Developing an experiment workbench to study software reuse from a cognitive perspective

Fabrice Retkowsky

November 2, 1998

Abstract

Software reuse, as a promising programming technique, has led to many technological developments. But it also involves programmers' cognition, and different theories compete as to how code reuse should be assisted by a computer tool. We decided to develop a reuse workbench, made of a reuse tool and an experiment toolkit, to simplify the evaluation of these various theories. We describe how we complemented the initial pen and paper design of the reuse tool by a simple experiment.

1 Introduction

Software reuse is an increasingly popular aspect of programming. In the face of the demand for an ever increasing volume of code, reusing code components to build new programs seems to be a good solution (?). Besides, reusing code brings the promise of improvement from a qualitative point of view. New technologies have been developed to support software reuse, such as object-oriented programming, database repositories, component distribution systems, etc.

Yet code reuse (like programming in general) raises another kind of issue. When a programmer wants to reuse an existing piece of code, he has to express his requirements, to identify a suitable component, and then to understand it, modify it, and integrate it in a new program. All these processes strongly involve the programmer's cognition: his memory, his understanding, his mental models, and so on.

Hence ? (?) consider that "the psycho-ergonomical approach can help supporting the design of help techniques for the search, selection and specialization of reusable assets", techniques which will be "as compatible as possible with the activity and the thought process of software engineers".

In this paper. we will explain why we decided to develop a workbench for conducting experiments on the cognitive aspects of software reuse, and how we supported its design with a simple experiment.

2 An experiment workbench for software reuse

2.1 Why a workbench ?

Many aspects of cognition are involved in software reuse, and many different theories or exist for each one of them. For example, data-flow charts, control-flow charts, and textual descriptions can be used to represent programs. But which one of them is the most suitable, the most efficient? And do their efficiencies vary depending on the type of software components they are applied to?

As a consequence, we are developing a tool which is aimed at running experiments in software reuse. This workbench will be made of two parts: a module-based software reuse tool, and an experimental tool-kit. In this paper we describe the development of the reuse tool.

? (?) used a similar approach to evaluate four software visualisation tools. He embedded the four visualisation techniques into one single environment, the Prolog Program Visualisation Laboratory (PPVL), hence unifying the user interfaces, and then conducted some experiments to test the techniques' efficiency.

2.2 Initial design of the reuse tool

The idea behind the software reuse tool is to give access to a database of reusable Java components. To allow simple experimentation, it will be based on different sets of exchangeable modules. For example, the search technique, the database and the documentation techniques will all be different slots, into which alternative modules can be fitted.

The tool is based on a four-stage reuse model. First, the user will look for a component in a database by browsing or using a search engine. Then, he will seek more detailed information about the chosen component and try to understand it. Finally, he will try to specialize it, and then integrate it into his own program.

We decided to focus on the Java language for four reasons. First, Java databases of components are freely available on the Internet. Second, it is object-oriented. Third, it is very popular and there is a large pool of programmers as potential experiment subjects. Finally, the tool itself will be developed in Java.

2.3 Supporting the design by a simple experiment

To assist the design process, we decided to conduct a simple experiment that would highlight the most important issues that a reuse tool should tackle. This meant having programmers perform a simple reuse activity with a simplistic reuse system, observe their performance, and tell us what their problems were and what they thought of reuse tools. By using novice programmers, who did not have any real experience with reuse, we mainly looked at the initial problems that programmers encounter when trying to reuse.

This experiment was designed to demonstrate how well novice programmers could perform with a rudimentary reuse tool. It will serve later as a benchmark against which we can compare more advanced reuse tools. Using beginners in

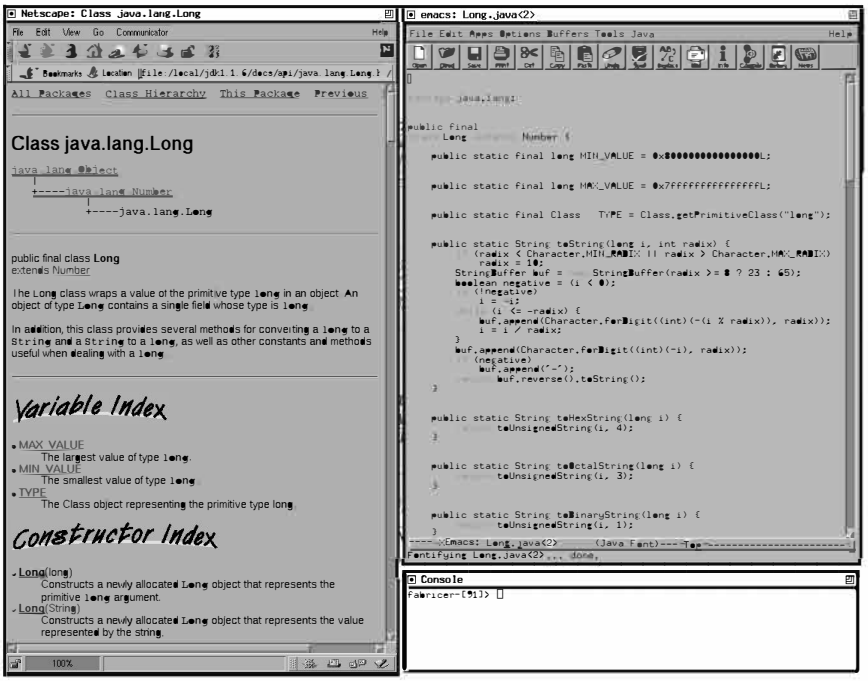

Figure 1: Experimental setup

reuse, and doing later on the same kind of experiment with expert programmers, will also help us understand how programmers improve their reuse skills.

? (?) conducted a similar experiment, where novice systems analysts had to perform a requirements analysis task. The authors used protocol analysis, where oral information was used to understand the reasoning behaviour of the subjects. In this study we focus much more on the interaction between the subjects and the computer system, and try to see which factors can influence both this interaction and the performance of the subjects.

? (?) conducted two experiments about novice programmers and code reuse. Though the first one was focused on program understanding during reuse, the second studied how confidence in software reuse can influence the whole act of reusing some code. Here we are interested in how some technical and cognitive (rather than sociological) aspects influence code reuse.

3 The experiment

3.1 Design and materials

The experiment consisted in asking 12 Java beginners to program a simple class by reusing a class from the Java API packages.

The Java API packages are standard packages of classes, written by Sun (the creators of Java). Though they were particularly aimed at being reused by

traditional object-oriented programming techniques (inheritance, class composition, method calls), the source code of these classes is available as well, so that code reuse is possible. We only had to remove from the classes the JavaDoc commands which were making the code longer and more cryptic. Furthermore, there is freely accessible documentation on the Web, in HTML. Hence, by supplying an Internet browser (Netscape Navigator in our case), the source code, and a text editor, we had a simple, rudimentary software reuse system (Figure 1).

Each subject had to perform two tasks amongst the 4 tasks available (Table 1). These four tasks are two different programming problems (programming a PhoneNumber class or a PhoneNumberFormat class). Each one of these problems (A or B) existed in two versions. In the first version (1), the problems were described without any kind of context, and reuse was, as much as possible, forced. In the second version (2), reuse was only suggested, and the class to program had a context: the subjects had to complete a large program in which a class was missing. Each subject either performed A1 and B2 or A2 and B1, in balanced order. The tasks descriptions can be found in Section 7.

	PhoneNumber A	PhoneNumberFormat B
Forced reuse, without context	A1	B1
Suggested reuse, with context	A2	B2

Table 1: The four tasks

The PhoneNumber (A) and PhoneNumberFormat (B) tasks were designed to be different from both a programming and a reuse point of view, so that the results would be less task-dependent. The PhoneNumber problem is a classic datatype problem, fairly easy to solve, and for which many reusable components exist in the API packages. By comparison, the PhoneNumberFormat class is a very short, but more original problem, probably more difficult to understand for beginners. There are some perfectly suitable components to reuse for this problem, but they are quite well 'hidden' in the API documentation, so that they are difficult to find out. Thus we predicted that A would be slightly longer to perform, though much simpler, and that it would lead to more and better reuse.

3.2 Subjects

The subjects were first-year undergraduates from the School of Cognitive and Computive Science (COGS) at Sussex University. They were either studying Computing Science or Artificial Intelligence and Computing Science, and had attended two courses of Java programming. We had 12 subjects, each one performing 2 tasks. Hence we had 24 tasks performed, i.e. 6 tasks of each type.

3.3 Protocol

First of all, the subjects were introduced to the idea of reuse and told what the experiment was about.

For each one of the two tasks, the subjects were given the requirements of the class they had to write. They had no more than 20 minutes to complete each task. They only had access to a text editor and to a local copy of the API on-line documentation. They didn't have to actually write a complete, working class: we didn't ask them to compile their programs. While the subjects were performing the tasks, the experimenter took notes on all their interactions with the API documentation, and on the subject's comments (they were asked to think aloud).

Once the two tasks were finished, they were asked three open-ended questions about software reuse and the rudimentary reuse system they had used. If the subject reused a component in their program, the suitability of this component was then evaluated, as well as the quantity of code that was actually reused (both by mark out of 5).

When all the tests had been done, the quality of the resulting pieces of code was evaluated, by a mark out of 5. For each task, we first gave to the six programs an initial mark which took into account (from the least to the most important): the syntax errors, the shortness/clarity of the code, the programming mistakes, and the conceptual errors. We then ordered the programs and checked that the marking was consistent for this task. Finally, we compared the consistency of the marking between the four different tasks, by verifying that the best and worst programs were of the same quality in each case.

4 Results

4.1 General remarks

The subjects all followed the same pattern of programming. First, they read and tried to understand the problem description (it took 1 minute on average). Then, they looked for a component to reuse (3 minutes on average, though some subjects didn't search at all), and finally did the programming. Only two tests (out of 24) required more than 20 minutes.

The 'Index of Classes' and 'Class Hierarchy' (cf 4.2) in the documentation pages were not used at all. There might be three possible explanations: either these pages are useless, or they are nearly 'hidden', or users need more experience to use them.

4.2 API use

The Java API documentation is based on a tree hierarchy. There are basically four levels of description: the Index of Packages, the List of Classes (for each package), the Class Description (for each class of each package) which includes a list of the class' methods, and the Method Description (for each method of each class).

When searching, subjects only used the first three levels of description (Figure 2). The Method level was hardly used. When programming, they used all the four levels, including the Method level. Comparing the searching and

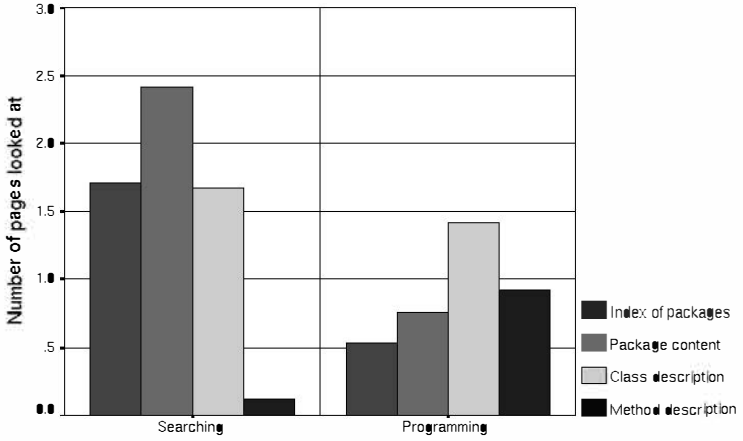

Figure 2: Use of the API documentation when searching and programming

the programming stages, the first two levels were less used for programming ($t = -2.53, df = 23, p < 0.02$ and $t = -2.08, df = 23, p < 0.5$), the third one (Class level) was used the same, and the Method level was much more used for programming ($t = 2.94, df = 23, p < 0.01$). The statistical results mentioned here are by default 2-tail t-tests.

This suggests that the Method Description level is too precise for the Search stage. A proper search tool doesn't have to display such information: it will only be necessary for the Programming stage.

4.3 PhoneNumber - PhoneNumberFormat

Variable	PhoneNumber	PhoneNumberFormat
Total time	15m 56s	16m 55s
No. of pages when searching	6.08 pages	6.00 pages
No. of pages when programming	4.33 pages	3.25 pages
Quality of the code	3.29 /5	3.29 /5
Percentage of reusers	42 %	42 %
Suitability of the components	2.80 /5	1.60 /5
Quantity of reuse	2.50 /5	0.90 /5

Table 2: Results for the PhoneNumber and the PhoneNumberFormat tasks

The PhoneNumber problem (A) was predicted to be simpler, if longer, but proved to be slightly shorter to perform (no sig.). Subjects looked at the same number of API pages for searching, but they used more API pages while programming (probably because the programming was longer). The quality of the resulting code is the same for both tasks, though the evaluation was suggestive.

We forecast that A would lead to more and better reuse. In fact as many subjects reused for both tasks. Yet, they indeed reused better for A: the com-

ponents they reused were more suitable (not sig.), and they reused more lines of code ($t = 2.5, df = 8$, 1-tail p < 0.02).

As a whole we can say that there were enough differences between the two tasks from a reuse point of view to give more credibility to the other results.

4.4 Forced reuse, without context (1) - Suggested reuse, with context (2)

Variable	Forced/NoContext	Suggested/Context
Understanding time	1m 4s	1m 35s
Searching time	5m 32s	0m 53s
Programming time	11m 35s	12m 11s
No. of pages when searching	9.92 pages	2.17 pages
No. of pages when programming	3.17 pages	4.42 pages
Quality of the code	3.00 /5	3.58 /5
Percentage of reusers	67 %	17 %
Suitability of the components	2.50 /5	1.00 /5
Quantity of reuse	1.87 /5	1.00 /5

Table 3: Results for the 'Forced reuse/Without context' and the 'Suggested reuse/With context' situations

Logically, in the forced reuse/no context situation, subjects spent less time understanding and more time searching, and they looked at more API pages while searching. They spent the same amount of time on programming for both cases. This is because the programs were too small for reuse to have any effect on programming time. The subjects also looked at about the same number of pages while programming, though they looked at fewer methods and more classes for 1. The difference of one page (3.17 compared to 4.42) comes from a technical aspect of the API pages and from the fact that most of the subjects in the situation (2) skipped the Searching stage.

With forced reuse, as forecast, more subjects decided to reuse ($t = 2.76, df = 22$, 1-tail p < 0.006). They also reused better, largely because the only two subjects who reused in the suggested reused condition did reuse poorly.

Finally, the situation (2) produced better programs, mainly because (2) led to less reuse, and programs based on reuse were judged of lesser quality (see 4.5).

4.5 Expertise

Though the subjects were all beginners, there were initially two measurements of their expertise in Java: the number of programming languages they knew (including Java), and the amount of time they had spent programming (the sum for all languages, where one year was counted as three terms). Though, the quality of the resulting programs turned out to be constant. This suggests that past programming experience did not help in performing this task.

We also used as a third estimate wether the subjects had used the API pages while learning Java ('No', 'A little bit' or 'A lot'), which denotes wether

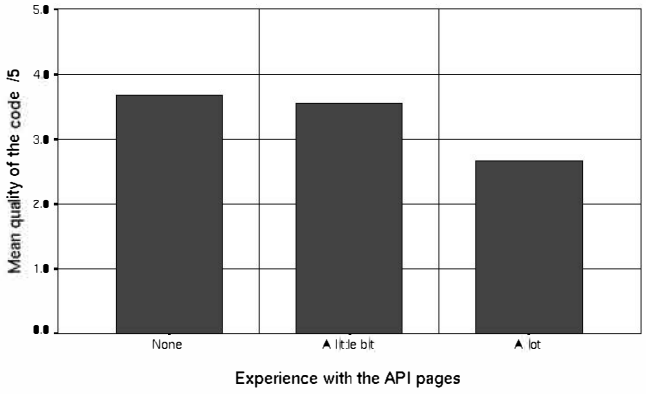

Figure 3: Expertise with the API pages has a negative effect

they were serious and/or curious about learning Java. Surprisingly, the API pages experience seems to have a negative effect on performance (Figure 3). 'A lot' of experience led to lower quality code than 'A little bit' ($t = -2.11, df = 14, p < 0.06$) and than 'Not at all' ($t = -2.67, df = 14, p < 0.2$). There are two explanations for that:

- Firstly, subjects with 'A lot' of experience with the API pages reused more often: 65% of them reused, instead of 37% for 'No' experience and 25% for 'A little bit' (not sig.). Besides, programs based on reuse were evaluated as of lesser quality than normal programs (average of 2.80/5 compared to 3.64/5, $t = 2.44, df = 22, p < 0.03$). In fact, comparing the quality of reuse-based programs and normal programs, for these simple tasks, is fairly subjective. But looking at the reuse-based programs shows that reusing led to many minor errors such as forgetting to rename the class, including the new class in a package, inheriting from a class without implementing some abstract methods, and leaving 'as-is' many useless methods. These errors are characteristic of programmers who never did any code reuse before.

- Secondly, Figure 4 suggests that 'experts' are under-performing whether they reuse or not. This is even more surprising. It might be explained by 'experts' trying to perform very well under experimental conditions, thus focusing on secondary details and not solving the main problems first.

Finally, subjects who used the API pages 'A lot' in the past looked at less pages per unit of time while searching than others, and more pages per unit of time while programming. This means that they probably read the descriptions more thoroughly when searching (as opposed to 'beginners' who just browse), and that they knew how to use the API pages as a programming help.

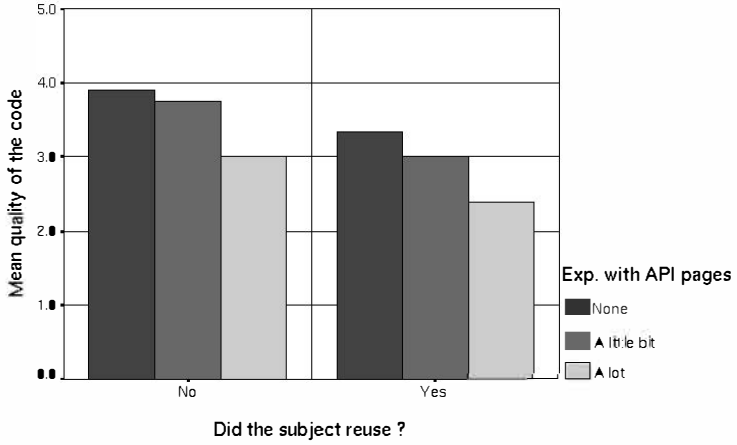

Figure 4: Experts under-perform whether they reuse or not

5 Consequences for the Design

Once the subjects completed their two tasks, they were asked three open-ended questions about software reuse:

- What are, in your opinion, the good aspects of the API pages as a reuse tool?
- What are, in your opinion, the bad aspects of the API pages as a software reuse tool?
- What should a perfect reuse tool look like?

The answers we collected can be found in Section 8. From these answers, and from the points we made in the numerical analysis, we can draw some guidelines for the design of the reuse tool. Some of these guidelines are already met by the initial design, some led to a few modifications.

Component description As we saw before, the reuse tool will be based on a set of modules. One of the most important modules is the component description. What appeared from the experiment's feedback is that the API documentation style is a good basis to start with, since:

- it includes a list of all the methods;
- the class hierarchy at the top of each class is a good thing;
- it is well structured and uniform;
- there are lots of links between classes;
- it shows the correct syntax via an example.

Therefore we will base the first version of the component description module on the API documentation. Yet the subjects suggested a few modifications:

- the class and package names should also be self-explanatory for beginners;
- it should have less technical terminology;
- it should also describe the code itself, and make it easily accessible (or even include it in the description?), particularly for the 'Understanding' stage;
- it should include some examples;
- the packages should have a description as well;
- it should be less complicated, and shorter. This is easily feasible for the search stage: the experiment proved subjects don't use the 'Methods' level.

Navigation Since navigation is an important issue, we initially designed a complete and efficient set of navigation tools. The subjects reminded us that the navigation should be very simple (i.e. like the API, in HTML), and that:

- it should provide something so that users don't get lost;
- the Search stage should actually have a search tool;
- it should always suggest alternative possibilities, so that the user does not get trapped in one not-so-good solution;
- it should assist but not be intrusive.

As a consequence, it was decided to keep the navigation tools to a minimum, that is, a bar menu and a small wizard that allows quick navigation between the four stages of reuse.

Structure Finally, the system should include some editing tools (to specialize and integrate the components) and a built-in compiler (which was lacking from the experiment's rudimentary reuse setup). These were not planned at first, but will be included in the Specializaton and Integration stages.

6 The next steps in the development of the tool

Since this experiment was completed, we have developed a mock-up of the user interface, and have had it tested by a few possible users. The next step consists in designing the experiment toolkit, and then programming the whole reuse workbench for real.

Once the system is completed, we will perform the experiment described in this paper again, but using our tool instead of the rudimentary reuse system used here. This will have two aims:

- to test wether there are any major flaws in the design of the tool, or whether subjects have any problem in using this kind of integrated tool that guides and assists them,

- and to know wether, in a simple configuration based on the API documentation, our system already brings some kind of benefits.

Finally we will develop and compare some new sets of modules, for example to evaluate alternative search techniques or documentation styles.

7 Appendix 1: The four tasks

7.1 Task A1

Write a PhoneNumber (PN) class by reusing a Java API class.

- A PN object will contain a telephone number, such as 1273275779
- It will be initialized using a String parameter, i.e. "1273275779"
- It will have a toString() method which will give back a String such as "(1273) 275779"

You HAVE to reuse a Java API class to write this class.

7.2 Task A2

Here is the PhoneList class, which is used in an 'Organizer' program.

- It is basically a Vector of PhoneNumber objects
- The 'Organizer' program creates such PhoneLists, adds PhoneNumbers to them, remove PhoneNumbers, and print the PhoneList on screen
- PhoneNumbers are created using strings, such as "1273275779"
- When the PhoneList is printed on screen, the PhoneNumbers should be displayed as "(1273) 275779"

Write the PhoneNumber class.

You can reuse an existing class file from the Java API if you want.

```
public class PhoneList
{
    int MaxSize = 3;
    PhoneNumber[] PhoneArray = new PhoneNumber[MaxSize];
    int NbNumbers = 0;
// position 1 for PhoneArray[0]

    PhoneList()
    {
// creates two default numbers
        PhoneNumber OneNumber = new PhoneNumber("1111111111");
        PhoneNumber NineNumber = new PhoneNumber("9999999999");

        this.addNumber(OneNumber);
        this.addNumber(NineNumber);
        this.printNumbers();
    }

    public boolean addNumber(PhoneNumber aNumber)
    {
        if (NbNumbers == MaxSize)
    return false;
        PhoneArray[NbNumbers]  = aNumber;
        NbNumbers++;
        return true;
    }

    public boolean removeNumber(int position)
    {
        int i;

        if (position > NbNumbers)
    return false;
        if (position == NbNumbers)
    {
        PhoneArray[position] = null;
        NbNumbers--;
        return true;
    }
        for (i=position; i<NbNumbers; i++)
    PhoneArray[i-1] = PhoneArray[i];
        NbNumbers--;
        return true;
    }

    public void printNumbers()
    {
        int i;

        if (NbNumbers == 0)
    System.out.println("Empty List");
        else
    for (i=0; i<NbNumbers; i++)
        System.out.println("Phone n. "+i+": "+PhoneArray[i].toString());
    }
}
```

7.3 Task B1

Write a PhoneNumberFormat (PNF) class by reusing a Java API class.

- A PNF object will be able to format some strings
- It will for example format "1273275779" into "Brighton 275.779"

You HAVE to reuse a Java API class to write this class.

7.4 Task B2

```
import java.awt.*;
import java.applet.*;

public class PhoneWidget extends Applet
{

// The interface attributes
    TextField input = new TextField();
    Button OK = new Button("OK");
    Label output = new Label("");

// The format
    PhoneNumberFormat theformat = new PhoneNumberFormat();

// The result
    String result;

    public void init()
    {
        setLayout(new GridLayout(3,1));
        add(input);
        add(OK);
        add(output);
    }

    public boolean action(Event evt, Object arg)
    {
        if ("OK".equals(arg))
    {
        result = theformat.format(input.getText());
        input.setText("");
        output.setText(result);
    }
        return true;
    }
}
```

Here is a PhoneWidget applet, which will be used
in an 'Organizer' program.

- A PhoneWidget object allows the user to type in a telephone number, such as 1273275779
- This number is then formatted using a PhoneNumberFormat object
- The result, in this case "Brighton 275.779", is displayed on screen

Write the PhoneNumberFormat class.

You can reuse an existing class file from the Java API if you want.

8 Appendix 2: Open-ended questions on reuse

At the end of the experiment, we asked three open-ended questions about software reuse to the subjects. Here is a summary of their answers. Each piece of answer has been included, and answers that occurred twice or more are presented as such (x 2, x 3, etc.).

What are, in your opinion, the good aspects of the API pages as a reuse tool?

General remarks

- The source code is free, easily available and frequently updated.
- It's useful for reusing large quantity of code, it gets programmers interested in reusing code.
- It's a good reference book, but nothing more.

Remarks to take into account for the design of the tool

- It's easy to use (x 3), easy to navigate (x 3).
- It's a good documentation (x 3), definitive, comprehensive list, all the methods are in (x 2), it is well structured (x 2), the descriptions are uniform, i.e. they all have the same layout. The class hierarchy at the top of each class description is a good thing. There are lots of links between classes.
- lots of the names are self-explanatory. It shows correct syntax.

What are, in your opinion, the bad aspects of the API pages as a software reuse tool?

Education is necessary for good software reuse

- It's difficult to use for 1st time programmers or users (x 2). It lacks some explanation of how the whole documentation works.
- It's difficult to use general code for specific purposes. It's quicker to write short new classes than long general/abstract class where you have to delete huge parts. It's only available on the Internet.

Remarks to take into account for the design of the tool

- It's easy to get lost, and difficult to find something you don't know exist. It lacks of search facility (x 3).
- The documentation is too complicated sometimes, too long. There is some heavy, technical terminology (x 2). There's no description of what packages are associated with. It lacks examples (x 4). The descriptions don't say anything about code, it's only for functions calls.

What should a perfect reuse tool look like?

General remarks

- It should be easy to navigate through (like a browser, but not necessarily using HTML), and to find something that suits you (x 2).
- It should have a GUI for today's level of programming, icons, images you can relate to (x 2), similar to Win95, point and click.
- It should make some provisions for first time users, be off-line, and be free
- It should support a whole range of languages, not just Java, have some links to other software reuse sites, and be cross-platform (x 2). It should be possible to add new (documented) classes.
- It must create templates for the most frequently used code.

Remarks to take into account for the design

- It should have some easily understandable descriptions, with the complete list of all the methods and variables, and the methods should point at other methods which could be of use. The descriptions must use plain English, which is quicker to understand (*this has been proved wrong in some studies*). It should include the actual code as well as the description.
- There should be an (intelligent) search engine which would suggest components (x 4). Something like dBASE IV. It should always propose different solutions, so that the user does not get trapped in one not-so-good solution. It should assist but not be intrusive.
- It should include some editing tools (i.e. a simple way of taking the code of a component and modify it), and a built-in compiler.