Software reuse from an external memory: The cognitive issues of support tools

Fabrice Retkowsky

November 12, 1997

Abstract

While early reuse techniques were based largely on the programmer's memory, more recent techniques give the programmer access to a library of existing programs or models. The problem arises of how to use these 'external memories': that is, how to structure the memories and their access methods effectively.

To study this problem from a cognitive point of view, we have to know how the programmer deals with programs, and more precisely, what knowledge a programmer has of a program when he reuses it. There are different models of those internal representations of programs. This paper looks at how programs are internalized (i.e. how this knowledge is build), and how programmers externalize their representations of programs.

The problem is made more difficult by the fact that software reuse is made of four steps (finding, understanding, specializing and integrating a component), which differ slightly from one reuse technique to another, and which involve different cognitive processes. Each step can be supported by a specific tool, and knowing which cognitive processes are involved should help us in optimizing those tools.

We suggest that a modular reuse system should be developed. The aim of this system would be to make comparisons between different theories on some important aspects of software reuse.

Contents

1 Introduction

Software reuse is an activity which includes many different reuse techniques, each one having its own specific properties, though, we can isolate a trend in the evolution of these techniques.

Early reuse was based on the programmer's memory: the basic principle was to recall a program written in the past, and then use it as a first approximation of the solution of the new problem. The major limitation of this kind of reuse is the programmer's memory. By giving access to a database of existing programs or models (such as a source code component library, a design component library, or a book of classical programs), this limitation can be removed: a programmer can then look at a large number of programs, and choose which one may be useful.

Yet the use of an external memory raises some cognitive issues. For example, how easy it is for the programmer to find a program that may be useful and then modify it for its new use.

In the first section of this paper we describe the most important reuse techniques, highlighting the distinction between internal memory based reuse and external memory based reuse. We then look at which important cognitive concepts are involved in the reuse activity. More precisely, we describe what is known of the internal and external representations of programs, as well as the internalization and externalization processes.

We see in the third section that reuse activity can be split into four different steps: finding, understanding, specializing and integrating a component. As such, we examine how each one of these steps, which can slightly vary from one reuse technique to another, relate to those four essential cognitive concepts. Finally, we study how far the designers of existing support tools have tried to tackle those cognitive issues and offer guidelines for the development of new tools.

2 From internal reuse to external reuse

Software reuse is a general term that includes many different techniques. For example, consider the case of a programmer who needs to program something that he (or somebody else) has already programmed, or alternatively who needs to program many instances of more or less the same program. He can either write a first version of the program, and copy/paste it where necessary, or scavenge some part of an existing program, or reuse a function which he found in a library, etc. Each one of these alternatives is a reuse technique of its own.

Here we will briefly describe the most common reuse techniques, by dividing them in two categories: internal memory reuse and external memory reuse.

2.1 Internal memory reuse techniques

2.1.1 Write/Copy/Paste

In this situation, the programmer has to write several instances of more or less the same program. He decides to write the first instance and then to copy/paste it for further instances, modifying it if necessary.

Détienne [4] describes this as 'new' reuse, as opposed to scavenging 'old' code. She suggests that the programmer first builds a high-level mental representation of the generic solution, and implements it in the first instance. Then, considering both the mental (high-level) representation and the first (low-level) instance, he implements a second instance, and so on.

The Write/Copy/Paste technique has not been studied a lot yet, possibly because of its small-scale aspect.

2.1.2 Code scavenging

The Code Scavenging technique is used when a programmer copies a part of the code of an existing program, modifying it if necessary. Hoadle [10] describes Code Scavenging as 'code cloning', as opposed to 'code invocation' (using code components, see 2.2.1) or 'patterns' (design pattern components, see 2.2.3).

In fact, we can make a distinction between two kinds of code scavenging. On the one hand, the programmer may reuse a program that he has written himself in the past, based on his memory or the listing.

On the other hand, the programmer may be given from the beginning a program to reuse that he did not write himself. This situation is usually the one studied in experiments, but also occurs in real life, for example when a programmer has to completely rewrite an existing program. In this situation, the programmer does not know anything from memory of the program, but has one hopes some kind of documentation, as well as the code listing itself.

2.1.3 Design scavenging

Rather than reusing code, programmers sometimes reuse code abstractions, for example when they just remember the structure of a program they wrote in the past. They can then decide to reuse the same general methodology (the design) without looking at the code itself (the implementation). For instance, Krueger [11] noticed that sometimes "a large block of code is copied, but many of the

internal details are deleted while the global template of the design is retained". Then, "the developer can directly reuse these abstractions in the new design".

We can make a distinction between two situations. On the one hand, the programmer may simply remember how he solved a particular problem in the past, and decide to reuse the same method. This can be seen as a 'memory' situation. On the other hand, he may have a large piece of code to reuse, and not reuse it with all its precise details, but just reuse the design. This can be seen as a 'listing' situation.

2.2 External memory reuse techniques

As we said before, external memory reuse is characterized by the use of some organized stores of specially designed, reuse-oriented components. Such a store can for example be a database of reusable components, or simply a book of example programs. McIlroy's [15] was the first to propose an industry of 'off-the-shelf' source code components.

2.2.1 Source code component reuse

Source Code Component Reuse (SCCR) deals with the reuse of code component, which are stored in a library. Library, once again, has a broad meaning, including databases of pieces of code, books of example programs, etc.

But the basic principle is that some software engineers first have to write the code components whose basic aim is to be reused. For example, they can be designed as part of a current project. Instead of writing a piece of code specific to their own problem, software engineers may sometimes realize that they can make it reusable in other situations, by other programmers, and decide to write a reusable component instead. Alternatively, some organizations may setup a group of programmers whose task will be to look at the software produced by the whole organization, and try to turn some of it into reusable artifacts, by putting them into a library and documenting them.

2.2.2 Software architectures

The principle of software architectures is that it is possible to map high-level problem descriptions or designs to existing architectures or implementations. By looking at some existing systems, inside a strictly defined domain, it is possible to identify the most common architectures. Then, by describing one's problem, an automated system should be able to select the most suitable architecture, to adapt it to one's particular needs, and to compile it automatically into a complete implementation. However, this software architecture reuse technique, described by Neighbors [16] [17] and Krueger [11], has not been adopted widely, perhaps because to design such a software architecture proved to be very difficult.

As a matter of fact, SAs now refer to the complete designs of some existing system, which can be adapted to a new problem, yet without the idea of an automated compilation process. Only the structure of the system is reused. As far as large systems are concerned, programmers consider that what is more important is the design of the system. From a well conceived design, the implementation is straightforward.

2.2.3 Design patterns

Design Patterns (DPs) are high-level representations of common aspects of designs. A pattern is a small structure of a few objects, with an accurate name and description.

Gamma, Helm, Johnson and Vlissides [6] made a list of 23 basic patterns, where each pattern is defined by about ten attributes. These attributes describe the structure of the pattern, the issue addressed by this structure, and the consequences of integrating the pattern into a system. They underlined the fact that the aim of design patterns is to "program to an interface, not an implementation" and to "favor object composition to class inheritance".

Design Patterns may succeed were SCCR failed. The original idealistic view of software reuse was based on the concept of 'building blocks', but the code level proved to be too complicated to allow this. By contrast, Design Patterns are in theory very simple to use: it is basically a 'selecting then assembling' task. Therefore, it seems possible that DPs could allow reuse to be much closer to the original 'building block' ideal. That probably explains why DPs have been so successful amongst researchers, and have focused so much interest.

As we have seen, the restriction of the programmer's own memory has been removed by developing reuse techniques which are based on external memories. Now the problem arises of how to use those external memories. While early reuse naturally involved the programmer's thoughts and his internal memory, we are now in a situation where the programmer has to relate to an external device which extends or even replaces his internal memory. This may lead to some sort of disruption of the reuse activity, and we should study how this gap can be reduced.

3 Internal and external representations of programs

Considering that the issue of using an external memory involves such concepts as memory, perception and understanding, it seems logical to study this problem from a cognitive point of view. Four major points should be studied:

- *Internal representations of programs*

  We need to know what knowledge a programmer has of a program when he reuses it. This includes identifying the knowledge itself as well as understanding its structure and organization.

- *External representations of programs*

  We also need to look at how programs can be described and documented, e.g. using the code listing, textual descriptions, or diagrams.

- *Internalization*

  Besides, we should look at how programs are internalized, that is how the internal representations are built. This may depend on the task the programmer is performing, and of course on the nature of the program, as well as its external presentation.

- *Externalization*

  In parallel, we should look at how programmers externalize their internal representations of programs, be it for programming or to describe a program. This may give us some information on the programming knowledge of programmers and on which external representations of programs they naturally use.

3.1 Internal representations

We can make the distinction between two types of programming knowledge: general programming knowledge, which is used to write, debug and understand programs, and knowledge of past programs, that is, internal representations of programs that are being (or have been) used.

3.1.1 Programming knowledge

An influential view, including the work led by Soloway [22], suggests that programming knowledge is made of two components:

- *Programming plans*

  These are small, language independent, units of code that perform a single basic task. For example, looking through a list of numbers is a programming plan.

- *Programming discourse rules*

  These rules guide the programmer on how to put different plans together. For example, by putting a 'compute the average' plan into a 'look through a list of numbers' plan, a programmer can compute the average of a list of numbers.

As a programmer grows in expertise, he learns more and more programming plans and discourse rules. However, Davies [3] showed that expertise does not only come from the amount of knowledge. Where intermediate programmers differ from novice programmers on the amount of plans and rules they know, experts programmers differ from intermediate programmers by the structure of their knowledge.

Indeed, experts can identify in a programming plan what is called a focal line. This focal line is thought to be the most important one, the line that sums up what the plan is doing. For example, in a 'compute the average' plan, which adds up different numbers and divide the total by the number of numbers, the focal line is the *average = total / number* line.

This theory of programming knowledge was based on Pascal-like languages. This supposedly includes most of the high-level languages (C, C++, Java, Ada, etc.). Some studies showed that the existence of plans for Basic or Fortran is not obvious.

3.1.2 Internal representations do exist

While the existence of general programming knowledge is a clear fact, the existence of internal representations of programs written by the programmer in the past has been a controversial issue. In this paper we make the hypothesis that programmers do have some internal representations of their programs. For example, it is clear that programmers are able to recall aspects of previous programs that they have written and perhaps the techniques they employed to derive these programs.

The work of Hoadley, Linn, Mann and Clancy [10] tends to support this hypothesis. They conducted an experiment where subjects were asked to look at some functions, before solving a few problems where they could in fact reuse some of these functions. Reuse occurred more often when the subjects were previously asked to summarize the functions. Reuse was even more probable if they happened to summarize them in an abstract way. This suggested that programmers indeed build some internal knowledge of programs when they try to understand them, those representations being more or less abstract.

3.1.3 Models of internal representations

Before looking at some models of internal representations of programs, it is interesting to notice that memory can be characterized as episodic or semantic (Ormerod, [18]). While episodic memory stores information about particular events, semantic memory stores global, general, abstract information. For example, general programming knowledge, which is either learnt in an abstract form or abstracted from different situations, is commonly considered as being stored in semantic memory. Alternatively, we can wonder whether internal representations of past programs are semantic or episodic knowledge. It would appear natural to regard code-level representations as largely episodic in nature while design-level representations are in part semantic in nature.

Semantic networks Semantic networks are a representational system of long-term memory. In semantic networks, knowledge is symbolized by 'cells', which hold one precise topic (or word, meaning, object, etc.). These cells are linked

together by many links which denote relations between topics, and which can be activated. Thus, starting from one topic (bird), we can go to other topics (animal, feathers, to fly), and again to other topics (plane), and so on.

Ormerod [18] describes how schema based knowledge (such as programming plans) might well be seen as being based on semantic networks. Since a plan is a piece of abstract, semantic knowledge, it may be associated to a particular node. The question is whether the same thing can be said of internal representations of programs, which as we have said before includes not only semantic but also (and mainly) episodic knowledge. This, as far as we know, remains to be studied.

The internal pseudo-language of experts One way of trying to understand a programmer's internal representations of programs is to look at the most instinctive, natural, elementary way in which he expresses them.

Petre [21] described how programmers "solve problems [...] in a private, pseudo-language that is a collage of convenient notations from various disciplines, both formal and informal", which "might be taken as the surface reflection of the expert's computational model". A program expressed in this language seems to be an "assemblage of overlapping and possibly incoherent fragments". This sounds quite similar to the idea of abstract, language-independent programming plans assembled by various programming rules.

As it is, this could suggest that programmers indeed represent programs internally using plans and rules. But the problem here is that these remarks are more about how programmers create programs (using plans and rules as we said in 3.1.1), rather than how their internal representations (such as their memories of past programs) are structured.

Program understanding von Mayrhauser and Vans developed, as we will see in 3.3, a complex theory of program understanding. According to this theory, programmers switch between three internal models (the top-down model, the program model and the situation model) when trying to understand a program. Therefore, we may think that these three internal models may be some kind of internal representations of programs.

But again, the problem is that this is basically a description of the knowledge that a programmer has of a program when he tries to understand it. But will he use the same internal representations when he remembers or recalls a program he has worked on in the past?

The fact is that, for now, we do not know a lot about internal representations of programs. There are many conjectures and hypotheses, but proven facts are rare. We would particularly need more information about the internal representations of old programs.

3.2 External representations

Here we present some external representations of programs, and how they can be accessed.

3.2.1 External representations

Code Code listing is the basic way of representing a program. It is the representation actually used to create the program, and therefore the best known by programmers. Code can be 'enhanced' by adding some more information by putting spaces between different functions, indenting each instruction blocks to the left, putting keywords in a different colour, and so on.

Textual description Text can be added to the code listing, by way of embedded comments or on separate pages. This allows the programmer to really describe not only his program, but also the requirements, analysis and design.

Graphical representations Many graphical representations have been developed to support program description. They range from simple diagrams to complex data-flow or control-flow diagrams, as well as structure diagrams. Besides, many alternative sets of more or less meaningful and intuitive symbols exist. But the problem is that, for now, no study has proven that one of these graphical representations brings some clear benefit over code or text description. As Green [9] said, "there are many more diagrammatic possibilities being touted by their supporters, than there are investigations".

3.2.2 Access techniques

Browsing Browsing was developed to navigate through large documentations, be they made of code, textual and/or graphical descriptions. The idea is that, by clicking on a word, the user can go to a part of the description which is related to this word. This system is usually called Hypertext, and it is widely used in the World Wide Web on the Internet. Furthermore, a search tool can be provided, so that the user can find out where one particular word is used in the documentation.

Intelligent agents von Mayrhauser and Vans [24] highlighted the fact that, since program understanding seems to be based on hypothesis testing, a program description system should include the possibility for the user to ask questions, i.e. to test hypotheses. This can be considered as a plea for intelligent agents to be implemented in such systems. The agent would be able to help and guide the user through the documentation, and to assist him in understanding the program. Ideally, the user could feel like he is interacting with the original programmer himself.

3.2.3 What should be displayed?

A lot of different types of information can be provided using text. Brooks [1] made a list of possible aspects to describe, not without noting that more description is not always better. Similarly, many different diagrammatic representations exist, such as control-flow, data-flow and structure diagrams. Their efficiency still remains to be proven. Though this issue will be studied more thoroughly in the following sections, we can recall two systems of representation which have been developed to describe programs.

The Description Level Green, Gilmore, Blumenthal, Davies and Winder [8] developed the cognitive dimensions theory, which defines a collection of different cognitive aspects of programs, such as viscosity, premature commitment and perceptual cueing. One of their conclusions is that object-oriented programming systems, instead of relying so much on code and object relationships, should include a Description Level. This Level could include the Facets approach (Diaz and Freeman, [5]), which gives a list of aspects upon which classes and functions can be described, and the Use of Multiple Views, which takes into account other relationships than simple code relations (e.g. class inheritance), such as chronological and design-based ones.

The integrated meta-model Alternatively, we can consider again the theory of understanding developed by von Mayrhauser and Vans [24]. As we said, it is based on three internal models of programs, and they argue that the information relevant to each one of these models should be provided, as a presentation system "should support the cognitive processes of understanding, not hinder it". Though, they don't explain how this knowledge should be externalized, that is which external representations should be used.

As we have seen, there are many different ways of presenting information. Though no study has yet proven that one given external representation has a definite overall advantage over the others, it seems logical to think that there are cognitive differences between them.

In fact, what seems the most important at first is to find what precise knowledge should be displayed. Then, a precise study could be led to compare how various representations can display this type of information. The theories of Green et al. and von Mayrhauser and Vans could lead to such experiments.

3.3 Internalization

The internalization of programs is basically program comprehension, that is, the process of understanding a program using different sources, i.e. different external representations.

Soloway [22] argued that programming plans and discourse rules play a strong role in program comprehension. For example, he showed that expert programmers often make strong assumptions about programs, since they think that they all follow some common discourse rules.

By contrast, Pennington [20] explains that programming plans just play a role in program comprehension and explanation, but that they are not the memory structure. The role of plans in internal representations of programs has not been, in her opinion, proven. As we see it, this confirms the distinction we made between general programming knowledge (which indeed seems to involve plans and rules) and internal representations of programs.

3.3.1 Models of program comprehension

Brooks Brooks [1] described a model of program understanding based on hypothesis testing. To understand a program, the programmer has to make a

global hypothesis on what the program does. Then he can split this hypothesis into sub-hypotheses, and so on, until the hypotheses attain a level of detail which make them comparable to the documentation available, e.g. code, texts, etc. A low-level hypothesis can thus be confirmed or disconfirmed by the documentation, in which case this hypothesis and its neighbours will have to be modified. This model is particularly focused on domain knowledge, or more precisely at how the program deals with the domain knowledge. Domain knowledge helps the reuser in formulating the hypotheses, and as a consequence, most hypotheses will be targeted at learning this domain knowledge.

Top-down model This model, described by Soloway, Anderson and Ehrlich, is quite similar to Brooks' one. The originality comes from the fact that they identify three types of programming plans: strategic plans (global methods used by the program to achieve a goal), tactical plans (local methods used to achieve a local goal) and implementation plans (language dependant implementations of a tactical plan). Here the focus is on goals and plans since the internal representation of the program is a decomposition of the overall goal in different plans, which are in fact local goals that are achieved by other sub-plans, and so on.

Bottom-up model The theory developed by Pennington is based on the building of two internal models of programs: the program model (which is a control-flow model abstracted from the code by spotting plans and independent chunks) and the situation model (which is a data-flow abstraction based on hypotheses which are checked on the program model). The reuser first begins to build the program model by analyzing the code. Then he can start building the situation model, which requires some domain knowledge. It is completed once the top-level hypothesis, that is the program goal, is reached.

Integrated meta-model von Mayrhauser and Vans [24] built a theory which integrates the top-down model, the program model, and the situation model. Its fourth basic component is the knowledge base of the reuser. During the comprehension process, the reuser unconsciously switch from one model to another depending on what he has just found or understood.

We can see that these models are focused on code understanding, without looking at design understanding. Besides, if some provisions are made for the use of different types of documentation, they do not actually address this issue.

They confirm that program understanding depends on the subject's task (maintenance, debugging or reuse will not require the same level of understanding) and on his knowledge of the domain and of the program itself.

Finally we can notice that what results from the comprehension process is an initial internal model of the program, which is stored in short term memory. It may well differ from the internal models, stored in long term memory, that a programmer has of previously used programs.

3.4 Externalization

Though not many studies have been undertaken on this subject, it may be interesting to look at how programmers externalize programs, that is how they express their knowledge of a program.

On one hand, this may tell us what programmers really know about programs. For example, having some programmers use a program, and then asking them to recall it at different moments in time could show us which program features they really remember. These features would be the basic knowledge held in the long-term memory internal representations. Such experiments have been led by Davies, but they were based at short-term memory. They confirmed the theory of focal lines in programming plans.

On the other hand, it might be interesting to see how programmers spontaneously express their knowledge (either programs, problems or requirements), and which external representations they naturally use. As we said before, Petre [21] found that, when programming, programmers used a "private, pseudo-language". Other experiments of this type may point at which external representations may be the easiest to use for programmers.

The aim of all this would be to teach programmers what knowledge they should externalize in order to make their programs easily understandable by other people, and how.

4 Representations applied to reuse

The problem of studying the use of external memories and representations in software reuse is made more difficult by the fact that software reuse doesn't just consist in looking at programs, be they internal or external. Each kind of software reuse is made of different successive steps, which have their own aims and requirements.

In this section we will describe the four important steps of software reuse: finding a reusable asset, understanding it, specializing it and integrating it. We should keep in mind that these four steps are not completely independent, as some understanding occurs during the finding and specializing stages. This is just a simple model of the reuse activity.

We will see how these processes differ in the use of the internal representations, external representations, internalization process and externalization process. We will also see how they differ depending on the reuse technique used. Moreover, each step can be supported by some specific tools, and knowing which cognitive processes are involved in each step should help us designing those tools.

4.1 The four basic steps of software reuse

4.1.1 Finding a component

In one's internal memory In the case of a programmer trying to reuse one of his own programs (the code itself or the design), he must browse through his own long-term memory to find a suitable object to reuse. Then, he may remember different possible solutions, and still have to choose only one of them.

- *Searching one's memory*

  As we said before, we think that long-term memory can store various representations of existing programs. Though we don't know that much about those representations, we know that they can be of different abstraction levels. By 'browsing' through his memory, a programmer is able to recall different programs, and to judge wether they correspond to his problem. It seems natural to think that this search is a top-down process: from a general idea of the problem, or a few key concepts, the programmer can remember many possible solutions. The search can then be refined to reduce the number of solutions. This implies that the programmer is able to abstract his problem and requirements.

- *Choosing between alternative solutions*

  Once the programmer has found some possible solutions, he must choose only one of them. This is supposedly done by comparing their internal representations, and particularly how close they match the internal representation of the problem.

- *Solution evaluation*

  Finally, the programmer may evaluate how suitable the component is. Presumably, if it doesn't reach a minimum level of suitability, the idea of reusing may be discarded, and the programmer will write something from scratch.

Here we see that the situation may be different depending on wether the subject wants to reuse the actual code of a program, or just a design. For example, comparisons and choices between possible solutions for code reuse should ideally be based on code details like presentation or comments, which may not be present in abstract internal representations. Therefore, the judgements may be more efficient and reliable for design reuse than for source code reuse.

In an external memory The core of component reuse is to search through a library to find a suitable component. Here, library has a very wide meaning: it can be a pile of design or program documentation, a book of example programs or a computer-based database of code or design components.

- *Searching the library*

  'Browsing' consists in looking through a whole library, until something interesting is found. As such, this method is only suitable for small libraries, such as books of examples. As more complex library structures were developed (particularly database-like libraries), some automated search tools were developed to speed up searches. We will describe some of these in 4.2.1.

  Though different, all these tools require the programmer to externalize his internal representation of the problem. The important issue is to find an easy way for the programmer to express his problem. Reciprocally, since the programmer evaluates whether each component is suitable or not, he also has to internalize the description of the component. It seems natural to think that a simple description may be enough at first.

- *Choosing between alternative components*

  Once he has browsed or searched through the library, the user must choose one of the different possible components he obtained. Here the internalization process seems crucial, as the programmer will compare his internal representations of the possible solutions. Some more precise information on those components may be necessary.

- *Solution evaluation*

  Once a programmer has been through the whole process of searching a library and choosing one component, he can evaluate whether this component is suitable for his particular problem, that is, if the search was, as a whole, successful. If not, he may go on looking for a more suitable component (possibly in another library), or just give up and create something from scratch.

4.1.2 Program understanding

Once a programmer has chosen what he will reuse, he has to understand more thoroughly how this object works. In an idealistic situation, he wouldn't have to do this: by putting together well-crafted 'building blocks', a programmer would just need to know what each block does. But we know that this is never the case. We already had a look at the program understanding issue in 3.3, but here we can make some further remarks.

First of all, this step is crucial for external memory reuse, where external representations are involved. But it sometimes occurs for internal memory reuse. In most cases, the programmer remembers general aspects of the program or the design, which are sufficient for the reuse to take place. Yet he can sometimes remember details at a low level which he needs to abstract to really understand the component. For example, he may remember precisely the structure of the objects in an object-oriented program, and need to understand why this structure was chosen in order to reuse the same design.

Secondly, in the case of external memory reuse, a programmer may reuse a program he has written before, or somebody else's program. Reusing one of his own program will make him use his old internal representations as well as internalize the external representations (such as the code itself). It may be interesting to study when each kind of knowledge is used. This may show us what is missing from the internal representations of programs written in the past, and what knowledge a reuser really needs.

4.1.3 Specialization

Specialization is the process of adapting a general piece of code to one's own specific problem. The simpler the specialization process is, the more reusable the reused asset will be. This process depends on the reuse technique employed.

- *Write/Copy/Paste*

  When a programmer must write some variations of the same program, two kind of specialization processes occur. First, he uses his internal representation of the general solution, as well as the first instance's specific properties, to write the first instance. Then, he will use this first instance, which is an external representation, and the internal representation of the general solution, as well as each other instance's specifics, to build the other instances. Here we see that both internal and external representations, and therefore the internalization and externalization processes are involved.

- *Code scavenging*

  As Krueger [11] put it, "a programmer specializes a scavenged code fragment by manually editing it". In order to know what must be specialized in the reused piece of code, the programmer presumably uses his internal representation of the program (which was built when he tried to understand it) and the internal model he has of his own problem.

- *Design scavenging*

  Here the programmer must specialize an existing design, supposedly using his internal representation of this design.

- *Code and design components*

  The use of ready-made components allowed for new specialization techniques to be developed. This specialization can be complemented by some manual editing.

- Parameters

  A reusable component can accept parameters when it is reused, or more precisely invoked, such as the stack's maximum size for a stack component. The drawback of this method is that it is quite narrow, and its applications are limited.

- Generic code

  A piece of code is said to be generic when it can use different types of input data: a programmer won't have to modify the component to take into account the data type he is currently using. The problem here is that it is quite difficult to accept a wide array of data types, since the programmer of the component always has to make some assumptions on the datatype he is using.

- The white box

  The original view of software reuse was to be able to assemble software components without looking at how they achieve their task, i.e. using 'black-box' components. But this often seems too difficult to be possible. By 'opening' the components, by looking at how they do what we want them to do, we may be able to select a more suitable behaviour. This principle of the white box has been applied to the Open Implementation Analysis and Design methodology, developed by Maeda, Lee, Murphy and Kiczales [14].

4.1.4 Integration

When a reusable piece of code has been found and specialized, the programmer still needs to put it into his program. This can lead to some problems, such as name clashes, incompatibility, etc., which must be solved, according to Krueger [11], by 'modify[ing] the fragment, the context, or both". Reused designs don't need any particular integration: they are high-level assets which ignore those low-level details. There are two different ways of integrating a piece of code in a program:

- *Code cloning*

  For the Write/Copy/Paste technique, and in some cases for Code Scavenging, the piece of code needs to be fully inserted in one's program. This method may require many heavy modifications of the program. It will obviously involve the programmer's internal representation of the whole program and of the component. Though, once the modifications have been identified, the externalization process seems fairly simple.

- *Code invocation*

  By contrast, for other Code Scavenging situations and for component reuse, the component can simply be referred to, just like an external program, i.e. a C library or function. This method is much lighter and simpler, and helps ignoring low-level details. This allows the programmer to work on a higher level of abstraction.

4.2 Providing tools for library-based software reuse

Now that we have seen the basic steps of software reuse, it is interesting to look at how they can be supported by computer-based tools.

What we originally wanted to know was how the use of external memory disrupted the cognitive processes involved in software reuse. As we have just highlighted the cognitive issues of the four important steps of reuse, we can now see whether existing support tools are adequate from a cognitive point of view, and how some new tools might be developed from this perspective.

4.2.1 Finding a component in a library

Searching the library Most libraries now provide some kind of search tool. They first require the programmer to describe their requirements or problem, and give back a list of possible solutions. Search tools must tackle two cognitive issues: to help the programmer externalize his problem or requirements, and to help him selecting some possible solutions.

- *Externalizing the problem*

  First, we have to see how the programmer should express his problem or requirements. We can see three possibilities:

  - Browsing through a tree of choices will obviously make it very simple for the tool to understand the programmer's choices, but it will reduce the width of choice the programmer has (and thus make his description less accurate). As a matter of fact, the structure of the tree is most important. The tree's designer must choose which successive choices the user should make, and how components can be assigned to one or many set of choices. Such a system can be found in the AltaVista Web search tool, which can display an activation network in which the user can select a few nodes.
  - Alternatively, the system can ask for some keywords, which can be matched with each component's own list of keywords. This gives much more freedom to the programmer, as long as the tool can analyze the words and look for synonyms or equivalent words. Though, it requires the programmer to express himself just using a few words, making him choose what is really important in his problem or required solution.
  - Finally, it seems at first simpler to ask the user to express himself in natural language. His description of his problem or requirements can then be matched with each component's textual description. For example, some Web search tools can deal with queries such as "I want to know how to get from Brighton to London by train". The AltaVista search engine for example is very accurate on this particular request, while Excite doesn't give very useful links. But such a tool may be very difficult to develop in the case of component libraries, and the gain in freedom and simplicity from the programmer's point of view might be compensated or reversed by the inaccuracy or inefficiency of the tool. On top of that, a full textual description may contain useless information that can hide the most important features of the

requirements. Keywords, by contrast, force the programmer to really think about what he wants. This is indeed an added cognitive weight on the programmer, but it may be at the end rewardful.

Besides, the programmer can describe two different things: a possible solution (which supposes that the programmer knows what he's looking for) or his problem (the tool then being able to match this problem to some solutions). Most keyword based tools allow the programmer to ignore this distinction, as keywords can indifferently describe the problem and the solution. By contrast, some natural language tools might provide a tool which will analyse the problem and find by itself what type of solution is required, though we still have to find how this can be done. The general idea is that it may well be simpler for a programmer, from a cognitive point of view, to express his problem rather than a possible solution, since problems are quite often already externalized, in the way of requirement drafts, contracts, etc.

As a conclusion, we can say that it may be interesting to conduct an experiment to support those ideas and see more thoroughly the advantages and shortcomings of the different methods of externalization.

- *Internalizing the component description*

  The other issue of searching in a database is to internalize the component descriptions, in order to find some components which, at first, might be suitable. Since this is just an early selection of components, it seems logical to think that a simple description of the component is sufficient. For example, the Asset Broker developed by BT gives short descriptions of just a few lines of its components, before giving access to longer descriptions. Most systems indeed use short textual descriptions (even sometimes the name of the component only) as an external representation. As a small amount of information is needed, textual description may well be sufficient.

  Besides, some search tools can provide added information which are adapted to the user's request. Yahoo! for example gives a probability rating (a percentage) of each Web page being about what the user is looking for. The user thus knows that Web pages rated at less than 70% might not be really interesting. The Asset Broker also gives an estimate of each component's suitability, using a Fair, Poor, etc. scale. This seems to be a very interesting trend to follow.

Choosing between different possibilities Here the internalization issue is crucial. The programmer has to 'cross-evaluate' a short-list of components. This undoubtedly requires further information on the components to allow a more precise evaluation and comparison. Though, the programmer may still not need to understand fully how the component works (for a code component) or its precise structure (for a design component). Yet the problem of which external representations to use remains. Most code or design component libraries provide textual descriptions at this stage. Some exceptions exist, such as the Design Patterns reference book [6] (see 4.2.2) which also provides a small diagram of each pattern's structure.

This once again can be supported by some information the library provides about each component for this particular search, such as the Yahoo! probability rating. Though, as the crucial choice of which component will be reused is made at this stage, programmers may be reluctant to trust an automated tool, and tend to rely on their own judgement only.

Evaluating the chosen component Finally the programmer has to evaluate whether, as a whole, the chosen asset is suitable enough, or whether he should try to look in another library or give up and write the component himself. This has more to do with confidence in component libraries than cognition, though once again some tools can be helpful (such as the Yahoo! probability rating).

4.2.2 Understanding a component

The basic aim here is to provide an appropriate component description so that the reuser can understand what he needs to understand as easily as possible.

Existing descriptions and tools As far as source code or software architecture description is concerned, we saw in 3.2 different external representations, such as the control-flow, data-flow and structure (object hierarchy) diagrams. Though, as we said before, no particular representation has yet been proved to be in overall more efficient than the others. Instead, each one of them is more suitable for some particular tasks.

Design patterns have been described by Gamma, Helm, Johnson and Vlissides in [6], by means of a 12-point list:

- *Intent* The short description of the purpose of this pattern
- *Also known as* An older name
- *Motivation* An abstract example of a situation where this pattern is useful
- *Applicability* Classic situations where we can use this pattern
- *Structure* The pattern itself
- *Participants* Other patterns that this pattern uses
- *Collaboration* Other patterns' behaviour re this pattern
- *Consequences* The advantages and shortcomings of the pattern
- *Implementation* Some techniques and advices to implement the pattern
- *Sample code* A complete example
- *Known uses* Actual uses in available programs
- *Related patterns*

The *Intent* might be used as a short description, for the early selection of a few possible patterns. A small diagram of the pattern's structure is provided as well.

Further developments Here we can try to see which directions could be followed to properly estimate which external representations are the most effective, or to try to design some new external representations.

- *Externalizing internal representations*

  A possible solution may be to map external program representations on our internal representations. The problem is that, as we said before, we do not know a lot on how our internal representations of programs are structured and organized.

  Object-oriented programming for example was such an attempt, since (as Ormsby [19] recalls it), it was usually claimed that OO programming (like nearly every new programming language in fact) 'introduces a software development model based upon the way humans think'. We could say for instance that the 'object' concept can be linked to the 'node' concept in the semantic network model of human memory. Inheritance and method calls could then be seen as the external equivalent to the connections between concepts. This was claimed by Goldberg and Robson [7] about the Smalltalk 80 object-oriented language.

  Yet this claim still has to be proved, and it may even be wrong as, for example, Lee and Pennington [13] showed that it was usually more difficult to understand an object-oriented program than a procedural one, since domain knowledge (as opposed to the task itself) tends to be more important in object oriented programming than in functional programming.

- *Natural representations*

  Another solution may be to look at which external representations programmers naturally use when they program, for example by looking at the small drawings they make while programming, and the notes they take. This may hint at which external representations programmers feel confident with and understand easily. This might give us some clues on what 'good' external representations should look like, though Petre showed that these representations differ from one individual to another.

- *Human laziness*

  Making programmers understand some piece of code that they are reusing may be more difficult than we think. Lange and Moher [12] described the "avoidance of comprehension" phenomenon: programmers are not inclined to make the (maybe unnecessary) effort to fully understand what they reuse. This has also been highlighted by Sutcliffe and Maiden [23], who said that programmers are usually "copying rather than reasoning while reusing specification components". The consequence is that providing a complete and accurate description of software components is probably not the ideal solution. Instead, by motivating reusers to actually try to understand the components they reuse, and more importantly by identifying the knowledge they really use in the program descriptions (that is, the information that they really need), we may find a way to make external representations shorter, simpler, and more efficient.

- *Programmers are individuals*

Another problem is that Davies, Gilmore and Green [2] showed that programmers do not classify pieces of code on the same criteria. Novice programmers tend to use surface criteria (such as class names), whereas expert programmers prefer deep semantic aspects, which may not seem obvious at first (such as some aspects of the code itself). Thus maybe different types of users also need different types of information to reuse a component. A solution to this problem may be to setup a library system that can alternatively display different kinds of representations, or allow the user to parameterize these representations, or even better that can build a profile of each user and display his favored representations.

4.2.3 Specializing a component

As we said before, specialization is the process of adapting a general piece of code to one's precise problem.

This step is quite different for Design Patterns. DPs don't need to be really specialized nor integrated into a program: instead they must be assembled together like building blocks. This was the original idealistic view of software reuse. From this point of view, DPs are the most successful kind of software reuse. Yet to assemble them is still quite like a programming task, but at a higher level, and combinations of patterns can be reused. It may be interesting anyway to find how a computer-based tool might support this assembly process.

As far as code components and software architectures are concerned, developments in new specialization techniques went in two opposite directions. The 'black-box' concept consists in hiding from the user what the component actually does (by restricting the specialization to some parameters or data types), while the 'white-box' concept tries to 'open' the component and automate the specialization at a low-level (in the case of the Open Implementation Analysis and Design methodology for example).

Apparently, the situation for code reuse is quite problematic: black-box components are not reusable enough, and white-box components are too difficult to specialize. Unless some more efficient and easy-to-use tools are developed, it seems logical to think that design-level reuse will focus most of the interest in the future.

4.2.4 Integrating a component

Finally the programmer must solve the disruption caused by integrating the new component in his program. This step does not usually occur for Design Patterns, unless only a part of a larger system has been redesigned with DPs. We can think that it is also not relevant for Software Architectures reuse where (a) a larger, stand-alone program is reused, and (b) this is done at a high level of abstraction, where problems such as name clashes do not exist.

As far as code reuse is concerned, we saw that the original technique was to copy/paste the component into the program, and that languages now supported the use of function calls to simplify this. That way a program is indeed more like a set of modules than one single list of commands. This system seems to be a satisfactory solution, as the frontier between the program and the component has been reduced to something which is easily understandable and manageable by programmers.

5 Conclusion

We have described different reuse techniques, and how the most interesting ones are based on the use of external memories. We also explained how the reuse process could be split in four steps. Though these steps are not completely independent (for example, some understanding occurs during the search and specialization stages), some computer-based tools can support each one of them.

Taking into account cognitive aspects of software reuse, we tried to bring new ideas about the design of these tools. But the main problem is that we still do not seem to know enough about some key issues such as the internal representations of programs. Therefore we could only draw up a list of advice, suggestions and ideas.

What appears is that on many issues, some alternative solutions have been suggested, without them being supported by any comparative experiments. Therefore we think that it would be interesting to develop a modular reuse system which would support such experiments. This system would be based on a set of modules, where each module can be chosen from a few alternative possibilities. These modules might include:

- the component type: code, design architecture, or design pattern
- what the user should externalize for the search: the problem or the behaviour
- how he should externalize it: using keywords, natural language, or by browsing through a tree of choices
- what knowledge should be displayed about each component for the early selection stage
- how this knowledge should be displayed: text, graphics, data or control flow diagrams, a combination of them, etc.
- what knowledge should be displayed, and how, for the cross-evaluation stage
- a specialization tool: to display further information, or to automate this process
- an integration tool

It would then be possible to ask some people to perform some kind of reuse activity with a given module configuration, and then with a slightly different one, and to draw comparisons between the efficiency of those configurations. It would also allow us to see whether different people prefer different configurations, and to identify some user profiles (for example, depending on the user's experience). Finally we would be able to isolate which kind of information users really need, by providing more or less complete representation components, and by comparing them.

Such a tool should not be designed towards supporting reuse in itself, but rather as a simple and modular system which will support experiments about software reuse.

References

[1] *Towards a theory of the comprehension of the computer programs*, R. Brooks, in the *International Journal of Man-Machine Studies*, n 18, pp 543-554, 1983

[2] *Are Objects that Important? The Effects of Expertise and Familiarity on the Classification of Object-oriented Code*, S. P. Davies, D. J. Gilmore and T. R. G. Green

[3] *Focal Structures in Program Comprehension: Implications for the Design of Programming Support Tools, Debugging Aids and Tutorial Environments*, S. P. Davies, in *Symbiosis of Human and Artifact*, Y. Anzai, K. Ogawa and H. Mori (Eds), Elsevier Science B.V., 1995

[4] *Reasoning from a Schema and from an Analog in Software Code Reuse*, F. Détienne, in ESP-4, pp. 5-22

[5] *Classifying Software for Reusability*, R. Prieto-Diaz and P. Freeman, in *IEEE Software*, pp 6-16, January 1987

[6] *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley Professional Computing Series, Reading, MA, USA, 1995

[7] *Smalltalk 80 - The language and its implementation*, A. Goldberg and D. Robson, Addison-Wesley, 1986

[8] *Cognitive dimensions of notations*, T. R. G. Green, in *People and Computers*, A. Sutcliffe and L. Macaulay (Eds.), Vol 5, Cambridge University Press, Cambridge, UK, 1989

[9] *Programming languages as information structures*, T. R. G. Green, in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds), pp 117-137, Academic Press - Harcourt Brace Jovanovich, London, UK, 1990

[10] *When, Why and How do Novice Programmers Reuse Code?*, C. M. Hoadley, M. C. Linn, L. M. Mann and M. J. Clancy, http://obelisk.berkeley.edu /tophe/ESP6/ESP95Final.html, 1995

[11] *Software Reuse*, Charles W. Krueger, in *ACM Computing Surveys*, Vol 24, n 2, June 1992

[12] *Some Strategies of Reuse in an Object-Oriented Environment*, B. M. Lange and T. G. Moher, in *Proceedings of CHI'89*, K. Bice and C. Lewis (Eds.), ACM Press, New-York, USA, 1989

[13] *The Effects of Paradigm on Cognitive Activities in Design*, A. Lee and N. Pennington, in *The International Journal of Human-Computer Studies*, n 40, pp. 577-601, 1994

[14] *Open Implementation Analysis and Design*, Chris Maeda, Arthur Lee, Gail Murphy and Gregor Kiczales, http://www.parc.xerox.com/spl/projects/oi-at-parc/ourpapers/oiad.pdf, 1996

[15] *Mass Produced Software Components*, M. D. McIlroy, in *Software Engineering: Report on a Conference by the NATO Science Committee (Garmisch, Germany, Oct.)*, P. Naur and B. Randell (Eds.), NATO Scientific Affairs Division, Brussels, Belgium, pp 138-150, 1968

[16] *The Draco Approach to Constructing Software from Reusable Components*, J. M. Neighbors, in *IEEE Trans. Soft. Eng.*, SE 10, n 5, pp 564-574, September 1984

[17] *Draco: a Method for Engineering Software Systems*, J. M. Neighbors, in *Frontier Series: Software Reusability: Volume 1 - Concepts and Models*, T. J. Biggerstaff and A. J. Perlis (Eds.), pp 295-319, ACM Press, New-York, 1989

[18] *Human Cognition and Programming*, T. Ormerod, in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds), pp 63-82, Academic Press - Harcourt Brace Jovanovich, London, UK, 1990

[19] *Software Engineering and Cognitive Science*, Andrew Ormsby, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, June 1996

[20] *Stimulus structures and mental representations in expert comprehension of computer programs*, N. Pennington, in *Cognitive Psychology*, n 19, pp 295-341, 1987

[21] *Expert Programmers and Programming Languages*, M. Petre, in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds), pp 103-115, Academic Press - Harcourt Brace Jovanovich, London, UK, 1990

[22] *Empirical Studies of Programming Knowledge*, E. Soloway and K. Ehrlich, in *IEEE Transactions on Software Engineering*, Vol SE10, n 5, 1984

[23] *Software Reusability: Delivering Production Gains or Short Cuts*, A. Sutcliffe and N. Maiden, in *Human-Computer Interaction - INTERACT'90*, D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Elsevier, Amsterdam, Netherlands, 1990

[24] *Industrial experience with an integrated code comprehension model*, A. von Mayrhauser and A. M. Vans, in the *Software Engineering Journal*, September, pp 171-182, 1995