plugin to support Markdown has also been added to the system. If the user wants to support other types of formatting, it is possible by programming the support for that formatting into the system.

correctness The supported formatting is correctly processed and receives specific treatment for display depending on the formatting type selected by the user.

completeness It is only possible to support text, lisp code and markdown at the moment. Other formatting are possible but must be added to the system by the user.

usability To use a formatting, the user has to add support for that formatting into the system. Then the user has to: (i) Retrieve an entity. (ii) Open the documentation editor. (iii) Write in the desired formatting. (iv) Set the type of documentation to the desired formatting. (v) Save modifications to see the processed output.

Add files (images, diagrams, media)

With support of the markup language, it is possible to add links to any content the user wants to point to. This feature is not supported by the system yet. This means that the user has to handle the media to be added manually.

correctness No direct support of files.

completeness There is no specific treatment for medias or files in the system. It is however possible to point to files with links written in markdown for instance.

usability The user has to: (i) Put the files in the static file directory of the server. (ii) Open the documentation editor of an entity. (iii) Use markdown to make a link that points to the resource. (iv) Save the modifications.

6.2.3 Handle tags

To tag an entity, two steps are required. The first step is to add a new type of tag into the system if the tag is not available. This happens in the administration section. Once a tag is added, the user can browse the desired entity and click to open the tag editor and do the changes. There are some tags that are handled by the system. The "todo" tag is added automatically to entities that are documented with a "todo" documentation entry and elements are tagged automatically with "undocumented" and the type of the entity.

Add new tags

correctness Any tag that is submitted by the user is saved and the tag can then be used to markup any browsable entity in the system.

completeness The user can use any submitted tag to markup the system excepted for the root tag that is the base tag of all tags.

usability Adding a new tag to the system requires the following user operations: (i) Go to the administration page. (ii) Enter a tag name and choose a parent for the tag in the dropdown. (iii) Submit new tag

Tag entities

correctness The user can choose any type of existing tag. It is however not possible to force some restrictions (user can choose multiple types like class, method and package together, which has no sense).

completeness An entity is automatically tagged with some specific tags and the user can tag entities with any all the tags of the system.

usability Adding a new tag to the system requires the following user operations: (i) Retrieve the entity. (ii) Open the tag editor. (iii) Check any tag that the entity must have. (iv) Submit changes.

6.2.4 Handle documentation persistency

correctness Any documentation that is submitted by the user is saved and the data retrieved correspond to the data that has been saved.

completeness Documentation is saved in a file on disk with one file per entity. The encoding is in Lisp. No versioning is done on these files. If the user makes a change, the previous data are overridden.

usability (i) The user has to open the document editor. (ii) Add or edit the documentation. (iii) Click on "*save modifications*" to save the documentation modifications.

6.2.5 Run code from examples

correctness The user can run any source code that is written and saved under type "*code*". Lisp-unit is loaded in the system automatically. Because the code is executed in a null-lexical environment, entities of the system must be preceded with their package identifier. Code that is copy/pasted from the system need to be modified to run correctly.

completeness The code written is either correct Lisp code and a response is sent back to the user or incorrect which generate an error message.

usability (i) The user has to open the document editor. (ii) Write code. (iii) save it as "*test-run*". (iii) Click on "*run*" to execute the code.

6.2.6 Support multiple views

correctness Any user can edit the documentation by default but end-users should not have that possibility. It is not possible to customize the views depending on the type of user.

completeness Options allow to activate or deactivate editing. No other options are implemented. Any user can change the options.

usability To activate editing the user has to: (i) Navigate to the options. (ii) Check the editing option. (iii) Click on "*save modifications*" to save the options modifications.

6.3 Discussion

We saw the documentation system at work to help programmers in the application of our methodology. A methodical analysis pointed out features that works and features that do not work yet. We can now go back through the different results obtained to see if the system responds to the requirements of the subject as they are stated in chapter 3. The features that are not working well or need to be reconsidered will also be pointed out.

6.3.1 Browse documentation

Entities are represented correctly. After some usage, the alphabetical sorter and the filter programmed in Javascript have been found very useful. This greatly improves the navigation of the system. It really helps finding the desired information in an easy way.

On the other hand, two problems remain. Firstly, not all types of entities are listed. `condition-classes` and `generic-functions` might be added to the system. It is however possible to add them to the structure of the system. Secondly, the menu is used to browse the main package of the system. This is defined as the default package in which the user is when he starts the application. It is however not possible to change that behavior and it is not mentioned properly in which package the user is. This could be resolved in several ways:

- A drop-down menu that let the user choose his main package
- An option in the option page
- When the user choose a package on the home page, that package is set as default package.

6.3.2 Edit documentation

It is possible to document any entity of the system. With this system, the user can easily document entities with lots of useful data. The fact that the code can be documented in a web browser makes it easy to use. It is convenient to add documentation that way. The possibility to support markdown is a convenient way to write documentation.

There are however problems that might deserve some ameliorations. It is not possible to add media to the documentation. Adding images and diagrams would help to have a better documentation. It is also not possible to make a documentation that address multiple entities. Adding documentation that concerns multiple entities is a way to have more structured documentation that can address more needs than technical documentation only. The rich-text editor for editing the documentation is also too restrictive for daily use if it had to be used as it is. An advanced Javascript rich-text editor like *wmd* [35] or *markitup* [36] should definitely solve that problem.

6.3.3 Handle tags

The system of tags has been proven very useful. It can serve in two ways. The first one is to retrieve information more easily, like the “todo” tag. The second one is to give more information about the documented entity. This is one of the features that has been the most interesting throughout the documentation of the system. The fact that tags are added automatically is very handy.

This feature can also become problematic if badly used. If to much tags are used, they become useless. The user could also tag these entities in bad sections. The fact that no restrictions are possible could lead to even more mistakes. In practice, the number of possible tags should remain small to keep documentation structured and more tags should be handled by the system.

6.3.4 Handle documentation persistency

The persistence support is minimal but transparent for the user. The advantages of having them on disk in a hierarchical folder structure identical to the system structure is useful to retrieve information and to edit it manually if needed. These advantage have been proved durning the first phases of documentation, when the system was unstable. A solution like elephant [24], a persistent object database for Common Lisp might have been simpler after some reflection. Because of time necessity, it was not possible test elephant in the implementation of the system.

6.3.5 Run code from examples

The idea of running code in general is very good and the possibilities offered by this feature really helps users as well as testers. With the jQuery documentation for instance, it is possible to execute the examples presented in the documentation. This is a very good way to understand code, to illustrate it and to show that it works.

The feature of the system is however very immature and does not fit very well to testers. A specific view should be implemented for the testers that gives more focus on tests and their result. The use of a unit testing library should be considered in this case. In real situations it is also dangerous to send code that must be executed on the server side. These possibilities should be guarded by administration mechanism.

6.3.6 Support multiple views

Navigation is easy for users wanting to read documentation as well as for programmers documenting their code. No switching is needed to pass from one task to the other.

However the distinction between both could be more precise like websites with an administration section that allows modification of the content of the site for the documenter and another section for the tester. Tasks should be supported in the system with information about the user. This involves users management, sessions, etc.

6.3.7 Extensibility

Section 4.2.1 and 4.2.2 show how the user can add a tag cloud to the system by creating a new query for the system and a new widget. This possibility is brought by abstractions that are used in the application. These abstraction can be used to define any new element the user might need.

The same is also the case for adding new features to the documentation system. The documentation system hierarchy presented in figure 5.4 allow the programmer to add new behavior without touching other pieces of the program. The tutorial add-on has been implemented with almost no changes in the core features of the system. The only feature that was not implemented was support for multiple entities. Once that feature was added, all the other components of the tutorial have been implemented in a separate module.

6.3.8 Modularity

The architecture and the design diagrams show that the system is made of several components that works together. These components have weak coupling between each other. The internal representation of the data is made up of lists that are a base structure of LISP. If the output had to change from HTML to XML, only minimal modifications of the system is required.

6.3.9 Reusability

The system mainly deals with display of information that is represented as LISP lists and build a full web-application based on these data. The way to retrieve information uses the query abstraction. The system needs minimal changes to become a good base for a web framework. Automatic support of extensions and configuration files allow the user to parametrize the elements to load at application startup. With that, the system could be reused to build web-sites easily.

6.3.10 Usability

CSS is omnipresent in the user interface and a strong identity is present. The same structure is conserved over all pages of the system. The design follows classical web user interfaces with a top menu, a side column and a central content. Links are clearly identifiable with a distinct color. User actions are defined by buttons that are clearly identifiable as well. Javascript enforce the user experience and brings some dynamics to the pages of the system. This allow the system to be more responsive and simpler to use.

6.4 Conclusion

The documentation system is at a first stage of development. It gave the possibility to use it and to apply the methodology presented to improve documentation.

As a tool to read documentation and to retrieve it dynamically, the program answers to the limitations of other documentation systems. It is possible to browse entities without compiling source code and retrieve them with a search feature. It is also possible to edit the entities, add any documentation that could be useful and still keep structure in the documentation. The tagging process reveals to be very powerful and answer to the problem of time consuming documentation. Tagging entities is really fast and straightforward, it also help to keep the code organized as long as the tag hierarchy is meaningful. Embedded in the web browser, the program is very convenient to use and fits perfectly to the methodology. Adding a more powerful rich-text editor would even improve the power of the program.

As a tool to execute test-units, the program does not fulfill all the expectations. Running code offers two possibilities. The one is to execute test units, the other is to show the result of an example. Regarding the test units, the program might never replace features proposed by an IDE. On the other hand, executing examples seems to be a very promising feature. In that direction, ideas of example-centric programming could be applied to re-inforce the use of examples as part of the documentation of a program.

Chapter 7

Conclusion

We saw that technical documentation raises some problems when there is a need for documenting source code. In general, developers focus on their source code and consequently forget to document it. As a result, documentation is often inadequate, does not correspond to the source code or is even inexistent.

When writing software in agile development, the accent is put on tests of the system and on self-documenting code with the aim to respond faster to change and more precisely to the requirements of the client. Code refactoring, for instance, is a common operation in that direction. Documentation on the other hand can become a burden because it is time consuming to document code that will change anyway. Agile development thus recommend that other elements like tests or self-documenting source code solve these problems.

7.1 Problem

When a project becomes wider, it becomes more difficult to understand it by only reading the source code or the tests. A lot of information can be added in docstrings to document software but it becomes difficult to understand them if docstrings convey to much information, like comments, code examples and notes at the same time.

Software documentation must also be used by different people like programmers, testers or end-users but today, documentation systems do not systematically provide different views for those people. Testers do not need to display the same information as programmers and documentation should thus be presented differently according to the needs.

Methodologies do not focus enough on documentation. Arguments state that documentation is time consuming. But is it possible to improve the time taken by documentation? Programmer should be supported in the documentation process with a methodology that focus on all the aspects of programming. That methodology could then be supported by a documentation system that increase efficiency in the documentation process and lets the programmer concentrate on the important tasks to realize.

7.2 Solution

To improve software documentation and especially technical documentation in a software development methodology we proposed a documentation system that helps in different steps of the documentation phases. A possible methodology has been proposed to explain how to document software in the development phases and how the documentation system can help.

Browse entities

Our documentation system propose to browse entities of the system dynamically, when the program is running.

The advantage is that, whenever a programmer adds some features to his program, it is possible to see the entities in the documentation system. The documentation do not need to be generated and is directly available in the web-browser. Ordered listings and different pages organize these entities naturally in a Javadoc-like technical documentation to retrieve them more easily. With dynamic web pages, features like search operations are also possible.

One of the biggest concerns of the documentation system is that it is not possible to handle code change. If entities are renamed or removed, it is difficult to track those changes in and update the documentation according to the changes.

Document entities

Each entity that is visible in the documentation system can be documented with any useful information. It can be comments like in classical documentation but also code examples or any desired information that could help understanding the program that is documented.

The advantages are that it is possible to add information to the technical documentation. A text editor allows edition of the entities in a web browser, making it unnecessary to go back in the source code to update the documentation. Technical documentation is always up to date with that feature.

On the other hand, it is not possible to see the source code from the documentation. For some programmers this might be important information. The program also need to be executed to be able to navigate the technical documentation. In some cases it might be necessary to browse the technical documentation without having to start all the system. If the system is able to generate dynamic pages, it is not impossible to generate a static version that can be consulted offline.

Markup entities

Adding comments in source code might be time consuming and is not always useful. Accessor methods for instance are methods that should need less comments. It is however important to know that such methods are accessors. To avoid commenting entities, it is possible to tag them with meaningful information like their type (class, function, etc.) or their purpose (accessor, printing, converting, etc.). Some of these tags are handled automatically by the system.

A tagging feature allows to add meaningful information to entities of the system without having to comment them. This information can then be used to retrieve specific entities like accessors or printing methods easily.

The tagging feature has some limitations. There are no restrictions on the tags that can be added to an entity and this might lead to incoherent documentation. It is for instance possible to tag an entity both it "class" and "method" type, which does not make sense. The tag "undocumented" is not automatically removed when an entity has documentation.

Run code from examples

Because source code is a good way to explain how to use a system, it is possible to document the program with code examples. These code examples can then be executed to see their results. It is also possible to run unit tests the same way.

Running code from examples is a way to illustrate source code in a meaningful context and help to better understand it. Adding such information in the system and being able to effectively execute the code is a way to provide better documentation. The fact that unit tests can be run as well give the possibility to testers of the system to verify that the system is still correct.

On the other hand, it is not possible to show detailed information like the execution trace of an example. This could inform the user of the system even more. Some limitations also make the feature difficult to use properly as it is. The namespaces of the entities must be specified, which makes it difficult to write concise and readable code. Finally, support for unit tests is minimalist. Specific views should be implemented to have a better support of those features in the future.

Aggregate information

There are many entities that might be documented in the system. It is possible to aggregate the available information like comments or code examples to make new useful information in the documentation system. It is for example possible to create tutorials (step-by-step guides) by referring to different entities of the system. The documentation of those entities is then used to be displayed in the tutorial.

This feature is an important step to documentation reuse. The programmers do only need to write documentation entries once. These entries can then be used in multiple distinct tools. Tutorials show how it is possible to have one documentation entry for multiple entities.

There are still limitations to that feature. No decent entity browser is implemented and links to targeted documentation entries must be added manually.

7.3 Difficulties

The joint use of the different technologies that where used to realize the web-based documentation system like HTML, CSS or Javascript where very instructive. With that, we have been able to learn what those technologies can bring to a system. The simple structure of HTML makes it a good choice to have fast screen previews of the system to be. CSS improve the user interface. Javascript makes web-pages more dynamic. As an example, some elements like an alphabetical sorter for listings have been proven very useful when using the documentation system.

On the other hand, we found some limitations to web technologies. HTML, for instance, does not allow to describe menus or user operations and a combination of HTML and Javascript is necessary, which makes the system more complex to implement. Moreover, the joint use of three technologies is not interesting for elaborating software models that can greatly evolve over time. Whenever a change occur, it is not only the server code that must be changed but also the client code like HTML, CSS and Javascript. At the end, that solution prove to be a significant loss of time in this case. It was however an instructive learning case.

7.4 Further work

We saw that a model of dynamic annotation system can bring improvements in software documentation at this early stage of development but lot of research still remains in that field.

Correct and meaningful organization of tags is crucial and deserves more consideration so that each tag that documents an entity gives more sense to it. Researches should indicate which tags are meaningful and how they must be organized to get the most information of them.

The documentation system should also better support multiple views, with some nearer to an IDE, allowing to resolve encountered problems like navigation and execution of tests or an entity browser to link entities and documentation more easily.

Documentation entries are saved with a name that is managed by the user. A work in defining the appropriate names in which those entries might be classified could deserve more consideration so that each user can decide the information that need to be displayed, like tests, notes, comments or examples.

A more complete editor would improve edition of the documentation and the visualization of the results. The editor would also support different syntax in an optimized way to allow users to write with their favorite syntax.

7.5 Final word

We saw that software documentation is often a problem and that Agile methodologies do not focus enough on solution to documentation that responds to change. There is certainly no silver bullet. It is however possible to propose a methodology that takes documentation into account. With that methodology, supported by a dynamic documentation system that helps organizing code, ease information retrieval, and accelerate software documentation by tagging entities, embedding examples and unit tests, it becomes possible to achieve better software documentation. That documentation system should be to documentation what an IDE is to source code.

Bibliography

[1] [Seibel, 2004] Peter Seibel, "Practical Common Lisp", Apress, 2004

[2] [Beaird, 2007] Jason Beaird, "The principles of beautiful web design", sitepoint, 2007

[3] [Bushmann, Meunier, Rohnert, Sommerlad, Stal, 1996] F. Buschmann & R. Meunier & H. Rohnert & Peter Sornmerlad & M. Stal, "Pattern-oriented software architecture - A system of patterns", Wiley, 1996

[4] [Gamma, Helm, Johnson, Vlissides, 1995] E. Gamma & R. Helm & R. Johnson & J. M. Vlissides"Design Patterns - Elements of Reusable Object Oriented Software", Addison-Wesley, 1995

[5] [Mens, González, Cádiz] Sebastián González & Kim Mens & Alfredo Cádiz, "Context-Oriented Programming with the Ambient Object System"

[6] [Costanza, Hirschfeld, 2005] Pascal Costanza & Rober Hirschfeld, "Language Constructs for Context-oriented programming", ACM, 2005

[7] [Mens, González, Heymans, 2007] Sebastián González & Kim Mens & Patrick Heymans , "Highly dynamic behaviour adaptability through prototypes with subjective multimethods", ACM, 2007

[8] [Bykov] Vassili Bykov, "Hopscotch: Towards User Interface Composition"

[9] [Fowler, Highsmith, 2001] Martin Fowler & Jim Highsmith, "The Agile manifesto", Software Development magazine, 2001

[10] [Knuth, 1984] Donald E. Knuth, "Literate Programming", The computer journal, 1984

[11] [Edwards, 2004] Jonathan Edwards, "Example Centric Programming", OOPSLA, 2004

[12] [Perrin, 2006] Chad Perrin, "Making manpages work for you", TechRepublic, 2006, `http://articles.techrepublic.com.com/5100-10878_11-6102898.html`

[13] [Jeffries, 2001] Ron Jeffries, "Essential XP: Documentation", XProgramming.com, 2001, `http://www.xprogramming.com/xpmag/expDocumentationInXP.htm`

[14] [Ziemer, 2002] Sven Ziemer, "An architecture for web applications", 2002

[15] [Hale, 2006] Kevin Hale, "Ajax wireframing approaches", ParticleTree, 2006, `http://particletree.com/features/ajax-wireframing-approaches/`

[16] [Russell, 2008] Russell C., "Bridging the Object-Relational divide", ACM Queue, 2008, `http://queue.acm.org/detail.cfm?id=1394139`

[17] [Common Lisp wiki] `http://www.cliki.net/index` : Wiki of the Common Lisp programming language.

[18] [Steel Bank Common Lisp] `http://www.sbcl.org/` : Website of Steel Bank Common Lisp, an open source compiler and runtime system for ANSI Common Lisp.

[19] [Lisp Cookbook] "The common Lisp cookbook project", `http://cl-cookbook.sourceforge.net/`

[20] [Lisp MetaObject Protocol] "The Common Lisp Object System MetaObject Protocol", `http://www.lisp.org/mop/contents.html`

[21] [Kiczales, des Rivières, Bobrow, 1991] Gregor Kiczales & Jim des Rivières & Daniel G. Bobrow, "The art of the metaobeject protocol chapter 5 and 6", MIT Press, 1991

[22] [Hunchentoot] Edmund Weitz, `http://www.weitz.de/hunchentoot/` : A lisp web-server.

[23] [Html-template] Edmund Weitz, `http://weitz.de/html-template/` : HTML templates for Common Lisp

[24] [Elephant] `http://common-lisp.net/project/elephant/` : Elephant, A Persistent Object Database for Common Lisp

[25] [Lisp-unit] `http://www.cs.northwestern.edu/academics/courses/325/readings/lisp-unit.html` : Website of Lisp-unit, a LISP unit-test library.

[26] [Java] `http://www.java.com` : Website of the Java programming language.

[27] [Smalltalk] `http://www.smalltalk.org/main/`: Website of the Smalltalk programming language.

[28] [Ruby] http://www.ruby-lang.org : Website of the Ruby programming language.

[29] [Python] http://python.org/ : Website of the Python programming language.

[30] [Python documentation] http://www.python.org/doc/ : Documentation for Python.

[31] [jQuery] http://docs.jquery.com/Main_Page : Website and documentation of the jQuery javascript framework.

[32] [Martin, 2008] Jeremy Martin, "Building your first jQuery plugin", 2008, http://blog.jeremymartin.name/2008/02/building-your-first-jquery-plugin-that.html

[33] [Martin, 2008] Jeremy Martin, "Efficient tag cloud algorithm", 2008, http://blog.jeremymartin.name/2008/03/efficient-tag-cloud-algorithm.html

[34] [jQuery example] http://docs.jquery.com/Tutorials:How_jQuery_Works : An example that shows how jQuery works.

[35] [WMD] http://wmd-editor.com/download : Markup editor in jQuery.

[36] [Markitup] http://markitup.jaysalvat.com/home/ : Markup editor in jQuery.

[37] [The dojo toolkit] http://www.dojotoolkit.org/ : Webiste of the dojo toolkit javascript framework.

[38] [script.aculo.us] http://script.aculo.us/ : Website of script.aculo.us javascript framework.

[39] [Yahoo User interface library (YUI)] http://developer.yahoo.com/yui/ : Website of the Yahoo User interface library javascript framework.

[40] [Google web toolkit] http://code.google.com/intl/fr/webtoolkit/ : Website of the Google web toolkit that transforms Java into Javascript to generate Ajax applications.

[41] [Graham, 2005] Paul Graham, "Web 2.O", 2005, http://www.paulgraham.com/web20.html

[42] [Hypertext Markup Language] http://www.w3.org/MarkUp/ : W3C website for the Hypertext Markup Language

[43] [Cascading Style Sheets] http://www.w3.org/Style/CSS/ : W3C website for the Cascading Style Sheets.

[44] [CSS selectors] http://www.w3.org/TR/CSS2/selector.html : W3C website for the CSS selectors.

[45] [Document object model] http://www.w3.org/DOM/ : W3C website for the Document object model.

[46] [XML User Interface Language] http://www.mozilla.org/projects/xul/ : Website of the XML User Interface Language

[47] [Extensible Application Markup Language] http://msdn.microsoft.com/en-us/library/ms752059.aspx : Microsoft's msdn website for Extensible Application Markup Language.

[48] [Unified Modeling Language] http://www.uml.org/ : Website of UML (Unified Modeling Language).

[49] [Python docstrings] http://www.python.org/dev/peps/pep-0257/ : Python's enhancement proposals about docstrings

[50] [Javadoc] http://java.sun.com/j2se/javadoc/ : Website of Javadoc, a Java API documentation tool.

[51] [Doxygen] http://www.stack.nl/~dimitri/doxygen/ : Website of Doxygen, a documentation system for C++, C, Java and many more.

[52] [Eclipse] http://www.eclipse.org/ : Website of Eclipse, an IDE for Java.

[53] [VisualWorks] http://www.cincomsmalltalk.com/userblogs/cincom/blogView : Website of VisualWorks, an IDE for Smalltalk.

[54] [Donald Knuth] http://www-cs-faculty.stanford.edu/~knuth/ : Donald Knuth's home page.

[55] [Agile manifesto] http://agilemanifesto.org/ : Website of the Agile manifesto.

Wikipedia

[56] [Literate programming] http://en.wikipedia.org/wiki/Literate_programming : Literate programming on Wikipedia.

[57] [Software Architecture] http://en.wikipedia.org/wiki/Software_architecture

[58] [Software Design] http://en.wikipedia.org/wiki/Software_design

[59] [Documentation logicielle] http://fr.wikipedia.org/wiki/Documentation_logicielle

[60] [Lisp] http://fr.wikipedia.org/wiki/Lisp

[61] [Javascript] http://fr.wikipedia.org/wiki/JavaScript

[62] [Hypetext Markup Language] http://en.wikipedia.org/wiki/HTML

[63] [Cascading Style Sheets] http://en.wikipedia.org/wiki/Cascading_Style_Sheets

[64] http://en.wikipedia.org/wiki/JavaScript

[65] [Document object Model] http://en.wikipedia.org/wiki/Document_object_model

[66] [Asynchronous JavaScript and XML] http://fr.wikipedia.org/wiki/Asynchronous_JavaScript_and_XML

[67] [Web 2.0] http://en.wikipedia.org/wiki/Web_2.0

[68] [XML User Interface Language] http://en.wikipedia.org/wiki/Xul

[69] [Extensible Application Markup Language] http://en.wikipedia.org/wiki/Xaml

[70] [Agile software development] http://en.wikipedia.org/wiki/Agile_software_development

[71] [Agile manifesto] http://en.wikipedia.org/wiki/Agile_Manifesto

[72] [Literate programming] http://en.wikipedia.org/wiki/Literate_programming

[73] [Intensional programming] http://en.wikipedia.org/wiki/Intentional_programming

[74] [Test-driven development] http://fr.wikipedia.org/wiki/Test_Driven_Development