National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Facilitating the Reuse Process in an Object Oriented Environment by Learning from Observation

by
Thierry Rovel

Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
Master of Computer Science

The Ottawa-Carleton Institute for Computer Science
Computer Science Department
University of Ottawa

ISBN 0-315-82546-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgments

I am greatly indebted to my supervisor, Robert Holte, for his guidance in the course of this research. I also want to thank him for his availability and his tremendous work on earlier drafts of this thesis. My thanks go to Christopher Drummond, who helped me in achieving the final version of the thesis.

The Ottawa Machine Learning Group provided me an excellent platform to exchange ideas. I particularly thank Peter Clark, Denys Duchier and Stan Matwin for all their suggestions. I also thank Stephanie Aha and Mark Israel for their assistance in correcting my 'quite imperfect' English.

I am grateful for the funding from the University of Ottawa and from NSERC through a strategic grant obtained by S. Matwin, R. Holte and F. Oppacher. Without this financial help I wouldn't have been able to do this research.

Abstract

The aim of this research is to ease the reuse of components in an Object-Oriented library. We ease the three main processes included in reuse: searching for, understanding and adapting components. The novelty of this approach is that all the information needed to achieve these improvements is learned automatically by spying on the programmer's actions.

This thesis describes a model for a learner in an Object-Oriented environment. A specific implementation for the Smalltalk language, called LEARNIST (Learning to EAse Reuse IN SmallTalk), is described and illustrated with examples.

Table of Contents

List of Figures

Chapter 1: Introduction

The aim of this research is to ease the reuse of components in an OO (Object-Oriented[1]) library. We ease the three main processes included in reuse: searching for, understanding, and adapting components. The novelty of this approach is that all the information needed to achieve these improvements is learned automatically by spying on the programmer's reuse. The system learns information about components that are built by copying and modifying other library components (*reuse by modification*) or that are executed in a program (*reuse by use*). These two kinds of reuse are frequent since *reuse by modification* is a common technique to build new components and components are designed to be *used* in programs. Once installed, this system learns incrementally each time a component is reused.

The main advantage of automatic learning is that our system is non intrusive. There is no additional cost for users since there is no visible change in the programming process. Our system doesn't require user's input nor does it force the use of a specific programming methodology. The user interacts with our system if he wants help in searching for, understanding or adapting a component. With this non intrusive option, even systems that bring only slight improvement are valuable since they don't have to overcome inherent cost.

One focus of this research is learning in a "real" OO environment. This imposes two design requirements. Firstly, the "spy" module should be able to spy on all the development tools the OO environment provides. Secondly, the system should handle big libraries: this requirement compels us to tackle efficiency issues.

---

[1] In the rest of the thesis, we will always abbreviate Object-Oriented by OO.

This thesis describes a model for this learner in an OO environment. A specific implementation for the Smalltalk language, called LEARNIST[1], is described and illustrated. LEARNIST works on a realistic application: the whole Smalltalk environment with a library of about 2 megabytes.

1.1 Improving the reuse process

OO languages have been designed to maximize reusability of software (see section 2.2). This has led to large libraries of reusable components. However, presently OO environments do not support reuse activities (see section 2.3).

Reuse activities are usually seen as composed of three phases: searching, understanding and adapting ([Fisher 91], [Biggerstaff 87, 89], [Maarek 91]). We examine each of these three phases to describe the improvements our system introduces.

a] searching:

In many OO environments[2], browsing is proposed as the main way to search for a component.

Normal browsers just exploit relationships which are part of the OO languages themselves. The main one is inheritance. [Rosson 91] analyzes the OO paradigm from a cognitive point of view. It states that the "inheritance hierarchy supports only the generalization relations among objects ...., it seems clear that there are many more relations that objects may have with one another" (pp. 373-374).

We improve browsing by adding a "similar" relationship. This relation links components whose function or algorithms are similar. This is a very natural relation: when a user has

---

[1] Learning to EAse Reuse IN SmallTalk.

[2] such as Smalltalk, Objective C, C++, Object Pascal.

found a component which is somewhat close to what he is looking for, this relation allows him to see if there is a closer or even a perfect match.

We also especially facilitate the location of components that have already been reused by drawing the user's attention to components which are more likely to be reusable.

b] Understanding

We can define three ways of understanding depending on the aim of the user:

- The user just wants to have a quick overview of the components to see if a component has some chance of meeting his requirements.
- The user wants to understand how to use the component (*reuse by use*). Here, he wants to see how to call this component from the exterior (the *interface methods*) and the possible parameters of the calls.
- The user wants to understand how a component is implemented in order to specialize it (*reuse by specialization*) or to modify it (*reuse by modification*). Here he wants a deep understanding of the component and its dependencies.

OO environments don't provide a lot of tools to support this. Moreover we show that the main barriers to understanding are deeply rooted in the OO paradigm and that static analysis can't solve this problem (see section 2.3.1). Our system overcomes the ambiguity created by highly dynamic features of the OO paradigm, namely polymorphism coupled with overloading and inheritance (see section 2.2).

Our system is able to answer questions like:

- Who has called this method? (context of use).
- What has this method called? (dependencies of the method).
- What has been the range of arguments for this call? (type information).
- What are the *interface methods* ? (role of the components).

We also provide abstraction to allow the programmer to have a quick but complete overview of a component. The importance of abstraction in reuse is shown in [Krueger 92].

c] adaptation

Our system improves this phase by displaying previous examples of adaptation which enables users to build their components by analogy with these examples. (see section 4.6.3).

1.2 How to get this information ?

All these improvements are based on new information that our system learns. In this section we explain how we acquire the needed information.

Our main source of information is the programmer's normal actions during reuse. These actions indicate how to reuse a component. By spying on the programmer, we can collect this information.

[Biggerstaff 89] and [Krueger 92] have defined frameworks for reusability. They describe three forms of reuse:

- Reuse by use.
- Reuse by specialization.
- Reuse by modification.

Reuse by use occurs when the programmer reuses components as they are. For simplicity we even consider that it is *reuse by use* when a component is used for the first time. This means that *use* and *reuse by use* are synonyms.

To spy on components when they are *used*, we spy on the execution of programs.

Reuse by specialization is supported by the inheritance mechanism. It is achieved by creating a *subclass* of an existing class. We don't need to spy on this type of reuse since it is denoted by the inheritance hierarchy.

Reuse by modification occurs when creating a new component by modifying existing library components. It is done by first copying then modifying source code. We spy on the copying action. In the rest of this thesis we refer to it as the *copy-paste* action.

1.3 Sketch of a solution

Our model is composed of three parts:

- Acquiring raw information by spying on a programmer's actions when he reuses.
- Processing this raw information into a meaningful form which is stored.
- Formatting and displaying this stored information in order to achieve the improvements.

The following figure gives a sketch of how, with this raw spied data, our system achieves the improvements described in section 1.1.

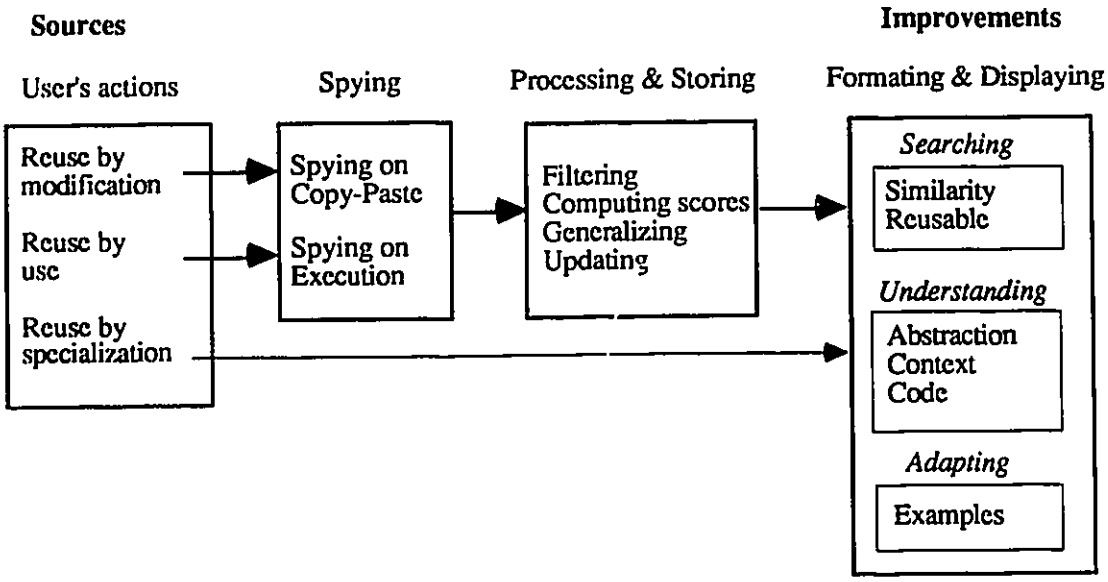


Figure 1: Sketch of a solution

For example, suppose that the programmer copies the code of method M1 and modifies it to create method M2. Our system intercepts this copy action. Then, it checks to see if this copy was used to create a similar method. If it is the case, it stores the fact that M1 and M2 are similar. Finally, when the user wants to look for methods similar to M1, it displays method M2 and a score reflecting the degree of similarity between the two methods.

1.4 Statement of the problem

We can state our problem as follows:

Design a non intrusive software tool for a "real" OO environment that learns from the programmer's reuse activities (by use, by specialization and by modification) in order to ease the reuse process (finding, understanding and adapting).

1.5 Research contributions

* Investigation of the advantages and drawbacks of the OO paradigm for reuse.

We explain how the "object model" and the dynamicity introduced by this new paradigm leads to more reusable code. We also discuss its drawbacks in terms of static analysis and understanding.

* Proposal of a new approach to the reuse issue.

We propose a non intrusive approach based on automatically spying on the programmer's actions.

* Development of a model applicable to "real world" OO environments.

This model is composed of three parts: acquiring raw information by spying, processing it, and improving the reuse process.

- Ways have been found and justified to spy on users' actions when reusing.
- Algorithms have been developed to process this raw information.
- Improvements achieved by this model have been discussed by referring to the literature.

* Implementation of this model in the Smalltalk environment.

- Core modifications have been made in order to spy on execution of programs.
- A specific procedure has been developed to store the spied information inside the component itself.

* Illustration of LEARNIST's ability to learn information and to improve to the reuse process.

1.6 Outline

Chapter 2 discusses software reuse in the OO paradigm. In chapter 3 an overview of related research is presented. Chapter 4 sets out the model applicable to OO libraries. Chapter 5 describes LEARNIST, a specific implementation for the Smalltalk environment. Chapter 6 illustrates LEARNIST's ability to learn and to improve the reuse process. Finally, in chapter 7, we conclude and discuss limitations of the present system and ways to overcome them.

Chapter 2: Reuse and OO programming: an overview

Our research is in the intersection of two fields: software reuse and OO programming. In this chapter we give an overview of each.

2.1 Software Reuse

In describing real world software development, Basset states:

"Numerous studies[1] have shown that more than 90% of typical business programs reinvent solutions that exist in other programs. ... Attacking the reinvention issue ... could therefore collapse development efforts to the 10% of code which is truly unique." [Basset 90, p. 17]

The benefit of software reuse is not only to decrease the time and cost of the software development but also to increase the reliability and maintenance [Boehm 83].

McIlroy first introduced the concept of "software reuse" in the NATO conference in 1968 [McIlroy 68]. He saw it as a means to overcome the "software crisis" which can be defined as the problem of building large, reliable software systems in a controlled, cost effective way.

More recently, [Krueger 92] defines it as: "software reuse is using software artifacts during the construction of a new software system." (p. 133).

Reuse affects all the aspects of software development. We can distinguish three main issues:

---

[1] One example of them is: [Jones 84].

- Software engineering issues. The goal is to ease the reuse process by providing new software engineering tools.
- Design issues. The goal is to make software more reusable. It involves the design of language, of programming methods, and of libraries.
- Psychological and organizational issues. Reusability has to become a "state of mind" as it tends to be in civil or electrical engineering.

This research tackles the first issue. We aim to facilitate a specific type of reuse: reuse of components in an OO library. In such libraries *components* are *classes* which can be further divided in *methods.*

The following aims at giving a broad context of reusability technologies and locating our type of reuse within it. We use the frameworks discussed in [Biggerstaff 89] and [Krueger 92].

Biggerstaff et al. define two major groups depending upon the nature of components being reused. These groups are "composition technologies" and "generation technologies".
"In *composition technologies*, the components to be reused are largely atomic and, ideally, are unchanged in the course of their reuse." (p. 2).
They further divide "composition technologies" into two groups. The first group emphasizes "application component libraries" such as libraries of subroutines. The second group puts the emphasis on organizational and compositional principles. They classify our type of reuse in this last category. In OO programming, organization is achieved by the *class* and *inheritance* notions. Composition is done by the *message passing* mechanism.
Krueger partitions "composition technologies" into three sub-levels depending upon the type of language used to implement the elements: "procedural languages" that emphasize reusable

functions, "object-based languages" that allow abstract data-types and "object-oriented languages" that add the notion of inheritance.

Generation technologies use a program generator to convert a very high level or a specification language into executable code. These technologies reuse design knowledge.

2.2 OO Paradigm

OO programming is a relatively new field of computer science based on the "object model". In this model, data and operations are united in such a way that the fundamental logical building blocks of systems are no longer algorithms but *objects*. A programs can be defined as a cooperative collection of *objects*.

In the next sections, after defining key vocabulary, we explain how the OO paradigm plays a critical role in software reuse. In the last section we show that even though the OO paradigm is well suited for reuse, new tools are still needed.

2.2.1 Definitions

OO language: "a language is OO if and only if it satisfies the following requirements:

- It supports *objects* that are data abstractions with an interface of named operations (*methods*) and a hidden local state (variables).

- Objects have an associated type (*class*).

- Types (classes) may inherit attributes from supertypes (superclass)." [Cardelli 85, p. 481].

Class: " is a set of *objects* that share a common structure and common behavior" [Booch 91, p. 513].

Object: is an instance of a class. The terms *object* and *instance* are interchangeable.

Method: is "An operation upon an object, defined as part of the declaration of a class" [Booch 91, p. 515].

Message: "is a request for an object to carry out one of its operations. A message specifies which operation is desired, but not how that operation should be carried out. The *receiver*, the object to which the message was sent, determines how to carry out the requested operation." [Goldberg 89, p. 6].

For example, we could define a *class* 'Human' and an *object* 'Thierry' as an *instance* of this *class*. Further in this class, we could define the *method* 'oneStep' with an argument: 'direction' (forward or backward). We would implement it, as moving one leg of the human forward or backward.

To make the *object* 'Thierry' move one step forward we would send it the *message* 'oneStep' with the *argument* 'forward'. We would write this (with a Smalltalk syntax) as the following:

'Thierry oneStep: forward'

Classes and *methods* are the two main levels of granularity in an OO library. We use the generic term *component* to refer to both of them.

2.2.2 Reusability and the OO paradigm

This section serves two purposes: defining the concepts of OO paradigm and showing their critical roles in reuse. It is based upon the definitions and the examples in two books: [Booch 91] and [Meyer 88].

Meyer says that "our main thesis is that OO design is the most promising technique now known for attaining the goals for extendibility and reusability." [Meyer 87, p. 51]. This claim is based on the three main concepts of the OO paradigm: abstract data type, inheritance and polymorphism. These will be illustrated with Smalltalk examples.

1. Abstract data type

An *abstract data type*, called a *class* in the OO paradigm, is a class of objects characterized by the operations available on them (*methods*) and their state (*variables*).

This notion combines the notions of type and module. It is the major concept in the OO paradigm.

This notion also leads to data abstraction: the data are local to the *class* and other *classes* manipulate them via the methods of this *class*.

For example, we can define the class 'Rectangle' with the following variables:

- X, Y which indicate the position on the screen of the lower left corner of the rectangle.
- 'height' and 'width' which define the size of the rectangle.

In Smalltalk, we declare this class as follows:

```
Object subclass: #Rectangle          define the class Rectangle as a subclass of class Object
        instanceVariableNames: 'X, Y, height, width'
```

We achieve data abstraction by defining methods to manipulate the variables.

For example we can define method 'position' which returns the position:

```
position
        " answer the position of the rectangle"
        ^ (X @ Y)
```

Now, when you manipulate an instance of 'Rectangle', say 'my_rectangle', you can access its position by: *'my_rectangle position'*

With this data abstraction, you can change the coordinate representation from cartesian to polar without any change but in the 'Rectangle' class. You just have to change the X, Y variables to ρ, θ and to modify the method position as following:

```
position
        " answer the position of the rectangle"
        | X Y |                                    declaration of temporary variables
        X := ρ cos θ .
        Y := ρ sin θ .
        ^ (X @ Y)
```

To access the position everywhere in the system, we still write:

'my_rectangle position'

This notion of *class* "packages" together all the related routines to rectangle. If someone wants to reuse the class 'Rectangle', he finds in one place all the methods for manipulating rectangles.

For example, we can define in this class the method 'area' which computes the area and the method 'draw' which draws the shape of the rectangle.

2. Inheritance

Inheritance allows the specialization of a *class* by creating a *subclass* which inherits *methods* and internal structure (i.e. *variables*), and adds its own.

When you write a method in a subclass, you have three possibilities:

- Adding specific features. You just define a new method, with a new method name, and implement the specific features.

- Overwriting existing inherited methods to specify a different behavior. This is achieved by defining a new method with the same name as the one you want to overwrite. It is called *overloading* a method.

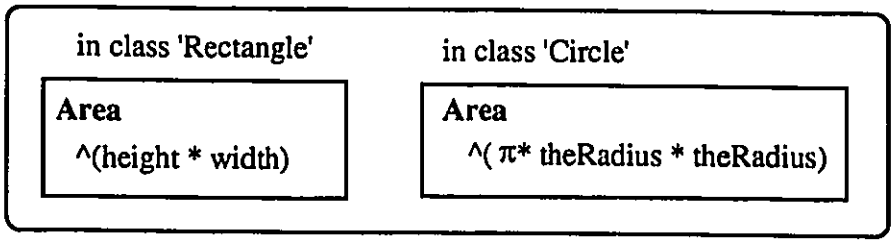
- Specialize an existing inherited method. This is achieved by *overloading* the inherited method and, in the new method, by calling the overloaded method.

The first possibility allows the grouping of common features in a *class* and the adding of specific features to the *subclasses*. For example we can create a generic class 'Shape' with the X, Y variables. We can also define the method 'position' that applies to any type of shape. Then, we can define classes 'Rectangle' and 'Circle' as subclasses of class 'Shape'.

Now the class 'Rectangle' has only 'height' and 'width' as variables. Its definition is:

```
Shape subclass: #Rectangle          define the class Rectangle as a subclass of class Shape
        instanceVariableNames: 'height, width'
```

We can implement the method 'area' in both subclasses:



The second possibility allows programming by difference: you take a class that closely matches your requirements as the superclass and you only describe the differences.

The last possibility allows the specialization of inherited methods. For example suppose that we want to create the class 'SolidRectangle', which behaves as a normal rectangle except that, when it is drawn, the enclosed area is gray: 

Let's suppose that class 'Rectangle' has a method 'draw' that draws the shape. We redefine 'draw' in class 'SolidRectangle' as following: first draw its shape by calling the 'draw' method of 'Rectangle', then color in the shape. We write this in Smalltalk as follows:

```
draw
        super draw.                          call to the superclass.
        { code to color in the shape}
```

3. Polymorphism and dynamic binding

Polymorphism allows a program to refer at run time to *instances* of different *classes*.

In Smalltalk, it is implemented via the message passing mechanism. For example, a method that sums the elements of an array works correctly whenever the *class* of the elements of the array understands the addition *message*.

Polymorphism is achieved via *dynamic binding*. It allows the run time system to automatically select the version of a method adapted to the corresponding *instance*. It is implemented in Smalltalk by performing a *dynamic look up* at run time to find the correct method.

For example, we can write a method that take a list of heterogeneous shapes (i.e. circles, rectangles and solidRectangles) from the user and computes the sum of the areas. In this situation, the compiler can't statically generate code to invoke the proper 'area' method, because the class of each shape is not known until run time. We write this method as following:

```
sumArea
        | listOfShapes  sum |                          temporary variables
        listOfShapes := { ask the user to enter the list }
        sum := 0.
        listOfShapes do: [ :aShape |                    iterate over the list
                sum := sum + aShape area].
        ^ sum
```

In Smalltalk the *dynamic look up* is performed as follows:

When a message (e.g. 'area') is sent to an object (e.g. 'aShape') the object looks it up in the list of methods of its class. If the method corresponding to a message exists (as for 'Rectangle'), the code for that method is invoked. If the corresponding method is not found, the object looks for a matching method in its *superclass* (as for 'SolidRectangle'), and so on.

2.2.3 Empirical investigation of the OO paradigm and reuse

This section briefly mentions empirical evidence showing that the OO paradigm is well suited for reuse.

Lewis et al. did some experiments to measure the relative effects of procedural languages and OO languages in term of software reuse. The experiments showed that "The OO paradigm substantially improves productivity over the procedural paradigm" and it "has a particular affinity to the reuse process." [Lewis 92, p. 41].

An interesting conclusion of experiments conducted in [Henry 90] is that, even though the OO paradigm results in better reusable code, programmers view OO technology as more difficult to use. The next section deals with this issue.

2.3 OO paradigm is not sufficient in itself to ensure effective reuse

The OO paradigm is not sufficient in itself to ensure effective reuse for two main reasons. Firstly, the new concepts introduced by this paradigm are difficult to manipulate and can decrease understandability. Secondly, given an OO library of reusable components, tools are need to assist in locating, understanding and modifying these components in the library.

2.3.1 Drawbacks of the dynamicity of the OO paradigm

Many traditional languages allow pointers to procedures to be passed as arguments thereby allowing execution of a dynamically determined procedures. But in the procedural paradigm this kind of dynamic binding is not encouraged, hence rarely done. However, in OO languages, invoking a method is usually dynamic because the *class* of the *object* being operated upon may not be known until run time (such as the 'shape' example in section 2.2.2).

[Ponder 92] analyzes this issue in the Smalltalk language. It states that "In order to understand a program the programmer must determine the dynamic types (classes) of a program variables." (p. 76). Without this "*class*" information, the programmer can't determine to which class a message is sent. Thus he cannot determine the method implementing the message. However seeing the method code is the only way to know exactly what this message is doing.

As Smalltalk is an untyped language, the *class* of an *object* can only be inferred by analyzing the messages sent to it. It can't be inferred statically because of the possibility of *overloading* and *inheritance.*

Every Smalltalk instruction is of the form: *anObject aMessage*. The only available information to determine the class of the object statically, is the message name.

In the following, we borrow the results of [Ponder 92].

Overloading lets you redefine a method in a new class with the same name. In Smalltalk-80 version 2.3, 20% of the methods are overloaded. This means that if one of these messages (i.e. method names) is sent to an object, it is impossible to determine its class statically. *Inheritance*, allowing a class to inherit methods from all its superclasses, further complicates the analysis. In Smalltalk, 55% of the methods can be used with more than one class, and 12% can be used with more than 25 classes. That means that more than half of the time we can't determine statically the exact class of the object. The conclusion of Ponder et al. is: "the result is considerable ambiguity." (p. 79).

The possible *classes* of the *object* can be narrowed down by studying all the messages sent to a given object within a method. But in OO languages, methods tend to be small: in Smalltalk, the average size is less than 10 lines. Therefore the chance of having a large number of messages sent to a given object within any given method, is small.

This problem is so important that programmers often indicate the type of the variables in the variable name, such as *anInteger* or *aString*. But, since this is informal and not enforced, it is not always done and it can be misleading if not correctly maintained.

Ponder et al. conclude by saying that "Polymorphism is a powerful and flexible programming mechanism" but it "can ruin program understandability". This problem can be addressed by "disciplined programming" and "by additional mechanisms such as the use of type assertion to inform the reader of the range of the possible types." (p. 79). In our system,

we do these assertions automatically for the argument variables and we provide, for each method, the list of the actual *callers*.

[Nielsen 89] agrees with Ponder. "It is difficult to understand which method actually gets executed in response to a message especially because you often don't know the *class* of the receiver." (p. 74). It also says that "Inspection of static code is probably the least effective way to understand (code). Even novices should use a run-time source debugger." (p. 76).

2.3.2 Facilitating the reuse process

[Fisher 91] notes that OO languages "by themselves do not support the location and understanding of modules (components)." (p. 319).

[Rosson 91] mentions that "little attention has been paid to the problem of determining whether and how to use a candidate component once located" (p. 277).

[Deutsch 89] states that inheritance allows "a type of reuse that is unique to the OO approach: reuse of design through frameworks of partially completed code." (p. 57).

"Single-class frameworks" are classes that are partially abstract; that is, classes that supply a partial implementation and expect subclasses to complete the implementation. The class 'Shape' is an example. "Multiple-class frameworks" expand the idea of framework to encompass the design of a family of related classes. An example can be the Model-View-Controller in Smalltalk-80. Deutsch mentions as a problem that "while the Smalltalk-80 system employs framework reuse to good effect, it does so entirely by convention. There is no support in the language, and little support in the programming tools." (p. 69).

Our system can detect these frameworks (see section 6.2.3) and provide examples of frames' usage.

Empirical studies ([Curtis 88], [Reeves 90]) highlight difficulties of OO programming in the three phases of the reuse process:

- Searching: How to know about the existence of components and how to access them?
- Understanding: How to facilitate the understanding of what has been retrieved?
- Adapting: How to combine and adapt components for particular needs?

Our system partially overcome the drawbacks of the OO paradigm and improves all three phases of the reuse process, namely, locating, understanding and modifying. It enables the user to better exploit what we see as the OO paradigm's greatest strength - REUSE.

Chapter 3: Related work

We can view our research in three ways:

- As a system that eases reuse.
- As a learning apprentice.
- As an alternative to relational code analyzers.

In this chapter, we give an overview of the related work in these three areas.

3.1 Easing reuse

We have clustered the related work on easing reuse by the way the information needed to achieve the improvements is obtained.

We can distinguish three ways of obtaining reuse information:

- Manual: all the information needed is provided by hand.
- Semi-automated: the computer helps the user to compute and organize the needed information but a person has to provide the raw information.
- Fully automatic: no information is asked of the user.

We don't cover systems that enforce a specific design methodology since, inherently, they are intrusive. The research reviewed is not "inherently" intrusive since it is not the user but a developer who has to provide the needed information and, therefore, these systems can theoretically be automated.

3.1.1 Manual approach

The View Matcher system [Rosson 91] assists a user attempting to incorporate an existing class into a new design. This work has the same goals as ours — to help the reuse process — but in this system the "help" information is provided manually.

The authors observe that the Smalltalk system doesn't provide help in the form of "usage examples", object-specific analysis (role of the object), how-to-use-it information, and object connections. Their View Matcher was designed to support these aspects of component reuse.

Their system is composed of three views:

- The "vignette view" lists the applications that use one given class. For each application we can have a brief animation of the use of this class. We can also divide this animation into episodes and see all the messages sent to this class.
- The "object communication map view" depicts, for a given application, the connection the object has to other objects through its variables. For example, it could tell that the object 'my_car' has a variable called 'owner' that is (connected to) the object 'Thierry'.
- The "commentary view" gives a text explanation about the usage of the inspected class.

Unfortunately "all of the documentation for the target class is developed by hand." (p. 283). For their future work, they speak about assisting in the preparation of their documentation components.

Our system provides, in a fully automated way, help in the form of usage examples and a limited form of how-to-use-it information. But we don't have the notion of episodes.

The main alternative to browsing for locating software is indexing. Numerous researches have tried to improve indexing of components through manual assignment of attributes for each component of a library.

[Prieto Diaz 91] improves indexing by introducing the notion of *facets*. Facets are sets of attributes describing the function and the environment in which a component is used. This

system requires an initial stage in which an extensive domain analysis is performed to select attributes and their values. For each component, a human librarian must assign the values of each facet. Then, the user can form a query as a tuple of facet's values.

Other researchers have looked at assigning a formal specification to each component. For example [Meggendorfer 91] uses precondition and postcondition to define a relationship.

3.1.2 Semi-automated systems

[Fisher 91] describes two prototype systems: *CodeFinder* for locating software components and *Explainer* to help understand the retrieved components.

CodeFinder uses a combination of two retrieval techniques to support retrieval without complete knowledge of what is needed. The first technique, *retrieval by reformulation*, allows users to incrementally construct a query. It allows the user to construct and refine retrieval cues "for problems that are not defined precisely to begin with, but are continuously elaborated and evaluated until appropriate information is found." (p. 320). The second technique, *retrieval by spreading activation*, uses associations to retrieve items that are relevant to a query, but do not match it, thus helping designers when they cannot fully articulate what they need.

Explainer gives refinable explanations of what is retrieved. The user can refine it by having text explanations for particular fragments of the code.

For each software component, a domain expert has to provide information like a category, description, parameters, keywords, related objects, and an explanation graph. Then, their system automatically organizes this information.

In our system, we provide a similar kind of incremental search by first displaying an overview of objects. If the user thinks this object matches well with his requirement, he may explore further and obtain more details. We also use examples not only to help the understanding of a component but also to show how to reuse it.

3.1.3 Fully-automated systems

Several researchers have investigated the *automatic creation of indexes* for a software librarian.

[Fraser 89] proposes a system where keywords are automatically extracted from the documentation of every component. Three Boolean operators (and, or, not) in combination with the keywords are used to form a query.

[Helm 91] proposes a new technique using *lexical affinities* for retrieving and browsing in OO libraries. The system uses readily available sources of information: the source code and its associated documentation. "The chief advantages of this approach, compared to previous approaches, are that information about classes is acquired automatically and cheaply — no human intervention is required to acquire information about each class; and it is readily scalable and extensible — classes can be added to the library with very little effort." (p. 47).

It uses the documentation to extract *lexical affinities*. In linguistics, a *lexical affinity* is the relationship between two words and represents the correlation of their appearance in a phrase. The lexical affinities which have a high "resolving power" are selected as indices. The resolving power is a score reflecting how well lexical affinity describes the component. The query is a free text sentence; the lexical affinities in that sentence are extracted by the same method as for the documentation. Then, these are matched against those associated with the documentation of each class using a distance measure and ranked according to the score. By using numerical clustering techniques, this distance measure is also used to cluster the classes and therefore allows browsing. The authors claim that this new hierarchy provides a means to browse "functionality" rather than "structurally related components".

This research is similar to our work in the sense that both help to locate components and both are achieved automatically. The difference is that they improve indexing and we

improve browsing. We do not view these as mutually exclusive means of search. It would be preferable for a system to support both methods.

[Drummond 92] proposes a new approach, called *active browsing*, to improve the speed and accuracy of the search phase. The system spies on the user's actions within a normal browser "to infer an analog representing the user's search goal. A relevancy measure is constructed from this analog and used by the system to scan the library independently of the user and to evaluate potentially interesting components. The results affect the browser display to emphasize relevant components and thus aid search." (p. iii).

Active browsing uses the same philosophy as the one adopted in this research which consists of spying and automatically inferring information. The major difference is time scale. The scope of Drummond's *active browser* is a single search: it acquires useful information to help the current search and then it forgets all. In our system, we have adopted a long term approach; i.e. we learn incrementally for the full life of the library.

3.2 Learning apprentices

Our system learns automatically all its information by spying on the user. In Machine Learning, such a system is called a learning *apprentice*. [Mitchell 83] introduced this term. It defines *learning apprentice* systems as "the class of interactive knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem-solving steps contributed by their normal use of the system." (p. 272).

It presents a system, called LEAP, which provides advice on how to refine the design of a VLSI circuit while allowing the user to override this advice and to manually refine the circuit. In the latter case, "LEAP records this problem-solving step as a training example of some rules that it should have. LEAP then generalizes from this example to form a new rule."

(p. 272). LEAP's explain-then-generalize method is based on having an initial domain theory (a knowledge-base of VLSI design) that allows LEAP to produce justifiable generalizations from single training example.

[Hill 87] states that "recent work on learning apprentice systems suggests new approaches for using interactive programming environments to promote software reuse." (p. 338). This approach is based on explanation based-learning, which aims at generating concepts using an explanation of a single training example in terms of a domain theory. Such a theory can be found in software specification or software validation. Hill's system, called LASR, "uses explanation-based learning with formal specifications to capture and generalize program abstractions ... to increase their potential of reuse." (p. 338).

These two systems rely on strong theories; we only use background knowledge that can be easily extracted from the OO library (such as the inheritance hierarchy).

3.3 Relational code analysis

By spying on the execution of programs, our main goal is to capture relations between components. A traditional *relational code analyzer*, such as Master Scope [Teitelman 74] or CIA [Chen 86], derives code-level relationships like function-calls-function or function-uses-variable by static analysis. This information is generally stored in a database. One can query this database using relational language. Others, like GraphBuild [Bigelow 87], have adopted a graphical way of using information. GraphBuild converts C source code into a hypertext source graph, based on the "program's call tree".

Unfortunately, in OO languages a program's call tree can't be inferred by static analysis (see section 2.3.1).

The Track system [Bocker 90] is a trace construction kit for OO language which overcomes this problem by dynamically tracing the execution. It is done in a graphical and interactive way by placing "obstacles" between icons representing classes. This system is used as a browsing and a debugging tool.

Track uses the same method as ours to record information: spying on the execution of code by intercepting messages sent to objects. But it uses this information for a different purpose: visualizing the trace of execution.

Chapter 4: Abstract view of the system

In this chapter, we describe an abstract model of our system.
This model is composed of three parts: the acquisition of raw information by spying, the processing of this raw information, and the improving of the reuse process.

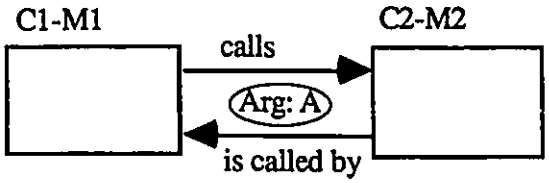
4.1 Acquisition of raw information by spying

In the introduction (section 1.2), we indicated that our system will learn from *reuse by use* by spying on execution of program and from *reuse by modification* by spying on the *copy-paste* actions. In this section we describe the methods used for these two kinds of spying. We also argue that the spied information captures essential knowledge about the two types of reuse.

4.1.1 Spying on the execution of a program

We want to spy on *reuse by use*, which occurs when components are reused as they are. Thus, we have to spy each time a component is used in a program.
OO programming uses *objects*, not algorithms, as its fundamental building blocks. Objects cooperate by sending *messages* to each other. So, to observe how an object is used (*reused by use*), we have to intercept these *messages*.

Suppose that method M1 of class C1 calls (*sends a message to*) method M2 of class C2 with the argument A. We can depict this by the following figure:



For each of these *messages* we store:

- for method M1: - it *calls* method M2 of class C2
  - the class of the argument can be *'Class of A'*.
- for method M2: - it *is called by* method M1 of class C1
  - the class of the argument can be *'Class of A'*.

Since other messages could be sent to method M2, we store information about each call as an element of a list. So we have for M2 an *'Is called by* list'. We also refer to this list as the *'caller* list'. Symmetrically, we also have a *'Calls* list'.

We regard the stored information, not as attributes stored for each method, but as links between methods, as shown in the figure. A link is defined by a name (*calls* or *is called by)* and a value which is the set of possible classes of the argument. We refer to these two links as *execution links.*

4.1.2 Spying on copy-paste

We want to spy on *reuse by modification.* This occurs when a new component is created by modifying existing library components.

The problem with this kind of reuse is that it decentralizes and duplicates functionality, making it difficult to propagate subsequent changes [Micallef 88].

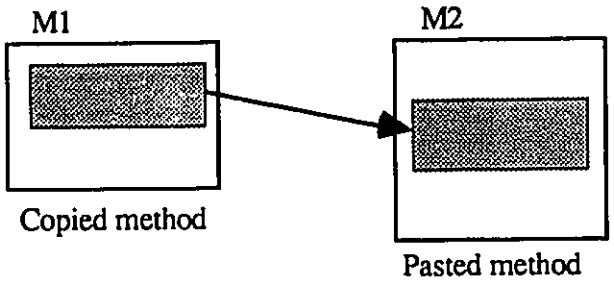
Despite its drawbacks the code "scavenging" (duplicating and modifying) strategy is commonly used, and moreover, it is often done using the *copy-paste* technique.

[Jette 89] states that "the paradigm of copy and specialize for OO programming system is still the most powerful one for the modify part of the reuse process" (p. 100).

[Lange 89] concludes, after observing *in situ* an experienced OO programmer, that "code reuse (*reuse by modification*) dominate other strategies for reuse." (p. 72). Moreover they report that for *reusing by modification* programmers use the copy-paste technique 66% of the time.

Thus, to spy on *reuse by modification* we spy on the *copy-paste* action.

Let's describe this action in detail to introduce the terminology used in this thesis. Suppose the programmer copies some code from method M1 and pastes it in method M2.



We call M1 the *copied method* and M2 the *pasted method.*

We characterize this copy-paste by three numbers:

- *Asize:* absolute size, which is the number of characters copied.
- *Csize*: The relative copy size, which is *Asize* divided by the size of the *copied method.*
- *Psize*: The relative paste size, which is the *Asize* divided by the size of the *pasted method.*

Since programmers are doing copy-paste to build the *pasted method*, its size is not fixed at the time of the copy-paste action. We compute the size of the *pasted method* when the pasted method is compiled.

4.2 Justification of the use of spying

We argue that the information obtained by spying on execution and copy-paste actions cannot be inferred by static analysis, the main alternative to spying.

Our system spies on execution in order to record calls and *classes* of arguments. This information can't be inferred by static analysis because of the dynamic features of OO paradigm, namely, polymorphism coupled with overloading and inheritance (see section 2.3.1).
Moreover, by spying on execution we record actual calls; static analysis states all the possibilities, and those can be quite numerous. Studies [Deutsch 83] suggest that 85% of the time, only one of those possibilities is actually used.

By recording copy-paste action, we capture some user knowledge. By doing this "physical link" with the mouse between two components, the programmer tells us that, in his mind, these two methods are related. Unfortunately, the copy-paste paradigm is so useful that it is not always used in this way. The next section deals with this problem. Static analysis is "blind": strings can be the same (or nearly the same) by pure coincidence. More importantly, similar things can look different: the programmer often changes the variable names and adds or deletes small parts.
Lastly, doing this detection of string similarity is quite costly when you have a "real world" library of components. One can argue that this detection could be done as a background activity. We believe that code can be added faster than even a smart static detection of similarity could manage.

4.3 Treatment of raw data

4.3.1 Copy-paste

In this discussion we assume that a method is the biggest block we can copy: we can copy part or all of a method and paste it in an existing or a new method. This is not a

limitation since one can always decompose a bigger copy-paste into several copy-pastes of this kind.

To see how to detect meaningful copy-pastes, let's consider the different kinds of copy-paste that can be done.

- Copy-paste of anything other than code[1].We discard them.

- Copy-paste of names or small expressions.

It is not worth recording them. Any threshold on *Csize* discards them.

- Copy-paste of one specific operation.

These are characterized by having a relatively small size. We want to record them if the operation has a "significant" meaning at the level of the *method*. One way of measuring if a copy-paste is "significant", can be to see if the 'copy-paste text' is a significant fraction in both the *copied method* and the *pasted method*. A threshold like *Csize* > 25% and *Psize* > 25% discards uninteresting copy-paste.

- Copy-paste of a schema.

This use of copy-paste is shown in [Lange 89]. The method is copied because the new method uses the same structure. It is worth storing it because it is a way of *reusing by modifying*. Moreover, it often means that the functionality of the two methods are similar. This copy-paste is characterized by a big *Csize*, a big *Psize* and important modifications in the paste.

Within the selected threshold (*Csize* > 25% and *Psize* > 25%), we record this kind of copy-pastes.

- Copy-paste of a method.

Here, the user wants to adapt a method or a part of a method to its new application. It is often characterized by a significant *Csize* and *Psize*. This means that the *copied method* and

---

[1] By code, we include comments.

pasted method are similar. We definitely want to record them; the threshold we have set selects them.

- Copy-paste of a set of methods.

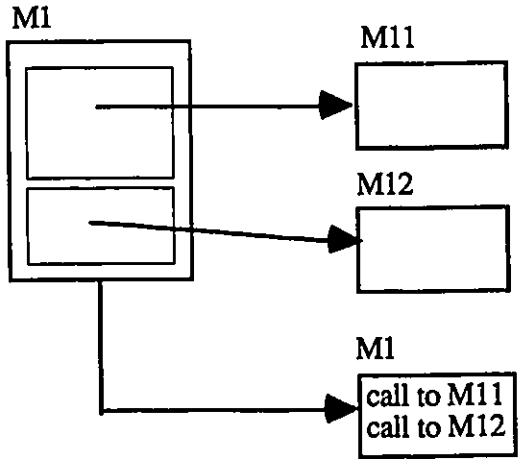
Here, the user wants to adapt a class or a part of a class to his application. It is characterized by several copy-pastes at the method level, so this is the same as for the former case.

- Copy-paste for *splitting* a method into smaller parts.

This occurs when a method becomes too big. The programmer splits it into several smaller methods [Graver 92].

This last operation is not a *reuse by modification*, it is just a new physical distribution of the code.

Let's take an example, suppose that we split method M1 in M11 and M12. We do two copy-pastes (from M1 to M11 and M12); then in M1 we just call M11 and M12.



We do not wish to record these copy-pastes but they are not distinguishable from previous cases until the initial method (e.g. M1) is modified. We will detect its large decrease in size and then erase the corresponding links. The updating of the links is explained in detail in section 4.3.3.

- Copy-paste within a method.

This is a common way to speed up coding when the programmer has to code two similar functions within a single method. This is not worth recording because it can easily be seen when looking at the method code. We detect it and we don't record it.

The resultant algorithm for detecting meaningful copy-pastes is:

if it is 'code copying' **then**

If Csize > 25% **and** Psize > 25% **then**

if it is not copy *within the method* **then**

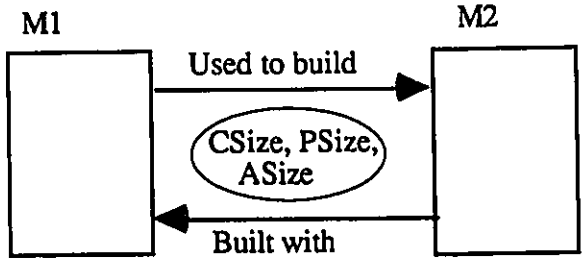
store this copy-paste.

This strategy assumes that after doing a copy-paste, the user will not make changes to the code that will destroy the "similarity". Unfortunately our system doesn't have any way of ensuring this.

If we have a copy-paste from M1 to M2, we store:

- For M1 that: - It was *used to build* M2.
  - The 3 characteristic numbers of the copy-paste (Csize, Psize and Asize).
- For M2 that: - It was *built with* M1.
  - The 3 characteristic numbers of the copy-paste (Csize, Psize and Asize).

As with execution data, we regard this copy-paste information, not as attributes of each method, but as links between methods. We can depict this by the following diagram:



A link is defined by a name and a triple (Csize, Psize and Asize).

We refer to these two links as the *copy-paste links*.

We must also consider the case when a copy-paste action is used, not for building a new method, but for modifying an old one. Suppose that the user does a copy-paste from M1 to M2.

If M2 had no *copy-paste links* there is no problem; we just store this new one.

If M2 had already *copy-paste links*, then we have to distinguish between an *additive paste* and a *destructive paste*. An *additive paste* is when the user adds a new part to a method with the paste. In this case, we just add the new copy-paste information. A *destructive paste* is when the user overwrites part of the existing code with the paste. If the *paste* overrides more than 20% of the code of the *pasted* method we consider that this method is deeply modified. Thus, we erase all the old information of the pasted method and store only this new copy-paste. Our system detects it by checking the variation of size of the pasted method.

if *new method size < 0.8 (old method size + size of the copy-paste text (Asize))*

then it is a *destructive paste*

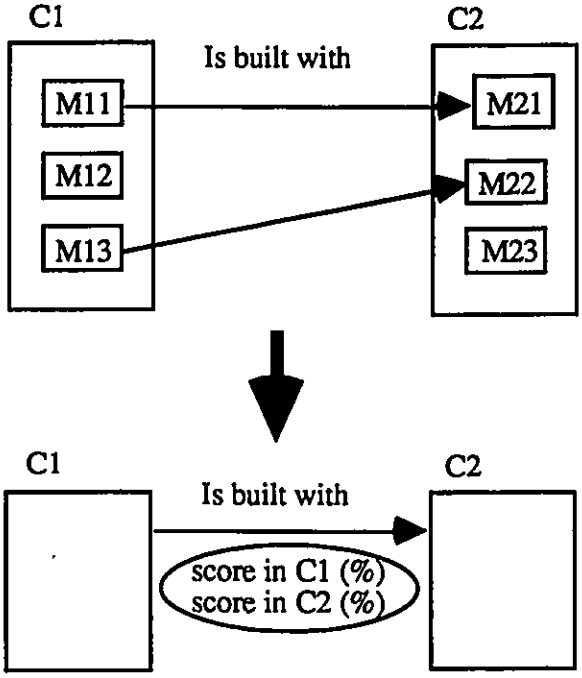
else it is an *additive paste*

4.3.2 Generalization of the information to the class level

We have said that an OO library is composed of classes that can be subdivided into methods. In the preceding sections, all the information (from copy-paste or execution) was stored at the method level. In this section we discuss how to generalize this information to the class level.

4.3.2.1 Copy-Paste

Each time we add a *copy-paste link* at the method level, we try to generalize it to the class level. So if we add the link: C1-M11 is *built with* C2-M21, we try to see if this information is not also true at the class level (i.e. C1 *is built with* C2). To answer this question we need a way of computing the strength of the link at the class level.



There are two main levels of granularity with which one could compute this strength:

- The method level. The measure of the generalized *Built with link* could be of the form:

$$\frac{\text{(number of } \textit{Built with links} \text{ between a method of C1 and a method of C2)}}{\text{(number of methods in the class)}}$$

There are two scores, one for class C1 and one for class C2.

- The character level. The measure of the generalized *Built with link* could be of the form:

$$(\Sigma\ (\text{Asize}_{1,i \dashrightarrow 2,j})) / (\text{size of the class}).$$

where $\text{Asize}_{1,i \dashrightarrow 2,j}$ represents the *Asize* of *Built with link* between a method of C1 and a method of C2. There are two scores, one for class C1 and one for class C2.

In section 4.2, we argued that the copy-paste link carries more information than string similarity (which is detected at the character level). There is some "user's knowledge" represented by the link between two methods. So, if we want this "user's knowledge" reflected also in the generalization, we must use the 'method level' measure.

We establish a link at the class level when the score in one of the two classes is "big enough".

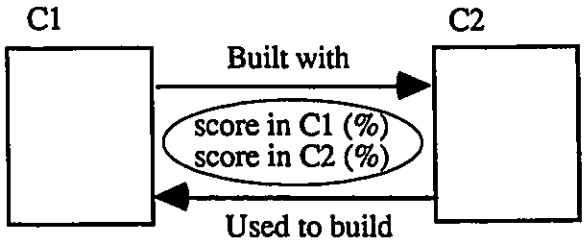
We store for C1 that: - C1 was *built with* C2

- The score in both classes.

We store for C2 that: - C2 was *used to build* C1

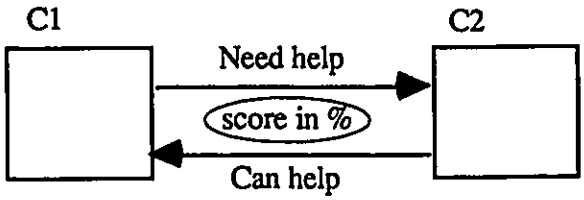
- The score in both classes.

We can also represent the information with the link perspective. We can depict this by the following figure:



4.3.2.2 Execution

Generalizing the *execution links* to the class level is in fact finding the classes that "collaborate". If methods in class C1 often call methods in class C2, that means that C1 *needs help* from C2 and also that C2 *can help* C1.



We want to record this link if these two classes "truly collaborate". An approximate measure to compute the *Need help* link, might be:

(the number of methods of C1 that call a method of C2) / (number of methods in C2)

This measure underestimates the strength because methods in C1 can only call the *interface methods* of class C2. Therefore a more accurate measure would depend only on the number of interface methods of C2. Our score for the *Need help* link is:

(number of methods of C1 that call a method of C2) / (number of *interface methods* of C2)

If this score is "big enough" we store the generalization.

4.3.3 The updating problem

Since we store information, we need to adapt it when a component is modified or erased. It is not our intent to keep track of all changes, instead we do the following.

- When a component is erased, we detect this and discard all its stored information.

- When a component is modified the situation is more complex: we have to see if the information is still valid. Our system detects each time the user modifies a component.

If it is modified by a copy-paste our system distinguishes *additive pastes* from *destructive pastes* as discussed in section 4.3.1.

If the programmer modifies the source code of a component by editing it, our system doesn't have an appropriate way of determining if the stored information is still valid. It simply checks the variation of the component size: if there is a significant decrease, it discards the related stored information. In all other cases, our system just records the fact that the method was modified by storing the component's name on a *'suspicious'* list. The system uses this information to warn users when displaying this 'suspicious' information.

Since it is a crude measure of modification, the system allows the user to manually inform it of an important modification.

The generalizations to the class level are very easy to update: each time we modify information at the method level we redo all the generalizations in which this information is involved.

4.4 Efficiency issues

In chapter 1, we have stated that our system should operate in a "real" OO environment. This imposes two design requirements. Firstly, the spy module should be able to spy on all the development tools the OO environment provides. Secondly, the system should handle big libraries.

We discuss the first requirement in the next chapter (see section 5.2.1) since it is highly dependent on the type of the environment.

The fact that our system should handle big libraries raises three design issues:

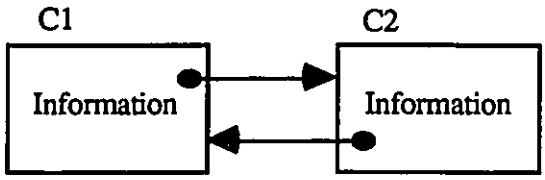
- How to access and update the stored information efficiently?
- How to have a reasonable loss of speed when spying on the execution?
- When to generalize information?

We discuss these questions in turn.

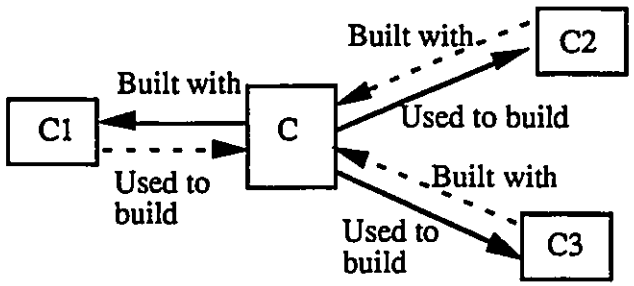
a] How to access and update the stored information efficiently?

Our information always involves two components, say C1 and C2. It is possible to store information in only one component, say in C1. But when we want to access information about C2, we would need to look at the information of every component in the library. Since we need to access this information frequently (display, update, generalization ...), even an operation whose complexity is linear with the number of components of the library is undesirable.

The solution is to store information in both components and to have symmetric links between them:

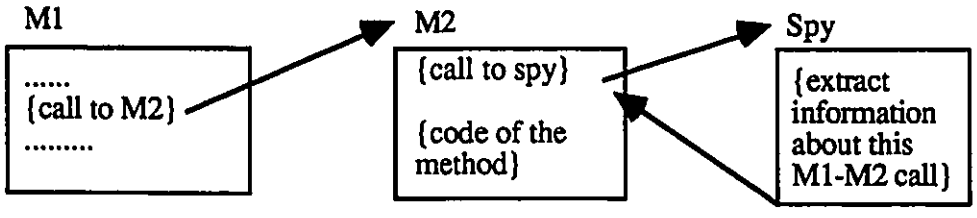


For the same reason, when we want to erase information about a component, we use these "symmetric links". Suppose that a component C has the copy-paste links shown in the next figure. If we want to erase all its copy-paste information we move via the plain arrows, into components C1, C2 and C3, and erase the dashed arrows.



b] How to have a reasonable loss of speed when spying on execution?

To spy on execution we intercept messages. We can't intercept all of them because this would degrade the execution speed too much. Our system requires the user to specify which classes he wants to spy upon. When he decides to spy on a class, the system automatically inserts at the beginning of every method of this class, an instruction that calls our "spy method". This method extracts the raw execution information. For example if the system spies on method M2, we have the following:



When the user decides to stop spying on this class, our system automatically removes this instruction.

While is true that it is intrusive to require the user to decide what and when to spy, we do not regard this as a large burden for the user. Our system does not need that many "spy sessions". The first one can be done in the final testing of a class; there the user often tests the full functionality of this class and therefore allows our system to learn about nearly every expected operational condition. Then, another one is needed only if the class is used in another context or if it is modified.

To overcome the drawbacks of not spying all the time, our system keeps an history of the "spy sessions" for each class. With such a list, the user can judge the validity of the execution information and whether or not this class needs to be re-spied.

c] When to generalize information?

We have said that the generalizations to the class level are very easy to update: each time we modify information at the method level we redo all the generalizations in which this information is involved. But this process can be time consuming. For the copy-paste generalization this is not an issue since users do copy-pastes at a slow rate.

For the execution information it as an issue: we can't generalize each time we intercept a message. The solution we have adopted is to update the generalization at the end each of "spy session".

4.5 Summary of the stored information

This array summarizes all the information stored by our system.

		method level	class level
reuse by use	link name	Calls - Is called by (Caller)	Need help - Can help
	Information	Argument class (if any)	score (in %)
reuse by modification	link name	Used to build - Built with	Used to build - Built with
	Information	Csize (in %), Psize (in %), Asize (# of char.)	score in each class (in %)
Updating	Information	suspicious method ?	Dates of "spy sessions"

Figure 2: Summary of the stored information

4.6 Use of the stored information to facilitate the reuse process

In this section, we show how the stored information is used to facilitate the three phases of the reuse process: searching, understanding and adapting.

4.6.1 Searching

We ease searching, more specifically browsing, by introducing a "similar" relationship and by allowing an easy location of "good reusable" components (methods or classes).

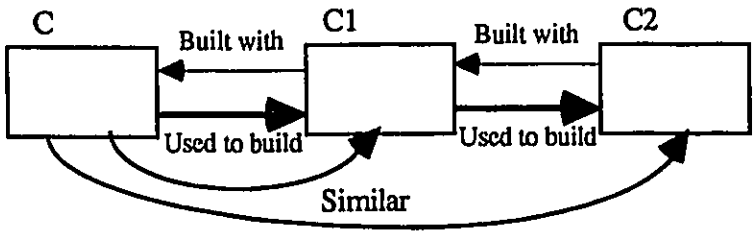
4.6.1.1 Adding a "similar" relation

Similarity is a very natural relation: when a user has found a component that is somewhat close to what he is looking for, this relation allows him to see if there is a closer or even a perfect match.

[Fisher 91] proposes this kind of search by allowing the user to refine a query using data associated with the first retrieved items.

The system in [Constantopoulos 92] supports the notion of similarity by two links called *similar* and *derived from*. Their *similar links* join two different levels of abstraction of the same object and their *derived from links* join two different versions of the same object.

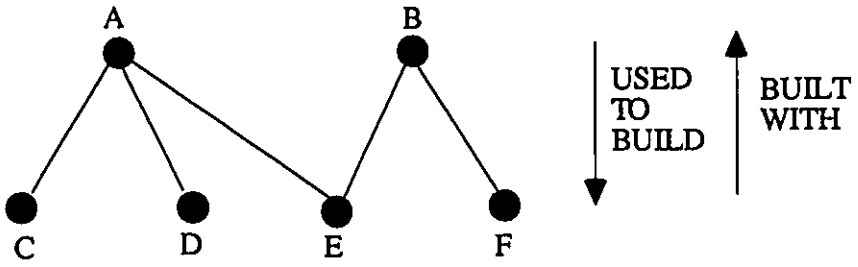
Our system provides, for each method, a list of similar methods ordered by their "degree of similarity". Our system computes this list by going recursively from the selected method/class through the *copy-paste links*. For each link, we compute a score depending on the relative size of the copy (*Csize*) and the relative size of the paste (*Psize*). If the similar components are not directly linked to the selected method/class, we multiply the score of every link of the chain. We also have a threshold to display only reasonably similar components.



In this example, component C is similar to C1 and C2 because C is *used to build* C1, which in turn is *used to build* C2.

4.6.1.2 Locating reusable components

To store all the spied information, we have created directed links at the method and class levels. These links create 3 kinds of directed graphs on components: one for each type of reuse. In each graph an component is connected to others that are conceptually close with respect to a given type of reuse. Let's take an example from the "reuse by modification graph". Suppose that we have adapted (*reuse by modification*) class C and D from A, class F from B and class E from A and B. We obtain the following graph:



The user can navigate through these graphs to locate reusable components, close enough to his needs. To draw the user's attention to reusable classes, we display a "reuse score" which represents how many times this component has been reused. This score reflects how reusable the component is.

For *reuse by specialization*, we use the existing inheritance graph. This graph is limited to a tree and it only exists at the class level.

4.6.2 Understanding

To understand code in a normal OO environment the programmer has three main choices: inspect the source code, run it or use the "caller"[1] facility. We have shown that static inspection of code is hard. [Nielsen 89] even states that "Inspection of static code is probably the least effective way to understand Smalltalk. Even novices should use a run-time source

[1] This is a standard feature. In Smalltalk it is called the 'sender'. In objective C it is called "is used by methods".

debugger " (p. 76). Executing the component and opening a debugger is not a satisfactory solution because:

- The user has to know how to execute and interact with this component.

- The user won't explore all the possibilities.

- It is time consuming and complex since you have to trace a flow of sent messages.

The "caller" facility is supposed to give the user a list of all the callers of an element. Unfortunately, it is done statically and the list often contains false information. Section 6.1.3 illustrates this point.

The next three subsections discusse the three improvements to understanding our system brings.

4.6.2.1 Overview of a component

Often a quick overview of the components is all that it needed for a user to decide if the component has some chance of meeting his requirements.

This abstract view of a component should provide the key information to allow a quick but complete overview of it.

[Booch 91] describes an OO design method in which such abstract views for classes and methods are proposed. It calls them "templates".

"A class template captures all important aspects of a class" (p. 164). It proposes the following template for classes:

- *Name.*

- *Documentation* in a free text form.

- *Cardinality*: the number of possible instances. It can be 0, 1 or greater than 1.

- *Inheritance hierarchy.*

- For each category of method (*public, protected and private*), they are three fields:

- *Uses*, which is a list of class names.

- *Variables*, which is a list of variables used in this category.

- *Method names.*

He also has specific fields which are inapplicable to the language we use: Smalltalk.
He proposes, as a method template, "to state the *name of the operation* (method), its *parameters*, and its *meaning* (using free text from)." (p. 166).

We provide both these templates except that we provide the *Uses* field at the method level and we provide the *variables* field at the class level.

A lot of this information can be extracted using the facilities an OO environment already contains, such as a parser or extraction of the inheritance hierarchy. We *reuse by use* these facilities.

Since our aim is to ease reuse, it would be a good idea to have abstractions related to reuse.
As we have already mentioned, for each component (method or class) and for each kind of reuse, we display a score reflecting how reusable the component is.
Moreover we provide a notion of "reuse effort". This was introduced in [Prieto Diaz 87] with two main criteria: complexity of the structure and quality of documentation.
In our system the complexity of a class, say C, is represented by:

- The size of C (in character).

- The number of methods in C.

The quality of documentation is represented by:

- The ratio: size of all the comments in C over total size of C.

- The number of methods documented.

4.6.2.2 Understanding the context

In this section we first show the importance of context in the OO paradigm and then explain how our system helps in understanding this context. We can define the *context* as the relations an object has with the rest of the library.

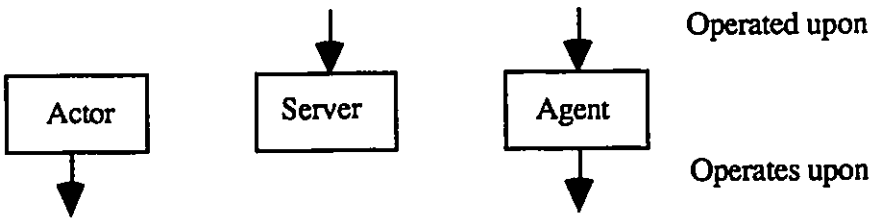
[Wasserman 91] points out that "An *object* is rarely reused by itself, but in a well-understood context of surrounding, related objects." (p. 55). It states that to reuse a component the programmer must understand, and sometimes modify, the context.

[Beck 89] introduces the *CRC cards* to describe context. These cards, simple 4" * 6" sheets of paper, show three key characteristics for a class: its name, its *Responsibilities*, which identify the role the class plays in the problem to be solved, and its *Collaborators*, which are the classes that help it to carry out its responsibility. The Responsibility-Driven approach [Wirfs-Brock 89] is based on the same idea: in it a program is viewed as a client/server model.

In our system, the user can see the *Responsibilities*, at the method level, with the *Callers link* and, at the class level, with the *Can help link*. He can see the *Collaborators*, at the method level, with the *Calls link* and, at the class level, with the *Needs help link*.

[Booch 91] classifies methods depending on their role in the context. The *Interface methods* of a class are the methods that can be called by other classes (outside view). In contrast, *implementation methods* can only be called by their own class. *Interface* methods are further divided into two groups: *public methods*, which are visible to everybody, and *protected methods*, which are only visible to *subclasses*. [Booch 91] also distinguishes three types of role for classes. An *Actor* is a class that can operate upon (*Needs help of*) other classes but does not operated upon by (*Can help*) other classes. A *Server* is a class that is only operated upon by other classes. An *Agent* is a class that can both operate upon other

classes and be operated upon by other classes. This can be summarized in the following figure:



These notions help the user to understand the role of the components. Unfortunately they are not part of most OO languages such as Smalltalk, Object Pascal and CLOS.

Our system computes all these notions. To compute the method roles, since we record all the *callers* of each method, we just have to compare this list of calling classes to the class, say C, in which the method is implemented.

If all the *callers* are in class C

then it is a *private* method

else if all the *callers* are in subclasses of C

then it is a *protected* method

else it is a *public* method.

We use a similar algorithm for computing the *roles* of the classes.

4.6.2.3 Understanding code

We have shown that it is hard to understand code by static inspection. Yet, that is an important way for programmers to understand programs.

To solve this issue, as [Ponder 92] suggests, we provide new information.

For each method the system displays:

- The *callers* of the method (*Is called by* link).
- The possible classes of the arguments for calling it.
- The *calls* the methods perform (*calls* link).

Because this information is collected by spying an actual calls to the methods, our system only displays possibilities that actually happened.

For each class, the system indicates if it is an *abstract class* or not. [Deutsch 89] describes abstract *class* as "Single-class frameworks": classes that supply a partial implementation and expect subclasses to complete the implementation. The OO languages don't provide any help to determine whether it is an abstract class or not. Though, it is important for users to know if a class is abstract: users can't create an instance of an abstract class, they must use one of its subclasses.

4.6.3 Adapting

In order to ease the adaptation of a component, our system provides previous examples of adaptation, allowing users to build their components by analogy with these examples.

For a given component, we display:

- Where and how it has been used (*reuse by use*).
- If it has subclasses (*reuse by specialization).*
- If some other components have been created by using the code of this component (*reuse by modification*).

Several authors have mentioned that these examples of previous reuse can play a crucial role in understanding how to reuse and allow adaptation by analogy.

[Lewis 87] states the importance of examples to adapt: "When grappling with new material, learners often try to adapt examples of earlier material to suit the present situation, modifying the example by analogy." (p. 9).

[Lenat 86] points out the importance of this analogy. "We assimilate new information by finding similar things we already know about and recording the exception to the analogy." (p. 66).

Another problem occurring in a system with a large library is that there are many possible ways of achieving the same functionality using different components. [Nielsen 89] calls this the "hacker trap": "Providing many features can result in an overwhelming problem of choices" (p. 75). By storing examples of use we can show the user the most common choices.

4.7 Conclusions

In this chapter, we have presented a general model for a learner in an OO environment. We have discussed methods to acquire raw data by spying on programmers' reuse actions. We further described how to process this raw data to produce meaningful information. The latter is then generalized and kept up to date.

Finally, we have discussed how this model can improve searching for, understanding and adapting components.

Chapter 5: LEARNIST: an implementation for Smalltalk

This chapter describes an implementation for the abstract model discussed in the previous chapter. This system, called LEARNIST (Learning to EAse Reuse IN SmallTalk), is implemented in Smalltalk-80[1] release 4.0. A good introductory book to Smalltalk-80 is [Goldberg 89].

The Smalltalk language is well suited our needs for two main reasons. Firstly, it is a "pure" OO language: everything is treated as an *object*. This coherence and uniformity greatly simplifies the spying process. For example, in every Smalltalk editor, the copy-paste facility is done in the same way. We need only to spy on two messages: the 'copy' message and the 'paste' message. Secondly, the source code is available for the whole development environment (e.g. for the browsers, compiler ...). Our technique for intercepting messages requires this source code.

We mention for readers familiar with Smalltalk that in the two following chapters, we don't distinguish the *class* from the *metaclass* (e.g. the class 'Foo' from the metaclass 'Foo class'). This is done to avoid confusing readers without a Smalltalk background.

---

[1] Smalltalk-80 is a trademark of ParcPlace systems.

5.1 Overview of the system

Figure 3 is a block diagram of the LEARNIST main components.

This system was designed form a modular perspective: each of these "blocks" are essentially independent.

- The "learner" spies and processes raw information.
- The "spy controller" consists of an interface to setup execution spying and a module to turn on and off the spy process.
- The "storer" stores the processed information.
- The "updater" keeps all this information up to date.
- The "display" module is subdivided into:
    - The *ReuseBrowser*, which is the user's interface.
    - The *ReuseInfo*, which is an interface between this *ReuseBrowser* and the stored information. The purpose of this module is to allow better modularity and to ease future reimplementation in other languages.

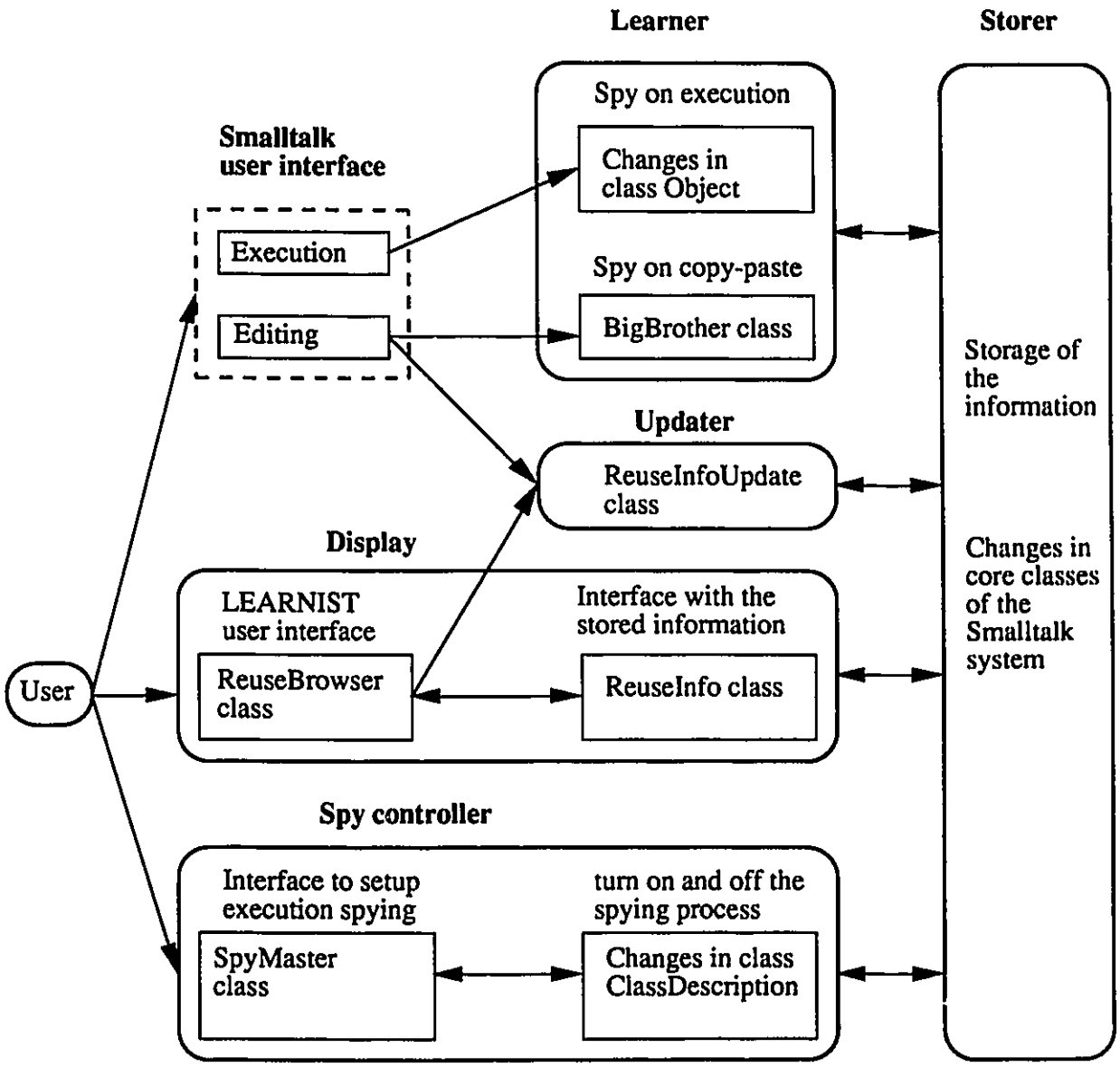


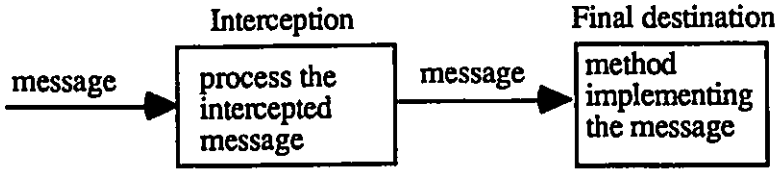
Figure 3: LEARNIST structure

5.2 Overview of the main techniques

5.2.1 Intercepting messages

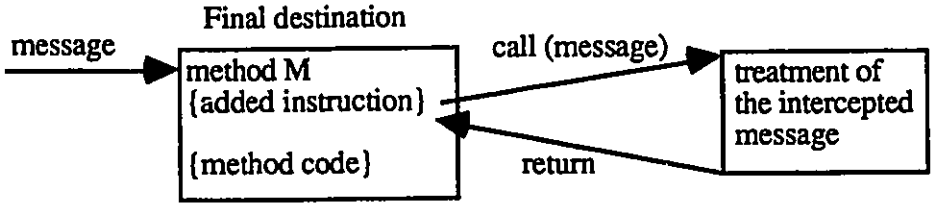
The spying process is based on intercepting messages. This is done in two main ways:

- Intercept the message before it reaches its final destination.



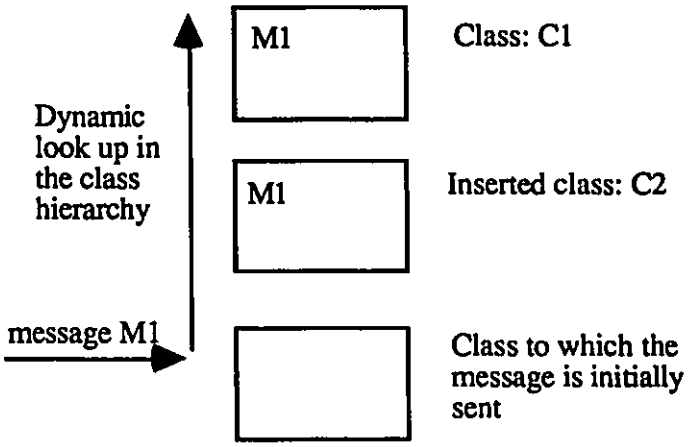
That is how we intercept the copy-paste actions.

- Intercept the message just after it reaches its final destination (method M). To do this, we add an instruction at beginning of the method M in order to call the spying method that processes the interception:



Spying on execution is done in this way.

The first possibility is the cleanest because we don't have to modify an existing method: we just add a new one. Suppose that we want to intercept message M1 and that the method implementing this message is in class C1, we have:



This technique has two drawbacks. We can only intercept messages that perform a dynamic lookup over the superclasses. Furthermore, we must insert a new class in the class hierarchy (e.g. C2).

In either case it is important that messages be intercepted at the system level. We shouldn't intercept the message at the user interface level since the message is dependent on the tool being used. To intercept messages from any tools of the user interface, even future ones, we have to intercept messages at a lower level: the Smalltalk system level. Due to the coherence and uniformity of Smalltalk, every tool within this environment uses (reuse by use!) the same core Smalltalk methods to implement a given functionality.

For example, let's explain the steps involved in intercepting the message sent when the user removes a class.

First we have to locate the message sent when the user asks the normal Smalltalk browser to remove a given class. The corresponding called method is *removeClass* in class *Browser*. Then, we have to analyze the method to see how it is implemented. It first confirms the user wishes to remove the class, and then it calls the method *removeFromSystem* in class *Class*. In turn, this method calls method *removeClassNamed* in the class *SystemDictionary*. This last method actually removes the class and also all its subclasses. So to intercept the 'remove

class' action we must insert into this *removeClassNamed* method a call to inform LEARNIST that this class and all its subclasses have been removed.

We have to be careful not to go too deep into the system. In our example, the spied method, *removeClassNamed*, calls method *removeElement* that actually performs the removal. But we should not intercept messages in this method since the Smalltalk system also calls it while managing private processes (i.e. moving classes).

One particular experience showed that LEARNIST can intercept messages even from future user interfaces. Another research group of the university had added a new browser tool called 'FullBrowser'. We have done some experiments with this new tool: without any modification LEARNIST was able to spy on programmers using it.

As a practical consideration, before determining at which level to intercept the message, we must determine which original message is sent when the user performs a given action (for example to copy something).

5.2.2 Robustness

To make LEARNIST robust, it never uses the stored information without first testing if the information exists, if it has the right format and if it is still valid. By valid we mean that the information contains only links to class or method which are still in the system.

These failures in format or validity of information should not happen but if they do, we handle them either by skipping the abnormal information or by erasing it.

LEARNIST also allows the developer to reset information for each particular fields.

LEARNIST also automatically resets some of its internal variables. An example is given in section 5.3.2.

An example of a failure occurred when developing LEARNIST. We had forgotten to intercept the 'rename' action, so when we did a rename operation, LEARNIST hadn't updated the reference to this class in the rest of the stored information. The system did not crash.

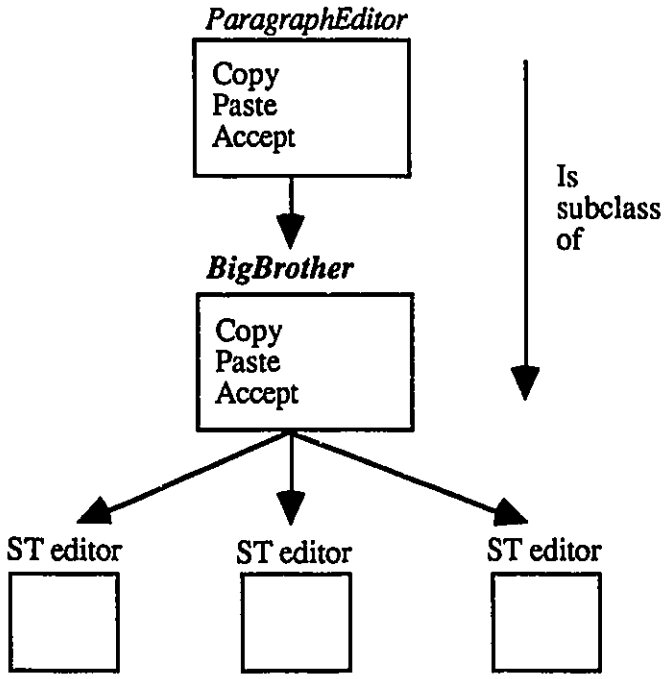
Later on, when LEARNIST tried to manipulate this outdated information (with the old class name), it detected that the class name was invalid and erased the corresponding information.

5.3 Learning information

5.3.1 Learning from copy-paste

The copy-paste spying is done with the *BigBrother* class.

Due to the uniformity of the Smalltalk system, the copy-paste facility is the same for every editor. The copy-paste function is implemented in one class (*ParagraphEditor*) of which every Smalltalk editor is a *subclass*. We spy on copy-paste by intercepting the *copy*, *paste* and *accept*[1] messages. In order to do this, we insert a class, named *BigBrother*, between the class *ParagraphEditor* and the Smalltalk editors.



[1] Accept is the message sent to compile a method.

We gave the general algorithm of the processing of copy-paste in section 4.3.1. The implementation is not so straightforward as it must cope with numerous details. Since one of our design requirements was to spy on the programmer within a real development environment, we have to take into account the facilities offered by the Smalltalk environment:

- We can *paste* a text from the last five copies.
- We can *cancel* a compilation and return to the previous state.
- We can work with different kinds of editors.
- We can use more than one editor at the same time.

The first issue was handled by creating a structure containing the last five copied texts and, at each paste, LEARNIST checks to which copy it corresponds.

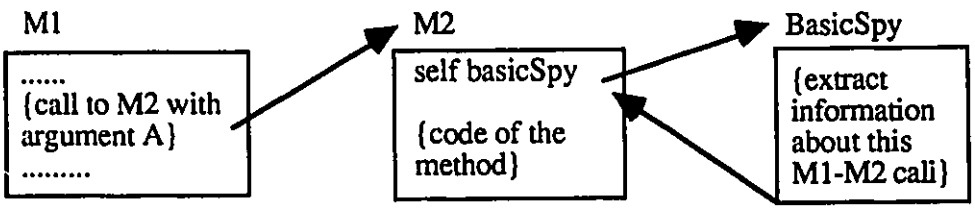
The second issue was solved by erasing the information corresponding to the cancel.

The third issue was handled by intercepting the messages at the Smalltalk core level and by using the protocol common to all editors to extract the needed information.

The last issue was solved by recording with which editor the copy, paste and accept actions were performed. Thus, when the user compiles a method (with accept) we know how to extract the copy-paste related to this method.

5.3.2 Learning from execution

To spy on the execution of method M2, we intercept every message sent to M2. This is done by adding one instruction ('self basicSpy') at the beginning of M2. This instruction calls the method *basicSpy*, implemented in class *Object*, the top most *class* in the hierarchy.



This method is a core method of the LEARNIST system. To extract information about a call, it must analyze the contents of the system's execution stack.
The reader can see the actual code of this method in appendix A.

We give in the following a sketch of the way we have proceeded to implement this method.
We first looked at the Smalltalk class Debugger, which manipulates this kind of stack. Unfortunately, the debugger is implemented in such a way that it simulates execution. By simulate we mean it doesn't proceed as is normal in Smalltalk. Thus, we had to explore the creation of the debugger. We explored classes like *NotifierView*, *Exception*, *InteractiveCompilerHandler* to figure out how to access this execution stack. We can access it through a 'context variable' called *thisContext* (it is the only context variable in the entire Smalltalk system!). The only description given in the manuals is that "it is a reference to the stack context of the current process" and it "is rarely needed by application developers" [ParcPlace 90, p. 27].
The next step was to understand how to manipulate this context variable. We have to know the detailed structure of this stack of context information. When we reach the appropriate context we have to extract the calling method (e.g. M1 in the previous figure) and its class, the called method[1] (e.g. M2) and its class, and the arguments of the call (e.g. A). The extraction of the arguments is difficult as a *context* has no way to distinguish between 'argument variables' and 'temporary variables'. The following code tests if it is an argument variable:

---

[1] In fact it is a bit more complex: we have to distinguish between the *receiver* class and the class that implements the method. Those classes are different if there is a dynamic look up through the hierarchy. Our system records both of them.

'(context localScope variableAt: name) class = ArgumentVariable'

To see the complexity behind this line of code let's explain it in detail.

context is an object representing the context of the called method (M2 in previous figure). We sent it the message *localScope* to give us an instance of class *LocalScope* containing the argument and temporary names for this method (note that it also includes compiler-generated temporary variables). We have added the next message, *variableAt:*, to access the type of the variable with its name. Finally, we test if its class is *ArgumentVariable*.

We also have to prevent infinite recursion for this *basicSpy* method: we don't want to spy on a call that *basicSpy* performs. We avoid this by having a global variable set to *true* when we are executing the *basicSpy* method. LEARNIST tests this variable to determine if it should spy on a message or not.

For robustness, this variable is automatically reset to *false* when the user starts or stops a "spying session".

5.3.3 Control of the spying process

As we said in section 4.4, for efficiency, the user tells LEARNIST which class he wants to be spied. This is implemented in *SpyMaster* class.

The look of the *SpyMaster* interface is shown in the following figure:

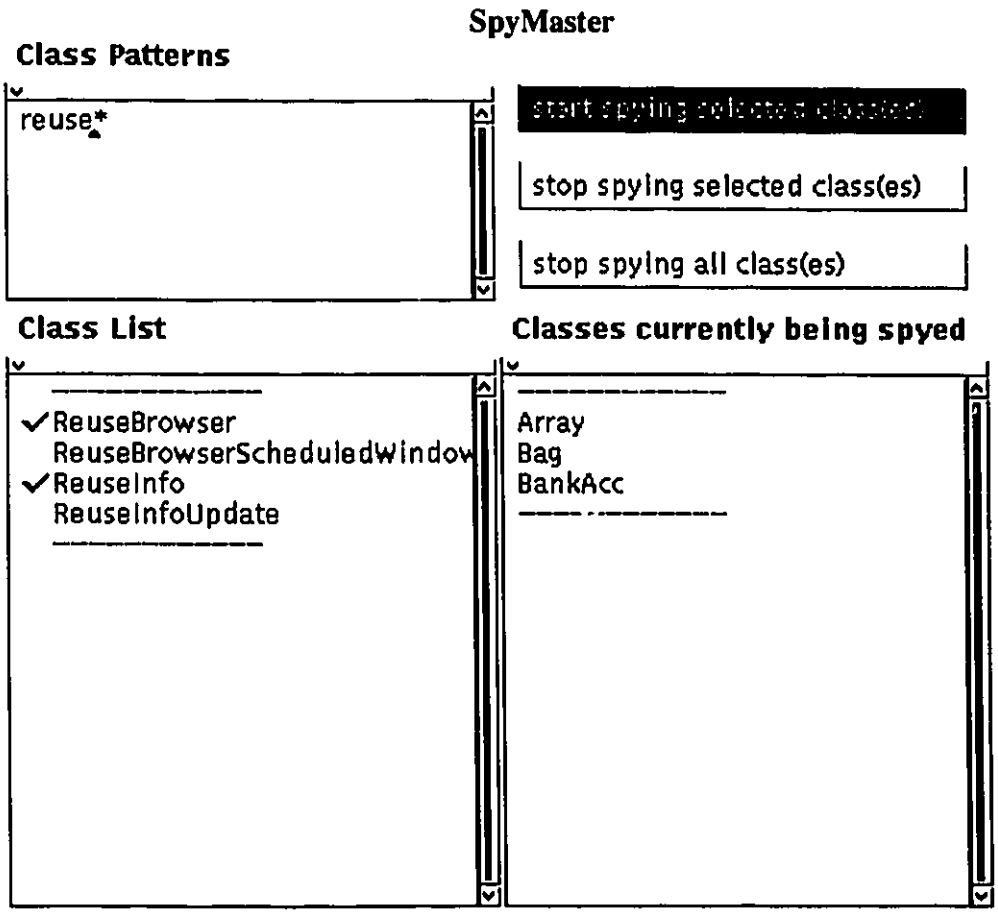


Figure 4: Interface of the Spy controller

In this figure, the two left panels let the user select classes. In the 'Class pattern' panel, the user has requested a list of all the classes that have a name starting with 'reuse'. The 'Class list' panel displays this list, and lets the user select the classes. Here, the *ReuseBrowser* and the *ReuseInfo* classes are selected.

If the user presses the 'start spying selected class(es)' button in the upper right panel, the system starts spying on the selected classes. LEARNIST provides two ways of stopping the spying process: The user can stop some particular classes (with the same selection process) or he can stop spying on all classes.

The bottom right panel displays all the classes that are currently being spied upon.

To spy on one class our system automatically inserts the spying instruction (*self basicSpy*) in every method of the class. In order to do this automatic insertion, we have to parse the method code to locate where to insert this spying instruction. It has to be inserted after the method name and possible comments or temporary variable declarations. Then, we have to recompile the methods. When recompiling these methods we have to address two problems. The first one is, not to record this modification in the LEARNIST's *suspicious* list. The second one is related to a feature of Smalltalk: it records, at each compilation, the text of the method being recompiled. Smalltalk allows the user to use this for different purposes (e.g. for recover changes after a system crash). We don't want this recompilation to be recorded and so, LEARNIST must directly access the compiler with its own special compilation message.

To stop the spying process, the system automatically removes this instruction using a similar procedure.

5.3.4 Class and method abstractions

In section 4.6.2.3 we have said that we obtain the abstraction information mainly by *reusing by use* the OO environment features. In this section we show how we have extracted the information needed to produce the *reuse abstraction* of a class. For that, we need to extract the size of the class, the number of methods, the size of all the comments and the number of methods documented.

This is done in method *reportInfo:For:On:* in class *ReuseInfo:* (See figure 5). In this method we reuse the Smalltalk knowledge about classes and the Smalltalk parser. The operations of the method is designated in comments on the right side of the following figure.

```
reportInfoFor: aClass on: stream                    method name and the two arguments of the call.
"@@     aClass: too many (>5)                       Type comments automatically inserted
        stream: TextStream  @@"                     by the LEARNIST system.
" report the reuse info for this aClass"            programmer's comment.
| sizeComment nbMethods sourceSize sizeSource parser thisComment nbCommented |
                                                    instance variables declaration.
"initialize the counters"
sizeComment := nbMethods := sizeSource := nbCommented := 0 .
"take in account the class comment"
        sizeComment := sizeComment + aClass comment size.
                                                    We ask the Smalltalk system for
                                                    the comment of
                                                    the class aClass and then its size.
"compute all the info for the class"
aClass selectors do: [ :selector |                  'aClass selectors' instruction
                                                    asks the Smalltalk system
                                                     for the methods of this class.
        "for each method of the class"
        nbMethods := nbMethods + 1 .
        "get the source code for the selector"
        source := aClass sourceCodeAt: selector.    Ask the Smalltalk system for the
                                                    source code about a given method.
        source isEmpty ifFalse: [
                " record the size of the source"
                sizeSource := sizeSource + source size .
                " open a parser on the class"
                parser := aClass parserClass new .  Use the Smalltalk parser ...
                "parse the selector"
                parser parseSelector: source.
                "parse the comment"                 ... to parse the comments.
                thisComment := parser parseMethodComment: source setPattern: [:none | none].
                thisComment isEmpty ifFalse:
                    [ nbCommented := nbCommented + 1.
                      thisComment do: [:temp | sizeComment := sizeComment + temp size ].
                ]]] .
```

Figure 5: Illustration of the 'reuse by use' of the Smalltalk features

5.4 Storage of information

All the information about a class or a method of a class, is stored in the class itself. This has three advantages over storing the information in an external structure:

- It is easier to access the information, since as soon as you have the component, you also have its information. For example, the code to access the list of classes that *can help* class C1, is: *C1 canHelpInfo.*
- It is easier to update the information. For example when the programmer erases a component, he also automatically erases its information.
- If you move a component to another platform, you still have its information.

At the class level, we store all this information in a variable called *classInfo*. It is an instance variable of the class *ClassDescription*. Since all the other classes are subclasses of this one, they all inherit this variable.

Each time the programmer defines a new class, our system detects it, and automatically initializes this *classInfo* variable with the fields shown in the following figure. The detection is done in three methods of class *Class* since the Smalltalk system distinguishes three ways to create a subclass (at the programmer's level there is only one choice).

- *builtWith*	: *builtWith - Is used to build* link. It is a dictionary: keys are the linked class and elements are scores in %. The value is computed by generalization from the method level copy-paste information.
- *derivedFrom*	: Reverse link of build with.
- *helper*	: *Helper* link. The value is computed by generalization from the method level execution information It is a dictionary: keys are the linked classes and elements are scores in %.
- *canHelp*	: Just the reverse relation of *helper*
- *changes*	: *Suspicious* list. It is a *Set* of method names.
- *spySessions* :	Store the dates when the class has been spied. It is an *OrderedCollection* of *Date* and *Time*
- *abstract*	: Indicates if the class is abstract or not.

At the method level, things are a bit more complex. The Smalltalk system uses a 'method dictionary'[1] to manage the methods of each class. To store the methods information, we create a "mirror" of this dictionary called *methodInfoDict*[2]. By mirror, we mean that the keys of the two dictionaries (which are the names of the methods) are the same. If the programmer adds a method, a new key is created in both dictionaries. If he erases, moves, renames, ... both dictionaries behave the same. The difference is that for each key (i.e. each method) in our dictionary, we store the spied information about this method. This "mirror" of the 'method dictionary' is done by intercepting all the messages that modify this dictionary (e.g. adding a method) and modifying our *methodInfoDict* in an appropriate way.
We also provide methods to manage all this information.

---

[1] This dictionary is stored in *methodDict* which is an instance variable of the class *Behavior*.

[2] *methodInfoDict* is an instance variable of the class *Behavior*.

5.5 Updating

5.5.1 Method level

As discussed in section 4.3.3 we erase the information about a method if:

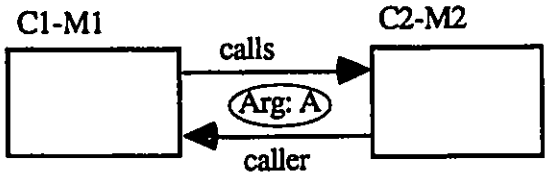
- The method is removed.
- Because of a modification, there is a large decrease in the method's size.
- The user specifies to do so through the LEARNIST interface.

The following is the general algorithm for removing information at the method level:

*** Remove execution information for method M1**

*** Remove execution reference to M1 in the rest of the system**

Recall that a link between two methods is stored in both of the methods involved in the call.



So if we erase M1, we need to go in M2 to erase the *caller* information concerning M1. From M1, we follow the *calls* link to go in M2 and then erase the reverse link (i.e. M1 is a *caller* of M2).

In order to make this process robust, LEARNIST checks if there is a lack of symmetry in the *calls-caller* links. If there is this abnormality is recorded in a "developer file".

Since we have modified some information in method M2, we need to update the generalized execution information for the class containing M2.

*** If it is the user who has requested the removal of this information,** M1 won't be removed, so we also have to erase the execution information in M1 itself.

*** Update the generalized execution information for the class containing M1.**

*** Remove copy-paste information for M1**

{Similar treatment}

In Smalltalk there is no facility to "move" a method. Instead, the programmer uses copy-paste: he copies the method, pastes it in a new location, and then erases it from the old location. Copy-paste does not copy the information about a method, so this technique for moving a method would cause the information about the method to be lost.
To detect such a copy-paste, when LEARNIST detects an erase action, it goes through the *used to build* list (of the 'old method') to see if there is a copy-paste where *Csize* + *Psize* is greater than a threshold (180). If so this means that the programmer has just duplicated the code. It is true that this way of proceeding does not distinguish between moving a method code or duplicating it. However in either case the new method is a nearly a verbatim copy of the old one, and so it is desirable to move the spied information to this new method.

5.5.2 Class level

Since we generalize the method information to the class level, we need to update the generalization when the user adds or removes a method or when our system updates the information about a method. The former is achieved by detecting the add and remove user actions: we intercept messages *addSelector:withMethod:* and *removeSelector:* in class *Behavior*. The latter was discussed in the preceding section.

When a class is removed, LEARNIST intercepts the "remove class" message and erases all information about the class and all its methods.
Like at the method level, we have to erase all references to this class.
The detection of this message is explained in section 5.2.1.

When a class is renamed, our system detects this, and moves all the class information to the renamed class and updates all references with this new name. The detection is done by intercepting the message *rename:* in class *Class*.

5.6 User interface

The user interface is mainly achieved with a new browser: the *reuse browser*.
This *reuse browser* is linked to a normal Smalltalk browser: the former displays the reuse information and the latter displays the code. When we browse (i.e. go to another class/method) in either one of them, the other is updated automatically.

The display of the class of argument is so useful that we have integrated it as "type comment" in every method. This is explained in section 5.6.2.

5.6.1 The reuse browser

Figure 6 shows the user interface called the *reuse browser*.
In this section we explain the role of each panel, and in the next chapter we describe how they are used.
Figure 7 is a schema showing the panels' names.

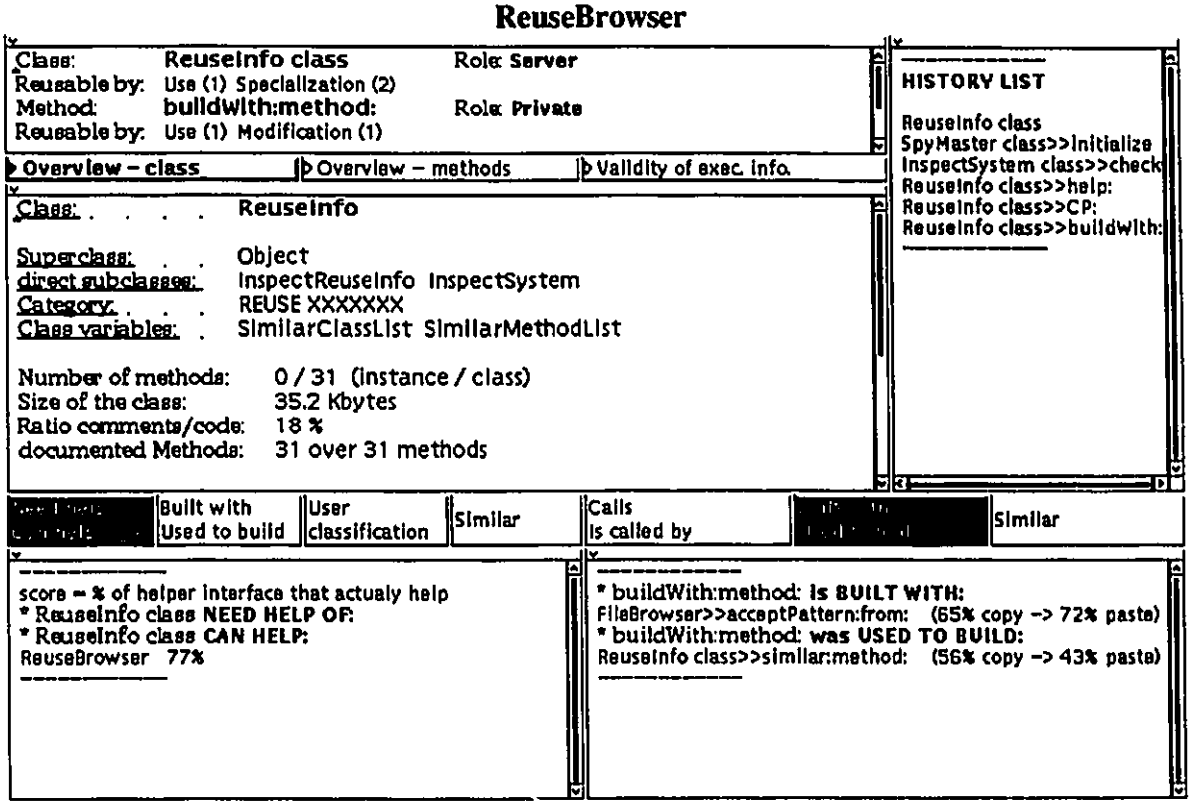


Figure 6: The reuse browser

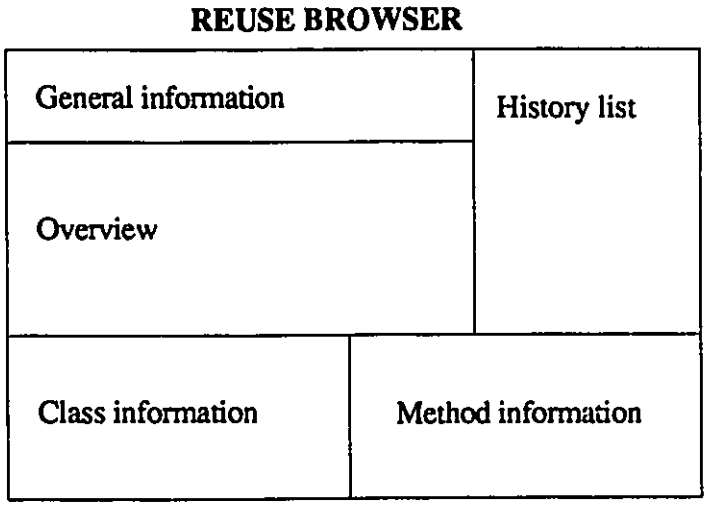
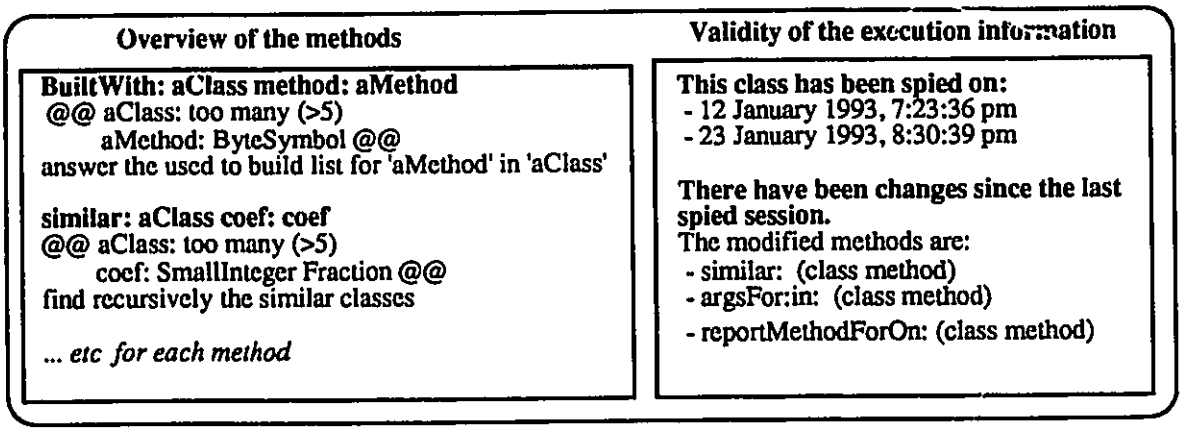


Figure 7: Names of the panels in the reuse browser

The top-left panel displays the **general information.** It shows that we are currently looking at the method *buildWith:method:* in the class *ReuseInfo.* It also displays crucial information about their roles and their potential for reuse. The latter is achieved by showing a 'reuse score' for each type of reuse. This score is the number of times the component has been reused. For example, one other class *reuses by use* this 'ReuseInfo' class and two other classes reuse it by *specialization.*

The middle left panel gives an **overview.** With the buttons the user can select:

- The overview of the class. This is shown in figure 6.
- The overview of the methods. See following figure.
- The validity of the spied information. See following figure.



The three other panels display lists of components. We can explore a component just by clicking on it. The *reuse browser* automatically goes to this component and displays its information (the associated Smalltalk browser does the same and displays its code).

The bottom left panel displays all the **class information.** With the buttons the user can select:

- The *need help - can help* information. The two lists are ranked according to a score. This score is represented by the percentage of the *interface methods* called. In

figure 6, we can see that the current class, *ReuseInfo*, can help class *ReuseBrowser*. The score means that 77% of the interface of *ReuseBrowser* is called by *ReuseInfo*.

- The *built with - used to build* information. The two lists are ranked according to a score. This score is represented by two numbers: the percentage of *copied methods* in the *copy* class and the percentage of *pasted methods* in the *pasted* class. An example is shown (on the method side), in the bottom right panel of figure 6.
- The user *classification*. This is the classification from an other tool we have designed to facilitate the finding of component by indexing techniques.
- The *similar* classes. This list displays all classes similar to the current class, ranked by their degree of similarity. The reader can refer to figure 11 to see an actual display.

The bottom right panel displays the **method information**. With the buttons the user can select:

- The *built with - used to build* information. The two lists are ranked according to a score. This score is represented by two numbers: the percentage of the copy-paste text in the *copied method* and in the *pasted method* (*Csize* and *Psize*). We can see that the current method is *built with* method *acceptPattern:from:* of class *FileBrowser*. This was performed by copying 65% of method *acceptPattern:from:*. The text pasted represents 72% of the current method.
- The *similar* methods. This list displays all methods similar to the current method, ranked by their degree of similarity.
- The *calls - is called by* information. The reader can refer to figure 9 to see an actual display.

The top right panel displays an **history list** of all the moves the user has done so far with the *reuse browser*. This panel is intended to prevent users from getting lost after too many moves (a well known phenomenon in hypertext systems).

5.6.2 Displaying information about possible classes of argument

LEARNIST displays, as [Ponder 92] suggests, the 'possible argument's classes' as a comment assertion in the beginning of every method (see figure 8). LEARNIST displays it in the format:

"@@@ <name of the argument> <list of possible classes of this argument> @@@".

The "@" are needed to distinguish it from the users' comments and to allow our system to locate it. The latter is required for updating these type assertions.

The update is done by removing the old type assertions and inserting the updated ones. This is performed while removing the 'spy instruction' at the end of each 'spy session'. This saves us from recompiling the method twice.

We have also integrated this "type information" with the Smalltalk 'explain' facility. You can highlight a word in the code and ask Smalltalk to explain it. If the user asks Smalltalk to explain an argument variable anywhere in the code, the normal Smalltalk system will answer: "it is an argument variable of this method". LEARNIST enhances this with information about the possible types of this argument variable. Figure 8 illustrates this possibility.

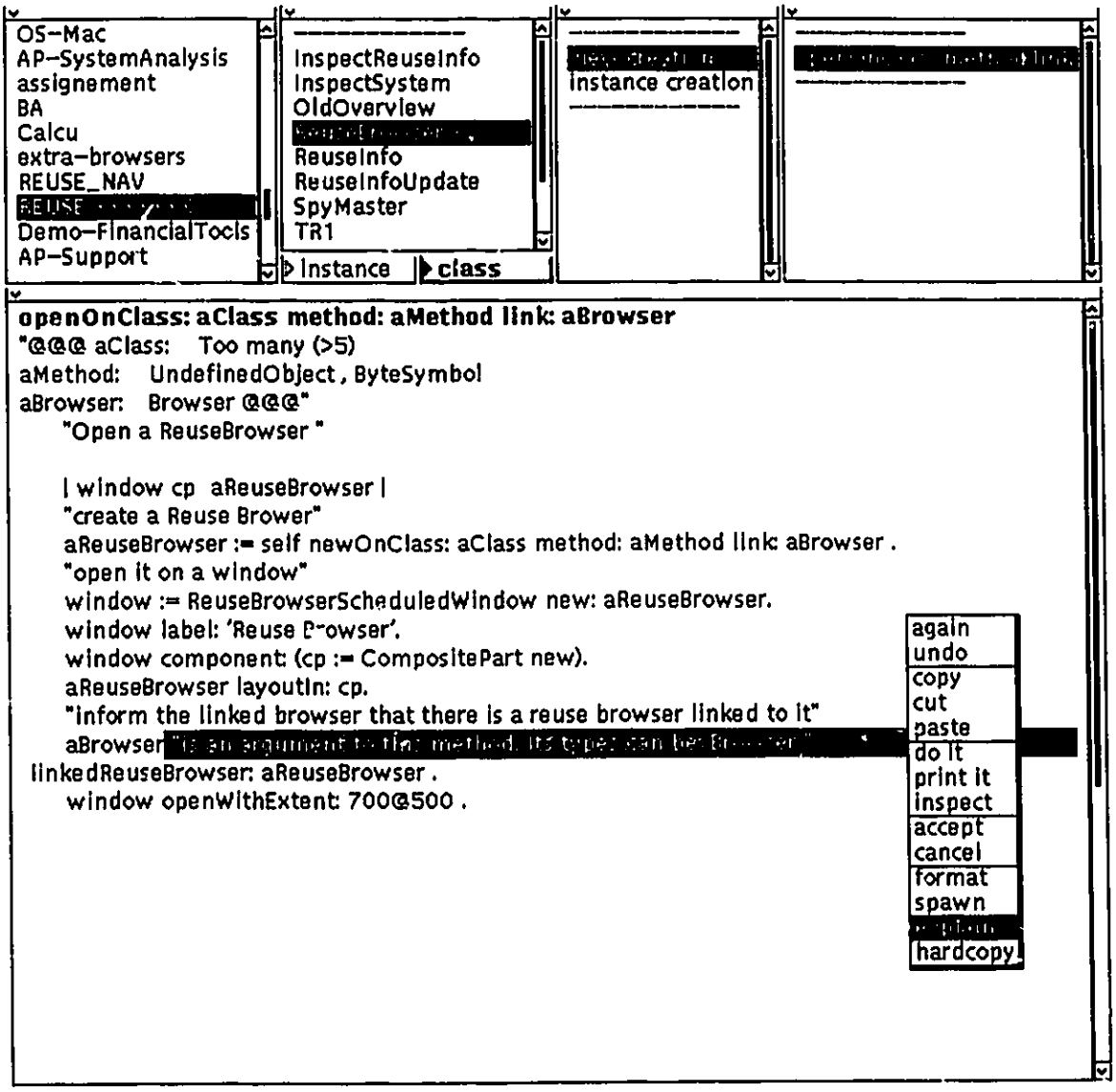


Figure 8: Illustration of the argument class information.

5.7 Integration with the Smalltalk environment

One of the requirements for this system is that it be easy to use. This implies that LEARNIST has to be well integrated in the Smalltalk environment.

This is achieved by three main features:

- The user can access the LEARNIST system very easily. He can open a *reuse browser* from any Smalltalk browser. He can open a *SpyMaster*, like any other Smalltalk tool, through the "launcher".
- The LEARNIST interface is similar to other browsers. We have used the standard panels.
- The *reuse browser* has a simple button interface: he inspects a component's information by clicking on the buttons. Moreover this *reuse browser* is tightly coupled to a normal Smalltalk browser so that the user can go back and forth easily.

5.8 Conclusion

LEARNIST, a specific implementation for the Smalltalk environment has been described. We have proposed techniques to intercept messages, to control the spying process, to store information, and to keep them up to date.

Finally, we have presented the users' interface of LE: the *reuse browser*.

Chapter 6: Illustrations of LEARNIST

This chapter provides illustrations of the two main parts of this system: learning and facilitating the reuse process.

6.1 Illustrative examples of use

In this section we illustrate how LEARNIST can help people to reuse with the two main types of reuse developed in this research: *reuse by use* and *reuse by modification.*

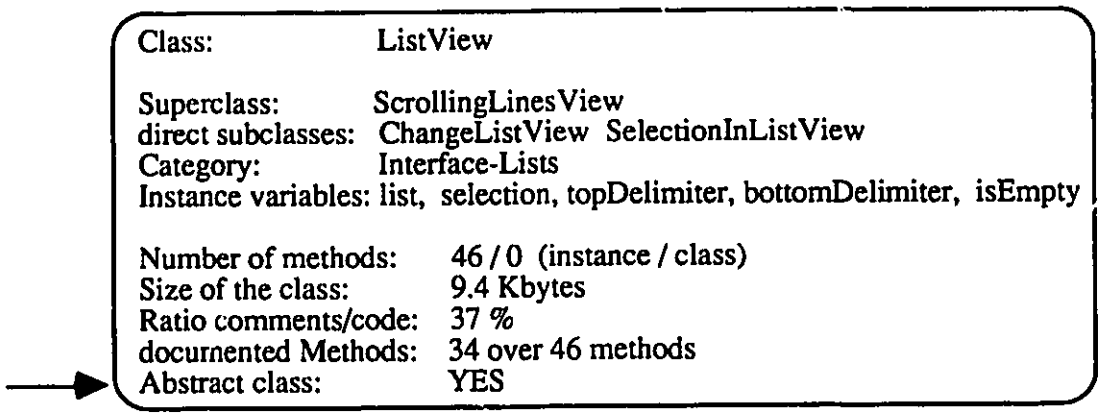
6.1.1 A 'reuse by use' example

In this example, our aim is to construct a program that displays a list of items and lets the user select one of them.

We should mention that this is a very common example: the LEARNIST interface includes three panels of this type (the *history list panel* and the *method* and *class information panels*).

Our strategy is to find a class that implements what we want and then to use it.

In Smalltalk there is a class called 'ListView' which seems to be a good starting point for our search. By looking at the overview of this class, we see that it is an *abstract class*. The next figure represents the *overview panel* of the reuse browser.



This means we cannot directly reuse this class, so we have to use one of its subclasses. By a glance at the same overview, we can see that there are two subclasses: 'SelectionInListView' and 'ChangeListView'. The *general information panel* tells us that 'SelectionInListView' was already reused 6 times and the 'ChangeListView' was never reused. This focuses our attention on the 'SelectionInListView' class (figure 9). By looking at the code of this class, we see that we can create our view with the method called 'on:aspect:change:list:menu:initialSelection:'. The *method information* panel displays the five methods known to call this method (i.e. 5 examples of use of this method). Figure 9 represents the screen dump at this moment; the arrow points out the list of these methods. We will use these methods as examples of how to create this view. If we want to see the 'ReuseBrowser>>classListView' example we just have to click on this line. Figure 10 shows the screen after this "clicking action".

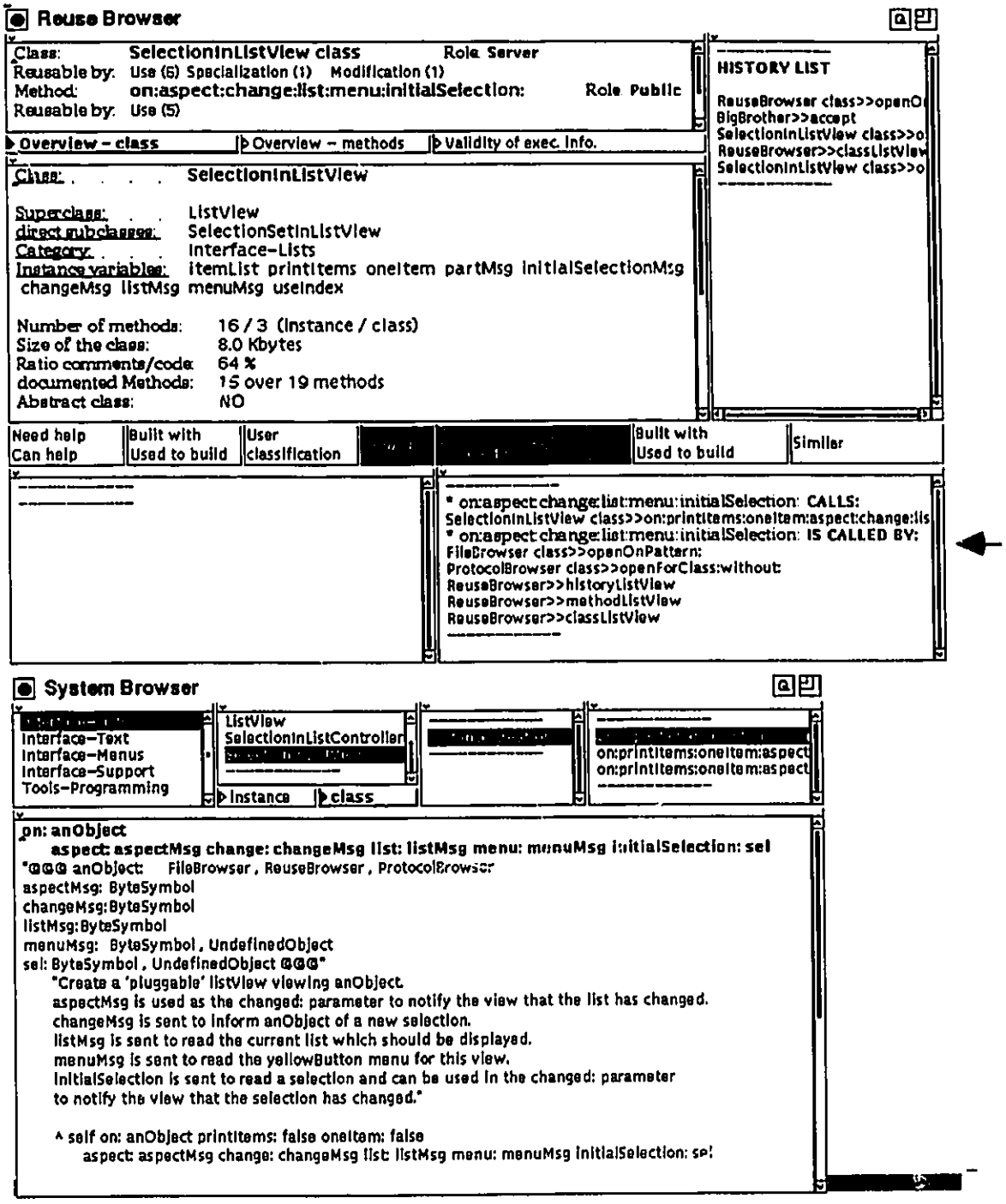


Figure 9: The *reuse browser* and its linked "normal Smalltalk browser"

They display information about the 'SelectionInListView' class.

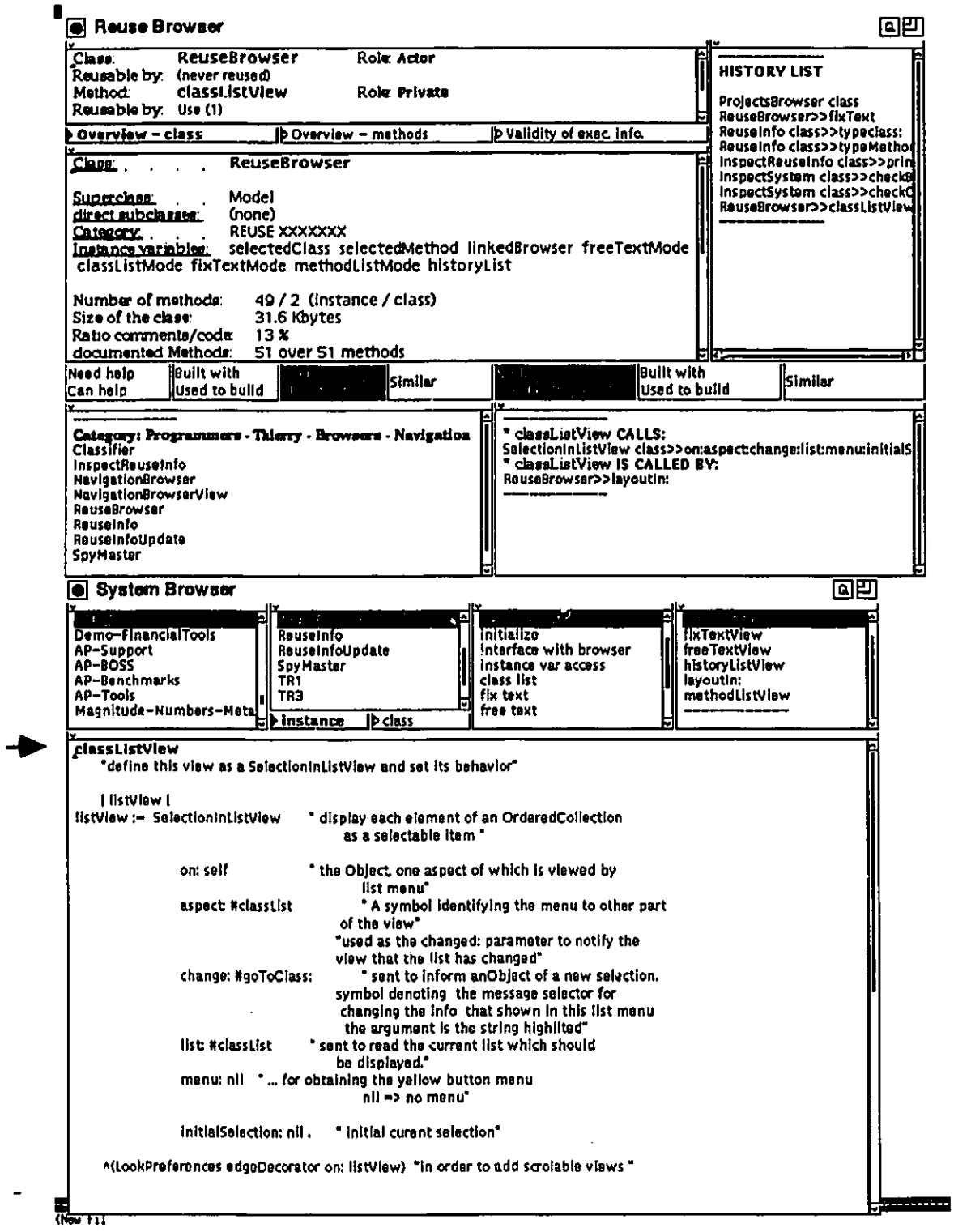


Figure 10: The *reuse browser* and its linked "normal Smalltalk browser" - Illustration of the browsing ability.

With this example (or another one if this one doesn't suit us), we can easily derive by analogy our own call to this creation method just by changing some arguments.

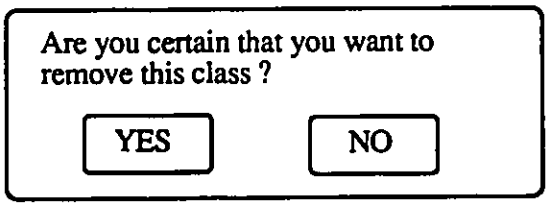
Our method to create this view might be:

```
myListView
 "define this view as a SelectionInListView and set its behavior"

 | listView |
listView :=  SelectionInListView      " display each element of an OrderedCollection
                                        as a selectable item "

   on: self                        " the Object, one aspect of which is viewed by
                                     list menu"
   aspect: #myList                 " A symbol identifying the menu to other part
                                       of the view"
                                    "used as the changed: parameter to notify the
                                     view that the list has changed"
   change: #goToElement:           " sent to inform anObject of a new selection.
                                     symbol denoting  the message selector for
                                     changing the info  that shown in this list menu
                                     the argument is the string highlited"
   list: #myList                   " sent to read the current list which should
                                     be displayed."
   menu: nil                       " ... for obtaining the yellow button menu
                                    nil => no menu"

   initialSelection: nil .         " initial curent selection"

 ^(LookPreferences edgeDecorator on: listView)     "in order to add scrolable views "
```

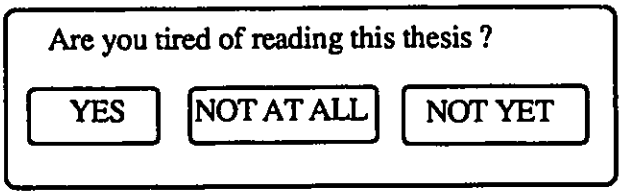
The reader can see that there are only a few differences from original example (bold parts).

6.1.2 A 'reuse by modification' example

In this example we try to adapt a Smalltalk view called 'DialogView'. It displays a question and lets the user answer 'yes or no':



We want to adapt this view to let the user choose between three possibilities:



Our first step is to go into this class (with a Smalltalk browser) and to open a *reuse browser* (see figure 11). Since this class does nearly what we want to do, we look to see if there is a similar class by clicking on the 'similar' button of the *class information* panel. The *reuse browser* displays :

InformView class *(55%)*

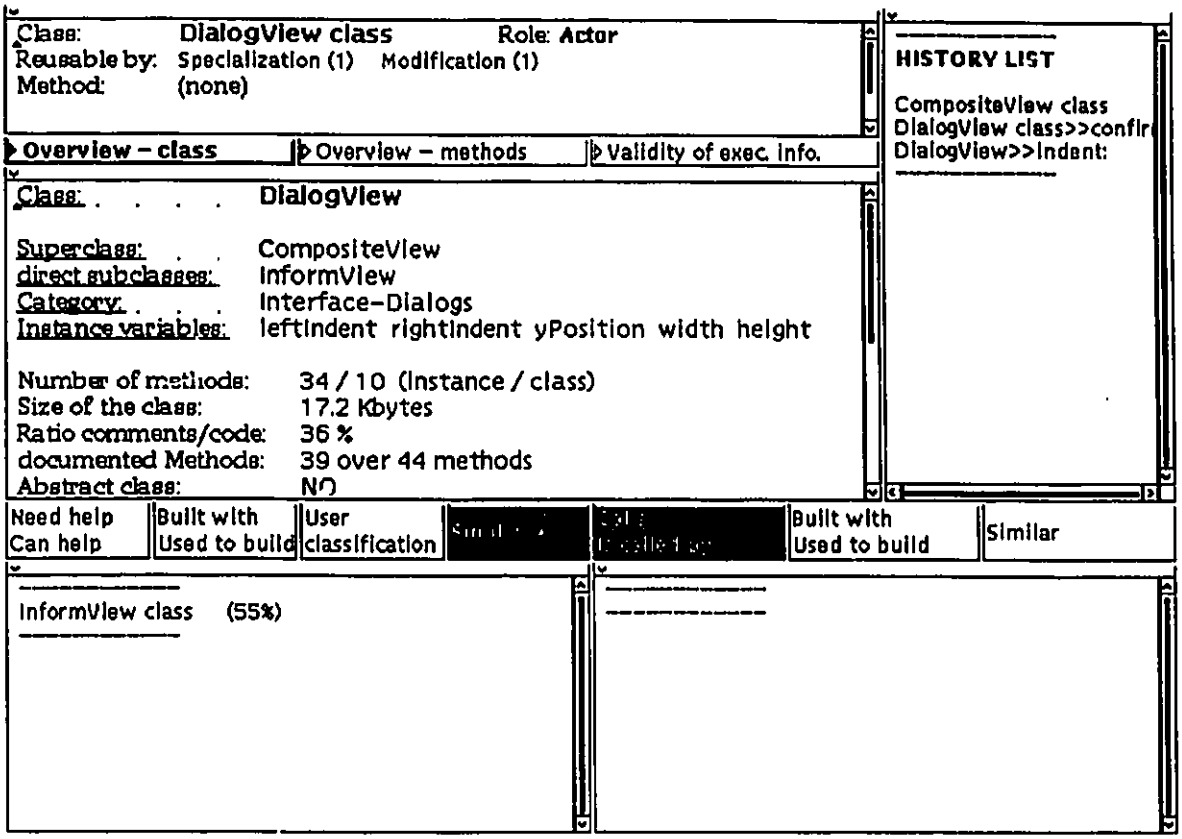
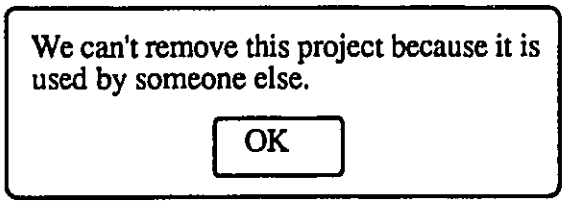


Figure 11: The reuse browser - Illustration of the "similar" information

We inspect this class by clicking on this line, we note in the *overview* panel that this 'inform view' has only one button: 'OK' (see figure 12). Nevertheless this class serves as a very useful example of how to change the number of buttons in the 'DialogView' class.

Actually, this 'Inform view' is one of the previous adaptations of this 'DialogView' class that we have used in previous work. We wanted a view to inform the user of something:



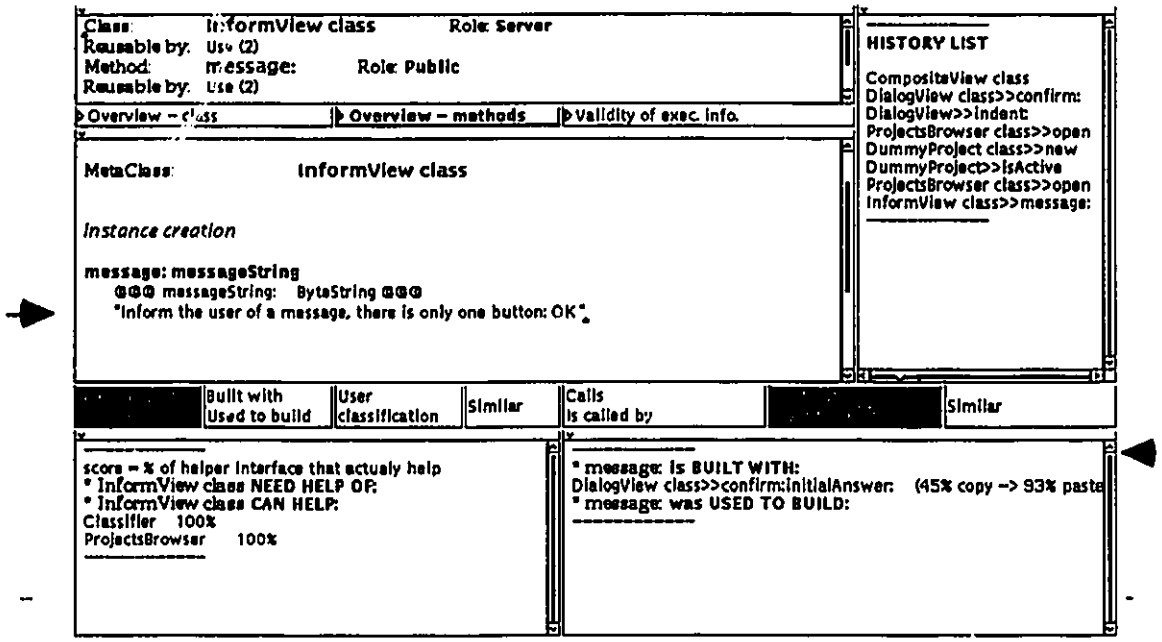


Figure 12: The reuse browser - Illustration of the *Build with - Used to build* information

By clicking on the 'Built with - Used to build' button of the *method information panel* we see that this 'InformView' class was created by modifying only one method of the 'DialogView' class, namely 'confirm:initialAnswer' (see figure 12). Now that we know which method to modify ('DialogView' has 44 methods), and we have an example of how to modify it, it is easy to implement our new requirement by again copying and modifying this 'confirm:initialAnswer' method.

6.1.3 Overcoming the drawbacks of the OO paradigm

We have said (section 4.6.2.3) that LEARNIST helps to overcome certain drawbacks of OO paradigm by providing:

- the types of the arguments.
- the *Calls - Callers* lists.

The first point was illustrated in section 5.6.2.

In order to illustrate the second point we show an example of the inability of the Smalltalk system to give accurate information about the *callers list* of a method and how LEARNIST overcomes this.

Let's take, as an example, the 'accept' method of class 'BigBrother'.

This method is designed to intercept the message sent to compile a method. If we use the standard 'senders' option from any Smalltalk browser, it gives a list of 23 methods that send the message 'accept'. Because of the overloading of that 'accept' message, some of these senders don't call this 'accept' method.

LEARNIST, by recording the actual calls, displays the only caller: method 'accept' of class 'TextController'. This is illustrated in the following figure.

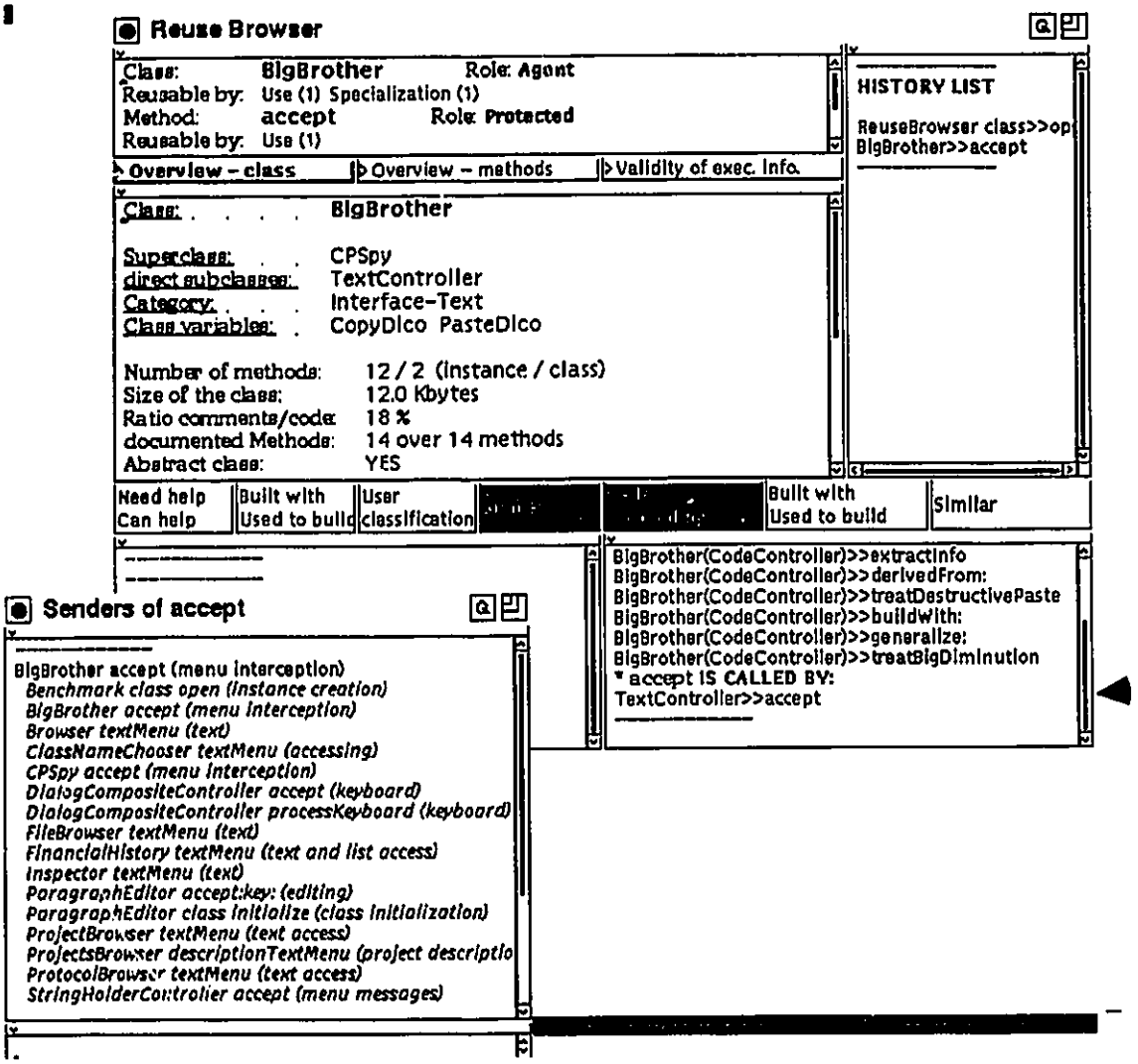


Figure 13: Reuse browser - Illustration of the accuracy of the *caller* list

We have shown that context is of great importance in the OO paradigm. We illustrate, with an example, how LEARNIST can help the user to understand this context.

Let's try to understand the context of the 'ReuseBrowser' class. The context is shown in figure 14 by the *role* of the class and the '*need help - can help*' information. Here, the role of the class is 'Actor' which means this class *collaborates* with other classes. The *need help* information displays these four classes. 'ReuseInfo' and 'ReuseInfoUpdate' classes are part of the LEARNIST system: the reader can check that these classes do collaborate on the block

diagram of LEARNIST (figure 3). The 'TextView' and 'SelectionInListView' classes are the two types of panels used to build this interface. We can use this information to detect the Model-View-Controller framework. We describe this in section 6.2.3.

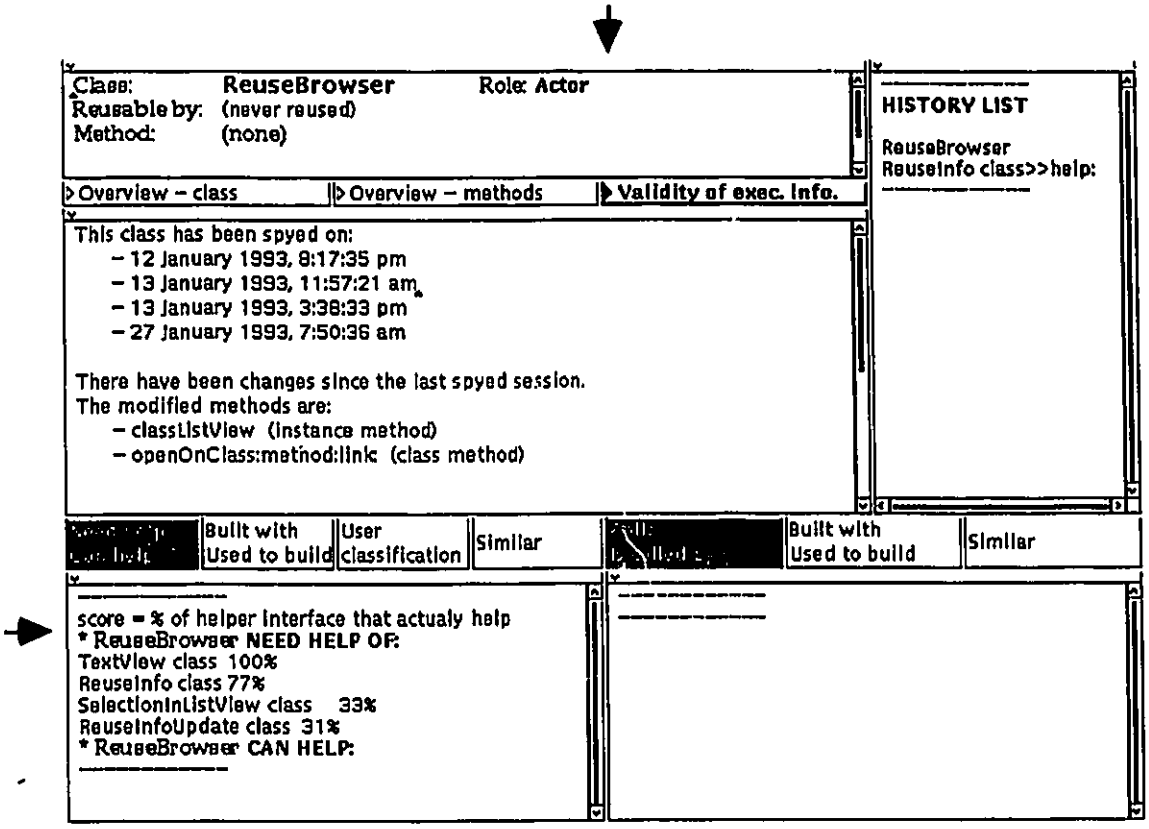


Figure 14: Reuse browser - Illustration of the context information

6.2 Illustration of the learning part of LEARNIST

This section aims at showing, with an example, that our system can spy and learn information. To be as objective as possible, we take an example described in one of the main Smalltalk reference books [Goldberg 84].

6.2.1 Description of the example

The aim of this example is to build a browser for creating and accessing projects (the "Project" feature is already available in the Smalltalk system). We call it a *Project Browser*. The book defines it as follows (p. 322):

> " It should have the following characteristics:
> - The browser should have two subviews, one containing a list menu and the other editable text.
> - The list menu should contain the title of the projects.
> - The text should be a description of the project; the user should be able to edit the text and cancel any changes.
> - Selecting a project title should display the description in the text subview.
> - The yellow button menu[1] in the list menu should support adding new projects, removing existing projects, and entering a project. Adding a new project should prompt the user to specify a project title. Removing a project should require a confirmation".

The next figure represents this project browser.

[1] This means the menu accessed by the mouse when you are in the list menu.

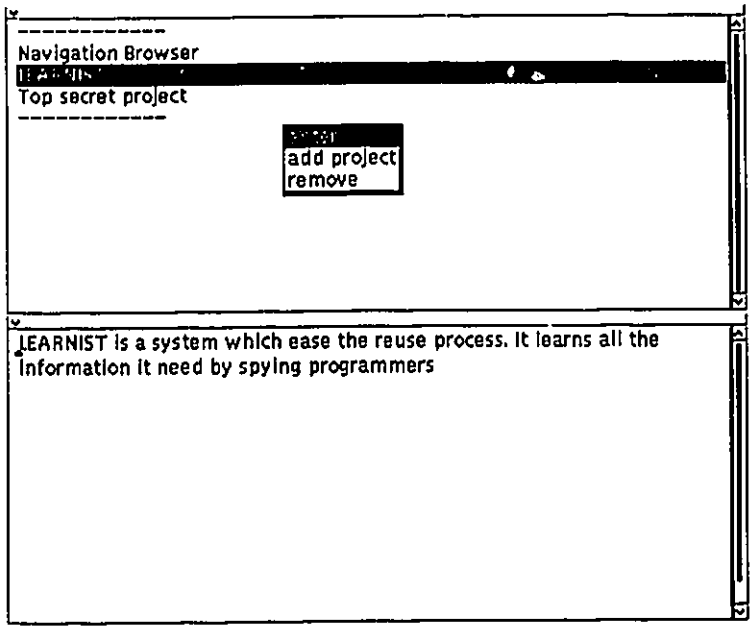


Figure 15: The Project Browser interface

The construction of this project browser is non-trivial and requires a good understanding of Smalltalk. We shall not give a detailed description of the construction process: our aim in this section is to analyze the information captured by LEARNIST during the construction process.

Appendix B shows the code of this class and Appendix C shows all the spied information.

6.2.2 General observations

The first observation is that LEARNIST has learned information.

For 15 out of 15 methods, LEARNIST has stored execution information. More importantly, on 8 of these methods the LEARNIST *caller* list differs from the one given by the standard 'sender' feature of the Smalltalk browser. We discuss this in the next section.

For 8 out of 15 methods LEARNIST has stored copy-paste information. This means that more than half of the methods were created by *reusing by modification* and that LEARNIST has detected it.

6.2.3 Analysis of the 'callers' information

Let's take as specific example, the method 'projectMenu'. With this method, we illustrate consequences of the dynamicity of the OO paradigm.

This method defines the menu shown in previous figure. This method describes what are the options in the menu (i.e. *enter*, *add project* and *remove*) and what to do when the user selects an option (e.g. how to enter in a project).

LEARNIST displays the following *caller* list:

'SelectionInListView>>yellowButtonMenu'

which means that the only caller is method 'yellowButtonMenu' in class 'SelectionInListView'. If we use the standard 'senders' option from any Smalltalk browser, it answers 'nobody'[1].

In fact the issue raised here is applicable to any method dealing with the Model-View-Controller framework. "The Model-View-Controller framework metaphor is a way to design and implement interactive application software that takes advantage of modularity ... to allow pieces already developed for one application to be reused in a new application. The metaphor imposes a separation of behavior between the actual *model* of the application domain, the *views* used for displaying the state of the model, and the editing or *control* of the model and views." [Krasner 88, p. 48]. Interested readers not familiar with this framework can refer to this article. This framework is important since it underpins every application with a user interface. For example in this small application 8 methods out of 15 use this framework. For each of them, the Smalltalk 'sender' facility answers 'nobody'.

In our example, when a user asks for the menu with the mouse, the method 'yellowButtonMenu' of the class 'SelectionInListView' (the *view*) asks its linked *model* (the

---

[1] With the latest version of Smalltalk-80 (version 4.1, released less than 6 months ago), it actually answers something but not the real callers of the methods.

class 'ProjectsBrowser') for the menu options. The point is that this process is dynamic and can't be inferred by static analysis. The Smalltalk system only performs a static analysis and thus can't detect it. LEARNIST, by recording the actual *caller*, can handle this dynamicity.

6.2.4 Example of the copy-paste information

Let's use the same method, 'projectMenu', to illustrate the copy-paste information that LEARNIST is able to learn.

LEARNIST displays the following copy-paste information:

Method 'projectMenu is BUILT WITH:'

'Debugger>>contextMenu (47% copy -> 101% paste)'

Method 'projectMenu was USED TO BUILD:'

'ProjectsBrowser>>descriptionTextMenu (75% copy -> 109% paste)'

This means that method 'contextMenu' was *reused by modification* to build the method "projectMenu'. And 'projectMenu', in turn, was *reused by modification* to build the method 'descriptionTextMenu'.

The fact that the percentage of the 'paste' is greater than 100% means that the text copy-pasted was bigger than the resulting method after modification.

6.3 Experiment on spying on copy-paste

We have observed the copy-paste actions of three programmers (including ourselves) during one week. This was achieved by designing a 'spier' that spies and records all the copy-paste actions. For each of them it records:

- The name of the *copied* method and of the *pasted* method.
- The three characteristic numbers: Csize, Psize and Asize.
- The text *copy-pasted* and the code of the *pasted* method.

This *spier* also records all the methods created, even without any copy-paste.

During this week, 149 methods were created. 52 methods were created or modified using copy-paste.

We have analyzed this copy-paste information by clustering it in 6 categories depending on the type of text copy-pasted. The following table summarizes the results.

	Number of copy-paste	% where the LEARNIST decision was "correct"	% where the *paste* text was modified
variable	10	100%	0%
one line of code	9	89%	0%
one line of comment	5	60%	20%
block	11	100%	100%
a whole method	17	100%	40%

The last two columns are obtained by comparing the *copy-pasted* text and the code of the *pasted* method. This comparison was done manually by myself.

By comparing these two texts, we decided if each copy-paste was worth recording or not (i.e. could it help a future re-user?).

The second column indicates the percentage where LEARNIST and myself agree on this decision.

The third column indicates if the user has modified the *pasted* text.

The second column shows that LEARNIST often makes the right decision.

Let's analyze further the case where LEARNIST made the wrong decision in the 'one line comment' category. These cases happened when the user has copied the "top comment" line. In Smalltalk, the usage is to include a "top comment" line in each method to describe the

function of this method. Thus, the methods were similar (in their functionality) but the text copy-pasted was small compared to the size of the two methods.

The third column shows that, within the interesting copy-paste , the programmer has often modified the *pasted* text. Thus, a static analysis could have problems in detecting these copy-paste . The low percentage (40%) in the 'whole method' category is explained by the fact the user has used copy-paste for duplicating code.

This is too limited an experiment to draw general conclusions on copy-paste usage. We consider this small experiment as an illustration that:

- Copy-paste is often used to create new methods.
- LEARNIST can detect meaningful copy-paste .
- Static detection of this copy-paste is non-trivial because *pasted* text is usually modified.

6.4 Context of use

The LEARNIST system is useful to every person who doesn't know the library well. Realistic libraries are so large that few people know them well. For example, the Smalltalk library contains 500 classes and 6000 methods, and requires about 2 Megabytes of storage. After more than one year of intensive Smalltalk programming, I know about 30% of the library.

The best scenario for LEARNIST is when one person has created a component and another person wants to reuse it. In this case LEARNIST has all the spied information and the second person won't know anything about the component. Since we have adopted a long-term view (the life of the library), this situation occurs often.

Consider the case of a university. Students often use Smalltalk during a course to learn the language and to do little projects. Since they are beginners, they don't know how to use the library. Moreover, because of their limited knowledge, the range of projects they can do is limited. LEARNIST can be of a great help: it can help them to find similar applications, and to understand how these applications work and interact with the rest of the library. LEARNIST can also show them the commonly reused Smalltalk features and examples of previous reuses. With this information students can exploit the real power of the OO paradigm: REUSE. Section 6.1.1 illustrates the scenario of a student constructing software which includes a 'selection in list view'.

In large organizations, there is often a human librarian. LEARNIST can help this librarian to select good reusable components and to get information about them. It can also help him to maintain the library: it gives him an indicator of the frequency of reuse and how the components have been reused.

Finally, LEARNIST can help the librarian to create better reusable components. This is based on the fact that the three forms of reuse do not have the same cost. *Reuse by use* has a low cost since the programmer doesn't have an adaptation phase and doesn't (shouldn't) need to understand the implementation details of components. In *reuse by specialization*, the programmer has an adaptation cost but this is not high since he doesn't modify the code of existing methods but he adds whole new methods. In *reuse by modification*, the programmer works at the source code level, and incurs a high adaptation cost.

LEARNIST can show to the librarian the components reused with a 'high cost' type of reuse (*reuse by modification*) so that he can create new components with a lower reuse cost (*reused by use*). This way of proceeding is validated by [Graver 91]. The author has analyzed the evolution of a library over a long period of time. He says that: "it is not always possible to anticipate and design reusable components based on the context of a single application. Only

after the components have been reused in several different applications can they be deemed truly reusable. This process often results in the creation of reusable frameworks" (p. 52). For example, if the librarian observes that a class is often r*eused by modification*, he could create a *"framework"* class and create subclasses for particular applications.

The example in section 6.1.2 illustrates this kind of scenario. The librarian with the help of LEARNIST sees that the class 'DialogView' has been *reused by modification* twice. Then, he can create an abstract class with subclasses implementing only the number of buttons. In this case, a user who wants a view with four buttons just has to create an other subclass and therefore *reuses by specialization*. The librarian can also create a generic class where the number of buttons is just a parameter. In this case, a user who wants a view with 4 buttons just has to specify this number as a parameter of the call and therefore does *reuse by use*.

LEARNIST can also help developers by giving them statistics about the real use of their software. Since LEARNIST spies on execution, it gathers statistics about how software is used. It can answers questions like: What are the most used features?, How are they used? or What are the unused features?.

Chapter 7: Conclusion

7.1 In summary

We have presented a system that implements the ideas summarized in the thesis' title: 'Facilitating the Reuse Process in an OO Environment by Learning From Observation'.

We have shown that we can observe the user *reusing by use* by spying on *messages*. We intercept these *messages* by automatically inserting a 'spy instruction' which accesses the execution stack to extract relevant information. We don't intercept all messages because this would degrade the execution speed too much. We address this issue by allowing the user to specify on which classes he wants to spy.

We have shown that we can observe users *reusing by modification* by spying on programmers' copy-paste.

We have developed techniques to process the raw spied information. We have shown how to filter "interesting" copy-paste, how to compute scores reflecting the quality of the information and how to generalize the information to the class level.

Our system also updates the stored information. It intercepts all relevant users' actions that indicate that a component, on which it has already collected information, has been modified.

Our system facilitates the three main phases of the reuse process:

- It improves searching by adding search for "similar" components and by facilitating the location of components that have already been reused.

- It improves understanding by displaying an overview, by showing the context of components and by facilitating the understanding of OO concepts.
- Finally, it improves adaptation by displaying previous examples of adaptation which enables users to build their components by analogy with these exampies.

We have illustrated the effectiveness of the two mains parts of our system, learning and facilitating the reuse process.

Finally we have discussed limitations of the present system and ways to overcome them.

7.2 Limitations and extensions

7.2.1 Scalability issues

In this thesis we have tackled the scalability of the system with the number of components of the library. LEARNIST performs only constant time operations; it doesn't even perform operations whose complexity is linear with size of the library.

There is another scalability issue: the amount of information stored. This can be further divided into two sub-issues:

- The operation requested to update the generalization to the class level is linear with the connectivity (copy-paste links and execution links) of this component. This operation needs to be optimized and performed as a background activity.
-The user may be swamped by too much information. Since nearly all stored information has an associated score, we can easily filter the "best" information.

7.2.2 Improving the generalization of the information to the class level

LEARNIST can be improved by adding some specific background knowledge. LEARNIST treats the *class* and the *metaclass* as separate classes. They are in fact linked in the Smalltalk system: for each *class* the user creates, the Smalltalk system automatically creates a corresponding *metaclass*. This knowledge, readily available in Smalltalk, can help LEARNIST treat together the information of the *class* and the *metaclass*.

The system could also use knowledge about the Model-View-Controller framework. With this knowledge, LEARNIST would be able to state clearly if a given class is using this framework and what are the related *model*, *view* and *controller*. This knowledge can be extracted from the Smalltalk system itself.

7.2.3 Combining spying and static analysis

We have discussed the weaknesses of static analysis in the OO paradigm but we do not mean that it is useless. A system can combine both spying and static analysis:

- The static analysis module would inform the user of all the possibilities.
- The spying would inform the user of what was actually done.

Static analysis can improve *copy-paste* spying in two ways:

- By performing some analysis on the copy-pasted code to store more precise information (e.g. is it only comments?)
- By implementing a better way to update information when the user modifies a component. Some code analysis could be performed to analyze the modified parts in order to decide whether or not the stored information about this component still holds.

7.2.4 Improving the spying process

The 'spying execution' module can be improved in two ways.

Firstly, we can avoid the compiling/decompiling phase needed to turn the spy on and off. This can be done by inserting in each method, once for all, an instruction of the form[1]:

if spyThisMethod = true then call method basicSpy.

Then, to spy on a method, we just have to set the variable *spyThisMethod* to true or false. There are two drawbacks to this technique: firstly the spying instruction stays for ever in the method code and secondly this added instruction decreases execution speed.

Execution speed is divided at least by two since for each intercepted message we send an "*ifTrue:*" message. We can't compile this call "in line" since *spyThisMethod* can be true or false.

The best solution is to combine the two techniques: if a class is likely to be spied we insert this spy instruction (with our *SpyMaster*) and turn on and off the spying process with the *spyThisMethod* variable. If we don't want to spy on this class any more, we remove the 'spy instruction' with the same *SpyMaster*.

Secondly, we can automate the choice of which component to spy. To implement this, we need to answer three questions:

- How should the system decide which class to spy upon?
- How long should a class be spied upon?
- How can the user have control over the CPU load of the spying process?

The system can have a priority list for the components that need to be spied. For example this priority list can be:

- First, components that have never been spied upon.

---

[1] The Smalltalk code of this instruction could be: '(spyThisMethod = true) ifTrue: [self basicSpy] .'

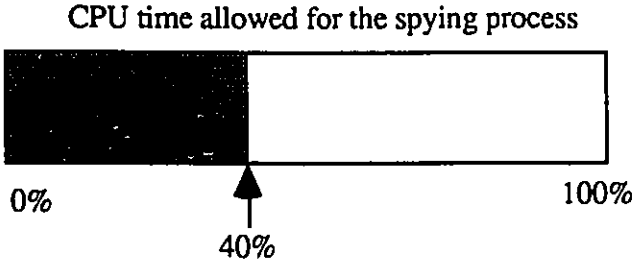
- Then, components that have been changed since last being spied upon (the *suspicious* list).

- Then, components on which the system has little information.

- Finally, components that have been spied upon a long time ago. (LEARNIST already stores the dates when components were spied).

We also have to decide for how long to spy upon the chosen components. The simplest possibility is to spy for a fixed length of time. Unfortunately, some messages are sent very frequently and some very rarely. To understand the consequences of this, let us suppose that we spy on a method for one hour. If it is a frequently called method, we spy on messages sent to it very often and therefore it significantly reduces the speed. If it is a rarely called method, it may be that, during this hour not a single message is sent to it. A good way to decide when to stop the spying process for a given method is: stop when the system has intercepted a given message a specific number of times. This requires the system to have 'spy selection' at the method level (and not, as now, at the class level).

Finally, we have to let the user fix the time he allows for the spying process. It can be controlled by a simple gauge:



The spier will start to spy a new component only if it is compatible with the CPU time allowed for the spying process.

This automatic spying only spies on components on which it needs more information and time needed for spying won't exceed the user specified limit.

7.3 Closing remarks

"The OO paradigm is often touted as promoting reuse" [Johnson 88, p. 22] but few software tools have been developed to promote the reuse process. In this thesis, we have presented a system which facilitates searching for, understanding and adapting components of an OO library. While the utility of this system has been demonstrated, there are still many other ways to promote reuse. It remains to be seen how this fairly recent area of research develops in the future.

References

Basset P. (1990). *Perspectives on Software Reusability.* CASE trends, July/August 1990, pp. 17-18.

Beck, K. (1989). *A Laboratory For Teaching Object-Oriented Thinking.* OOPSLA '89 Proceedings. pp. 1-7

Bigelow, J. and Riley, V. (1987). *Manipulating Source Code In Dynamic Design.* Hypertext '87 proceedings, pp. 397-408.

Biggerstaff, T. and Richer, C. (1987). *Reusability framework assessment, and directions.* IEEE software. 4, 2 (March), pp. 41-49. Also in Biggerstaff, T. and Richer, C. [1989].

Biggerstaff, T. and Richer, C. (1989). *Reusability framework assessment, and directions.* In Frontier Series: Software Reusability: Volume 1-Concepts and Models. Biggerstaff, T. and Perlis A. , Eds. ACM Press, New York, pp. 1-17, Chap. 1. Originally Biggerstaff, T. and Richer, C. [1987].

Bocker, H. and Herczeg, J. (1990). *Track - A Trace Construction Kit.* CHI '90 Proceedings, pp. 415-422.

Boehm, B. and Standish, T. (1983). *Software Technology in the 1990's: Using an Evolutionary paradigm.* IEEE Computer Society, Vol. 16, No. 11, Nov. 1983, pp. 30-37.

Booch, G. (1991). *Object-Oriented design With Applications*. Ed. The Benjamin/Cummings Publishing company, Inc. pp.93.

Cardelli, L. and Wegner, P. (1985). *On Understanding Types, Data Abstraction, and Polymorphism*. December 1985. ACM Computing survey, Vol. 17 (4), pp.481.

Chen, Y. and Ramamoorthy, C. (1986). *The C Information Abstractor*. COMPSAC proceedings.

Constantopoulos, P., Jarke, M., Mylopoulos, J. and Vassiliou, Y. (1992). *The Software Information Base: A Server For Reuse*. (unpublished report).

Curtis, B., Krasner, H. and Iscoe, N. (1988). *A Field Study Of The Software Design Process For Large Systems*. Communication of the ACM 31,11 (November 1988), pp. 1268-1287.

Deutsch, P. (1983). *Efficient Implementation of the Smalltalk-80 System*. Proceeding of the 11th annual ACM Symposium on the principles of Programming languages. pp. 299.

Deutsch, P. (1989). *Design Reuse And The Smalltalk-80 System*. In Frontier Series: Software Reusability: Volume 1-Concepts and Models. Biggerstaff, T. and Perlis A. , Eds. ACM Press, New York, pp. 57-71.

Drummond, C. (1992). *Automatic Goal Extraction from User Action to Accelerate the Browsing of Software Libraries*. M.A.Sc. Thesis, University of Ottawa, December 92.

Fisher, G., Henninger, S. and Redmiles, D. (1991). *Cognitive Tools for Locating and Comprehending Software Object for Reuse*. CHI '91 Proceedings. pp. 318-328.

Fraser, S., Duran, J. and Aubin, R. (1989). *Software Indexing For Reuse*. Proceedings of 1989 IEEE International Conference On systems, Man and Cybernetics, pp. 853-858.

Goldberg, A. (1984). *SmallTalk-80: The Interactive Programming Environment*. Publisher, Addison-Wesley.

Goldberg, A. and Robson, D. (1989). *SmallTalk-80: The Language*. Publisher Addison-Wesley.

Graver, J. (1991). *Evolution of an Object-Oriented Compiler Framework*. Software-Practice and Experience, Vol. 22(7), July 1991, pp. 519-535.

Helm, R. and Maarek, Y. (1991). *Integrating Information Retrieval And Domain Specific Approaches For Browsing And Retrieval in Object-Oriented Class Libraries*. OOPSLA '91 Proceedings, pp. 47-61.

Henry, S., Humphey, M. and Lewis, J. (1990). *Evaluation of The Maintainability of Object-Oriented Software*. Proceedings of the Conference on Computer and Communication System, Vol. 1, Hong Kong, September 1990, pp. 404-409.

Hill, W. (1987). *Machine Learning For Software Reuse*. IJCAI '87 proceedings, pp. 338-344.

Jette, C. and Smith, R. (1989). *Example Of Reusability In An OOP Environment.* In Frontier Series: Software Reusability: Volume 1-Concepts and Models. Biggerstaff, T. and Perlis A. , Eds. ACM Press, New York, pp. 73-101, Chap. 4.

Johnson, R. and Foote, B. (1988). *Designing Reusable Classes.* JOOP June/July 1988, pp. 22-35.

Jones, T. (1984). *Reusability In Programming: A Survey Of The State Of The Art.* IEEE Transactions on Software Engineering, September 1984, pp.488-494.

Krasner, G., Pope, S. (1988). *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* JOOP August/September 1988.

Krueger, C. (1992). *Software Reuse.* ACM Computing Surveys, Vol. 14, No. 2, June 1992.

Lange, B. and Moher, T. (1989). *Some Strategies Of Reuse In An OOP Environment.* CHI '91 proceedings, pp. 69-73.

Lenat, D., Prakash, M. and Shepherd, M. (1986). *CYC: Using Common Sense Knowledge To Overcome Bittleness and Knowledge Acquisition Bottlenecks.* The AI Magazine, Vol. 6, No. 4, pp. 65-85.

Lewis, C. and Olson, G. (1987). *Can The Principle of Cognition Lower The Barriers Of Programming ?.* In Olson, G., Soloway, E. and Sheppard (Ed.), Empirical Studies of Programmers, Vol. 2.

Lewis, J., Henry, S., Kafura, D. and Schulman, R. (1992). *On The Relationship between The Object-Oriented Paradigm and Software Reuse: An Empirical Investigation.* JOOP July/August 1992.

Maarek, Y., Berry, D. and Kaiser, G. (1991). *An Information Retrieval Approach For Automatically Constructing Software Libraries.* IEEE Transaction on Software Engineering, Vol. 17, No 8, August 1991, pp. 800-813.

McIlroy. (1968). *Mass Produced Software Components.* NATO Software Engineering Conference.

Meggendorfer, S. and Manhart. (1991). *A Knowledge And Deduction Based Software Retrieval Tool.* Proceedings of the 6th National Knowledge Base Software Engineering Conference, pp. 126-137.

Meyer, B. (1987). *Reusability: The Case For Object-Oriented Design.* IEEE Software, March 1987, pp. 50-63.

Meyer, B. (1988).*Object Oriented Software Construction.* Prentice Hall.

Micallef, J. (1988). *Encapsulation, Reusability, and Extendibility in OOP Languages.* JOOP 1,1 (April/May 1988), pp.12-36.

Mitchell, T., Mahbadevan, S. and Steinberg, L. (1983). *LEAP: A Learning Apprentice For VLSI Design.* In Machine Learning An Artificial Intelligence Approach. Vol. III. Kodratoff, Y. and Michalski, R. Eds. Morgan Kaufmann Publisher Inc. pp. 271-289.

Nielsen, J. and Richards, J. (1989). *The Experience Of Using Smalltalk.* IEEE Software, May 1989, pp. 73-77.

ParcPlace. (1990). *User's Guide.* Objectworks\Smalltalk. ParcPlace system, Inc.

Ponder, C. and Bush, B. (1992). *Polymorphism Considered Harmful.* SIGPLAN notices, june1992, pp. 76-79.

Prieto Diaz, R. and Freeman, P. (1987). *Classifying Software For Reusability.* IEEE Software, January, pp. 6-16.

Reeves, B. (1990). *Finding and Choosing The Right Object in a Large Hardware Store.* Department of Computer Science, University of Colorado, CO.

Rosson, M., Carroll, J. and Sweeney, C. (1991). *A View Matcher for Reusing Smalltalk classes.* CHI '91 proceedings, pp. 277-283

Teitelman, W. (1974). *The INTERLISP Reference Manual.* Section 20.

Wasserman, A. (1991). *Object-Oriented Software Development: Issue in Reuse.* JOOP, May 1991, pp. 55-57.

Wirfs-Brock, R. and Wilkerson, B. (1989). *Object-Oriented Design: a Responsibility-driven Approach.* OOPSLA '89 Proceedings, pp. 71-76.

Appendix A: Code of method 'basicSpy'

basicSpy

```
        | context  callingContext   orderedCol theClass |
        "BasicSpyActive (global variable) is there to avoid infinite loop
        due to a call to this method when we execute this very method"
        "to change the value do: Smalltalk at: #BasicSpyActive put: false "
BasicSpyActive ifTrue: [ ^nil ].
BasicSpyActive := true .
        context := thisContext sender .
         "context of the target method: (the one which have called this method)"
        "self class= context receiver class ~= context mclass (if go through the super class to find the method)"
        theClass := context mclass .
        callingContext :=  context sender  .
        "context of the method which called the target method"
        "*callingClass := callingContext receiver class.*"

        "ARG TYPE PROCESSING"
        i := 0 .
        orderedCol := OrderedCollection new.
        "extract the type of the args"
        context tempNames do: [ :name |
                "Answer an OrderedCollection of the name strings of the temporary
                variables for this context and all its enclosing scopes. Note that this
                does not include compiler-generated temporary variables."
                "This OrderedCollection is ordered with args in the order of apparition
                in the selector and then the temp var"

                i := i + 1 .            "i is the index of this var."
                 (context localScope variableAt: name ) class = ArgumentVariable
                "check if it is argument variable"
```

```
        ifTrue: ["store the info in orderedCol for further processing"
                        orderedCol addLast:
                                (Array with: name with: ( Bag with: (context  tempAt: i ) class)) .
                        ] .
                                        ] .

"store the types in methodInfoDict"
orderedCol isEmpty ifFalse: [
        (theClass argsDictAt: context selector) isEmpty ifTrue:
                [ (theClass argsDictAt: context selector ) become: orderedCol ]
        ifFalse:
                [ i := 0 . " i is an index for the arguments"
                (theClass argsDictAt: context selector ) do: [ :elm |
                i := i + 1 .
                elm at: 1 put: ((orderedCol at: i) at:1) .                      "name <ByteString>"
                (elm at: 2) addAll: ((orderedCol at: i) at:2 ) . ] ]      "types <'real' classes>"
                                                                ] .

"CALLING METHOD PROCESSING"
"SpyTranscript cr ; show: callingContext mclass printString .
SpyTranscript  show: '>>' , callingContext selector printString ."
(theClass callersDictAt: context selector)
        add: (Array with: callingContext mclass          "calling class <'real' class>"
                                with: callingContext selector]   "calling method <ByteSymbol>"
((callingContext mclass) callsDictAt: ( callingContext selector))
        add: (Array with: theClass
                                "called class (in which the method is implemented) <'real' class>"
                                with: context selector      "called method <ByteSymbol>"
                                with: context receiver class ). "receiver class <'real' class>"

"ABSTRACT CLASS ?"
(theClass abstractClassInfo = nil or: [ theClass abstractClassInfo = true ])
        ifTrue: [ (theClass someInstance = nil)
                ifTrue: [ theClass abstractClassInfoPut: true ]
                ifFalse: [ theClass abstractClassInfoPut: false ].
```

```
                                ].

        "CALLING CHAIN PROCESSING
        callingChain := context  stack .
        ""the calling chain start with the method spied a then tack back
        all non-returned calls.
        callingChain is an OrderedCollection of MethodContext""
        SpyTranscript cr ; show: 'Calling chain :' ;cr .
        callingChain do: [ :cxt |
        SpyTranscript  show: cxt printString ;cr ] ."
        "CLASS AND INSTANCE VARIABLE PROCESSING
        theClass isMeta ifFalse: [  ""it is a non-MetaClass: take care about instance var""
         theClass instVarNames do:                    ""it is an array of instance var names""
                [ :name | ""name are ByteString""
                SpyTranscript cr ; show: name , ':* ',
                                (context receiver instVarAt:( theClass allInstVarNames indexOf: name))
class printString ] ]
        ifTrue: [ ""It is a Meta_class:        display class var.
                        we do it without the help of the context because instance var of the metaclass
                        are not the class variables but more complex stuff""
                        theClass classPool associationsDo: [ :ass |
                                ""it is a Dictionary. key: <ByteSymbol> ,name of the var
                                                                value: <'real' class> , value of
this var""
                                        SpyTranscript cr ; show: ass key asString , ': ' ,
                                                        ass value class printString ]
                        ] ."

BasicSpyActive := false .
```

Appendix B: Code of class 'ProjectsBrowser'

```
Object subclass: #ProjectsBrowser
    instanceVariableNames: 'listOfProjects selectedProject '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'extra-browsers'



ProjectsBrowser methodsFor: project description

changeDescription: aText
"@@@ aText:    Text @@@"
    "change the description text of the selected project"

    selectedProject description: aText.
    ^true

descriptionText
    "display the text in the description text part
    The text is the description of the project selected"

    selectedProject isNil
        ifTrue: [^'']
        ifFalse:[^(selectedProject description)].

descriptionTextMenu
    ^PopUpMenu
                        labels: 'accept' withCRs
                            " aSting of the menu item, each item separatee by a Ck
```

```
                    The withCRs message subsitute a CR in place of \"
                lines: #(1 )
                values: #(#accept)
                        " array of message selector, the message will be send to the
                        object being viewed but exception (edit ...)"

ProjectsBrowser methodsFor: project list

project
    "Answer the currently viewed project."

    ^ selectedProject

project: aProject
"@@@ aProject:      DummyProject , UndefinedObject @@@"
    "Set aProject to be the currently viewed project. "

    selectedProject := aProject.
    self changed: #descriptionText .

projectList
    "answer the list of all the project"
    ^listOfProjects

projectMenu
    "Answer a Menu of operations on the projects "

    selectedProject isNil
            ifTrue: [^nil]

            ifFalse: [^PopUpMenu
                        labels: 'enter\add project\remove' withCRs
                            " aSting of the menu item, each item separatee by a CR
                            The withCRs message subsitute a CR in place of \"
                        lines: #(1 )
```

```
                values: #(#enterProject #addProject # removeProject )
                        " array of message selector, the message will be send to the
                        object being viewed but exception (edit ...)"
            ].

ProjectsBrowser methodsFor: view-layout

descriptionTextView
    "define this view as a TextView and set its behavior"

    ^(LookPreferences edgeDecorator on:   "in order to add scrolable views"
            ( TextView
                on: self        "where to send the message"
                aspect: #descriptionText            "for the change meca and for geting the new text"
                change: #changeDescription:
                menu: #descriptionTextMenu  ))        " ... for obtaining the yellow button menu
                                                    nil => no menu"

layoutIn: aCompositePart
"@@@ aCompositePart:  CompositePart @@@"
    "build the global view"

"DESCRIPTION VIEW"
    aCompositePart add: self descriptionTextView
        borderedIn: (LayoutFrame new
                leftFraction: 0;
                topFraction: 0.5 ;
                rightFraction:1 ;
                bottomFraction: 1).
    aCompositePart add: self projectListView
        borderedIn: (LayoutFrame new
                leftFraction: 0;
                topFraction: 0;
                rightFraction: 1;
                bottomFraction: 0.5).
```

```
projectListView
    "define this view as a SelectionInListView and set its behavior"
    ^(LookPreferences edgeDecorator on:
        (SelectionInListView on: self
            printItems: true  "if printItems is true, then the
                view will show the printStrings of the items in the list, rather than
                assuming they are already text-like objects."
            oneItem: false    "if oneItem is true
                the list works as a read-only list of one item. this is mainly used
                for the root list of various sub-browsers spawned from the browser."
            aspect: #project
            change: #project:
            list: #projectList
            menu: #projectMenu
            initialSelection: #project))

ProjectsBrowser methodsFor: initialize

initialize
    "initialize the two instance variables"

    listOfProjects := DummyProject allInstances.
        "gather all the existing projects.
        DummyProject is a class that simulate a project"
    listOfProjects isEmpty ifTrue: [listOfProjects :=
            OrderedCollection with: ( DummyProject new title: 'LEARNIST')].
    selectedProject := nil.
        "there is no selected project at the begining"

ProjectsBrowser methodsFor: manipulation of projects

addProject
    "add another project"
```

```
    |aProject aString|

    "Prompt the user to enter the title of the project"
    aString := DialogView
                    request: 'title for the project ?'
                    initialAnswer: 'a project'.

    "create a project with this tiltle"
    aProject := DummyProject new title: aString.
    "add it in the project list"
    listOfProjects add: aProject.
    "make this project the selected one"
    selectedProject := aProject.
    "inform the views to update themselves"
    self changed: #project.
    self changed: #descriptionText.

enterProject
    "enter in the selected project"
    selectedProject notNil ifTrue: [selectedProject enter]

removeProject
    "remove the currently selected project"

    (DialogView confirm: 'this action will remove the selected project
do you realy want to remove it ?')
        ifTrue: [
                "check if the project is active"
                selectedProject isActive  ifTrue: [
                    InformView message: 'The project is active. We can''t remove it'.
                    ^self                                ]
                                        ifFalse: [
                    "remove the project"
                    selectedProject release.
                    "remove it from the project list"
```

```
                listOfProjects remove: selectedProject.
                "deselect the project"
                selectedProject := nil.
                "inform the views to update themselves"
                self changed: #project.
                self changed: #descriptionText
                                                ].
        ].


ProjectsBrowser class
    instanceVariableNames: ''



ProjectsBrowser class methodsFor: view creation

open
    "Open a ProjectsBrowser"

    "ProjectsBrowser open"

    | aWindow cp  aProjectsBrowser |
    aProjectsBrowser := self new initialize.
    aWindow := ScheduledWindow new.
    aWindow label: 'Projects Browser'.
    aWindow component: (cp := CompositePart new).
    aProjectsBrowser layoutIn: cp.
    aWindow openWithExtent: 475@400
```

Appendix C: Spied information about class 'ProjectsBrowser'

Class: **ProjectsBrowser**

projectMenu

ROLE: Public

CALLERS:

'SelectionInListView>>yellowButtonMenu'

COPY - PASTE Information

Method 'projectMenu is BUILT WITH:'

'Debugger>>contextMenu (47% copy -> 101% paste)'

Method 'projectMenu was USED TO BUILD:'

'ProjectsBrowser>>descriptionTextMenu (75% copy -> 109% paste)'

changeDescription: aText

@@@ aText: Text @@@

ROLE: Public

CALLERS:

'TextView>>accept:from:'

descriptionText

ROLE: Public

CALLERS:

'TextView>>getContents'

descriptionTextMenu

ROLE: Public

CALLERS:

'TextView>>yellowButtonMenu'

COPY - PASTE Information
method 'descriptionTextMenu is BUILT WITH:'
'ProjectsBrowser>>projectMenu (75% copy -> 109% paste)

project

ROLE: Public
CALLERS:
'SelectionInListView>>initialSelection'
COPY - PASTE Information
Method 'project is BUILT WITH:'
'Debugger>>context (100% copy -> 89% paste)'

project: aProject

@@@ aProject: DummyProject , UndefinedObject @@@
ROLE: Public
CALLERS:
'SelectionInListView>>changeModelSelection:'

projectList

ROLE: Public
CALLERS:
'SelectionInListView>>getList'

descriptionTextView

ROLE: Private
CALLERS:
'ProjectsBrowser>>layoutIn:'
COPY - PASTE Information
Method 'descriptionTextView is BUILT WITH:'
'ReuseBrowser>>fixTextView (100% copy -> 93% paste)'

layoutIn: aCompositePart

@@@ aCompositePart: CompositePart @@@
ROLE: Public

CALLERS:

'ProjectsBrowser class>>open'

projectListView

ROLE: Private

CALLERS:

'ProjectsBrowser>>layoutIn:'

COPY - PASTE Information

Method 'projectListView is BUILT WITH:'

'Debugger class>>openFullViewOn:label: (66% copy -> 227% paste)'

'SelectionInListView class>>

on:printItems:oneItem:aspect:change:list:menu:initialSelection:

(32% copy -> 48% paste)'

initialize

ROLE: Public

CALLERS:

'ProjectsBrowser class>>open'

addProject

ROLE: Public

CALLERS:

'SelectionInListController>>yellowButtonActivity'

COPY - PASTE Information

Method 'addProject is BUILT WITH:'

'Browser>>prompt:initially: (71% copy -> 62% paste)'

enterProject

ROLE: Public

CALLERS:

'SelectionInListController>>yellowButtonActivity'

removeProject

ROLE: Public

CALLERS:

'SelectionInListController>>yellowButtonActivity'

COPY - PASTE Information

Method 'removeProject is BUILT WITH:'

'ReuseBrowser>>removeInfoMethod (66% copy -> 30% paste)'

MetaClass: ProjectsBrowser class

open

ROLE: Public

CALLERS:

'ProjectsBrowser class>>DoIt'

'UndefinedObject>>DoIt'

COPY - PASTE Information

Method 'open is BUILT WITH:'

'ClassReporter class>>open (100% copy -> 75% paste)'