Making navigation easier in object-oriented programming systems

Thesis

Making Navigation Easier In Object-Oriented Programming Systems

Yibing Li (B.Eng.)

Thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in Human-Computer Interaction

The Open University
Milton Keynes
U.K.

May 1992.

Author's number : M7039433
Date of submission : 11th May 1992
Date of award : 19th June 1992

Abstract

It has been reported that non-expert users have difficulties in finding reusable software components in large object-oriented programming systems and there is a need for help tools. The research reported in this thesis addresses this issue. Described in this thesis is the design of a tool called BRRR, which aims to help non-expert users overcome such difficulties. It is developed for Smalltalk-80, the target system of this research.

BRRR is a query tool with a browsing capacity. It allows users to find necessary components by query. Its design is based on the 'retrieval by reformulation' paradigm (Williams, 1984) which was originally used in the domain of information retrieval. This paradigm allows users to incrementally specify a query by reformulation. When users specify an initial query, BRRR presents the users with an example component which satisfies the query. The users can then construct further queries by using the information presented by the system. In this way, users who are not familiar with the system or who do not know exactly what they want can be guided towards the appropriate information.

During this research, two versions of BRRR were developed: BRRR1 and BRRR2. BRRR1 was developed initially, based on the 'retrieval by reformulation' principle. After its implementation, a formative, empirical evaluation was conducted on it with a group of users. Based on the findings of the evaluation, BRRR2, an improved version of BRRR1 was developed. BRRR2 incorporates enhanced classification methods and explanation facilities. This new version of the tool was then evaluated empirically with a group of ten users.

The empirical evaluation of BRRR2 showed encouraging results. It demonstrates that the 'retrieval by reformulation' approach used in this research could be used successfully in helping users find reusable software components in object-oriented programming systems.

Acknowledgements

First of all, I would like to thank my internal supervisors Tim O'Shea and Pat Fung for helpful advice and continuous encouragement throughout this work. Without their help, this thesis would not have been possible.

Special thanks go to Tim O'Shea for introducing me into the area of object-oriented programming.

Thanks to my external supervisor Steve Cook for the encouragement and technical advice he has given me. He has always been patient in discussions with me during the course of this research.

Thanks to Pat Fung and Ann Jones, for helping me overcome the difficulties of not being a native English speaker.

I am indebted to the following people who have provided input to my research or read my thesis drafts: Alan Borning, Mike Brayshaw, Mark Elsom-Cook, Benedict Heal, Jeoff Mohamed, Claire O'Malley, Marian Petre, Royston Sellman, Ronnie Singer, Randall Smith and Josie Taylor.

Thanks to Eliot Miranda at Queen Mary and Westfield College for giving me access to his M.Sc students for my empirical study. I wish to thank the students for participating in the study.

Thanks also to all members of CITE (past and present) for providing an exciting working environment.

I thank my wife Meng Hua for her love and support which helped me overcome many difficulties I encountered during my studies.

Finally, I wish to thank my parents for their love and encouragement.

This work was supported by a studentship from the Open University.

Table of Contents

List of figures and tables

Chapter 1 Introduction

In large object-oriented programming (OOP) systems, non-expert users have difficulties in finding required components in their programming (Nielsen et al., 1989, O'Shea, in press). The research described in this thesis addresses the issue of how to help non-expert users find reusable components in object-oriented programming systems. Literature related to helping users with this problem of component reuse (Gibbs et al., 1990, Ramamoorthy et al., 1988, Fischer, 1987) has shown that there is a need for tools to provide help in this task. This thesis describes the development of such a help tool. This tool is based on the 'retrieval by reformulation' paradigm (Williams, 1984) which was originally used in the domain of information retrieval. An object-oriented programming system, Smalltalk-80 (Goldberg et al., 1983, Goldberg, 1983) is used as a vehicle to explore design ideas for this tool and to test the implementations built upon those ideas. The term 'non-expert users' in this thesis refers to people whose Smalltalk programming experience is approximately that of a person who has completed an introductory Smalltalk course (usually consisting of between three to ten days tutorials plus hands-on experience) and has a certain familiarity with the Smalltalk interface. In discussions throughout this thesis, unless specified otherwise, the terms 'user' or 'users' refers to such 'non-expert' users.

The chapter is organized as follows: we first briefly introduce several basic concepts of object-oriented programming. After that, we present a typical problem that users might have when programming in object-oriented programming systems. This is followed by an analysis of the causes of that problem and a solution which we put forward to address that problem. Finally, we outline the methodology used in this research and present the overall structure of the research undertaken in the course of developing this solution.

1.1 An Introduction to object-oriented programming

In this section, we briefly introduce the basic elements of object-oriented programming. This is to provide a preliminary background for the work described in this thesis. It draws on the descriptions of object-oriented programming given by Blair et al. (1991); Lalonde et al. (1990) and Collins (1990). More complete and technical descriptions can be found in Blair et al. and Budd (1991). We first describe the basic concepts of object-oriented programming and then list a number of frequently used OOP systems.

1.1.1 Object-oriented programming

Object-oriented programming is a new style of programming. In the purest sense, it is defined as programming implemented by sending messages to objects (Pinson et al., 1988). More specifically, object-oriented systems contain three elements: **objects, class** and **inheritance.** In this section, we describe these three mechanisms and two supporting mechanisms: **polymorphism** and **dynamic binding.**

i) Objects

An object is an essential concept of object-oriented programming. It is defined as follows:

> *An object is an encapsulation of a set of operations or methods which can be invoked externally and of a state which remembers the effect of the methods* (Blair et al., 1991, p.26).

The methods are the set of operations which we are allowed to perform within the context of the object. They are the only procedures by which that object can be accessed, and are also referred to as the object's **external interface**. The external interface will be made up of exactly the information that is required to operate on the object but nothing more. The **state** gives the the status of the object at any particular time. This could be defined by the contents and values of the data structure of an object.

The operational interface to an object is restricted to only what is required by the user, with the implementation of the methods externally invisible. In addition, the operational interface provides the 'user view' of the behaviour of an object, i.e. it is known that an object provides certain functionality but beyond that no further details are known. This is important in handling complexity in a problem as once an object is implemented, it is no longer important to know the internal details of the algorithms and data structures. It is only necessary to know the interface it presents.

Another aspect of the object mechanism is that in solving a problem, programmers are expected to decompose the problem in terms of objects rather than functions such as they do in traditional 'structured' approaches. The real world entities can often be directly mapped into objects in programming systems. The proponents of the object-oriented programming approach thus believe that this kind of design for an application tends to be easier to understand, thus easier to implement and maintain (Collins, 1990).

ii) Objects communicate via message-passing

In object-oriented systems, objects interact with each other to complete a computation. The communication between objects is achieved by objects sending messages to each other. When an object receives a message, it invokes an appropriate method in that object, which then returns an object as a result. This is similar to a procedure call in traditional languages.

iii) Classes

The motivation in supporting classes is to provide a rudimentary form of classification. Some objects share common characteristics and thus can be described by the same general description. A **class** is a description of a set of objects with similar characteristics, attributes, and behaviours. In OOP systems, each individual object is a member or an **instance** of a particular class. All instances of a class share common characteristics. The operation interfaces of all instances of a class are identical. However, each instance has its own state which may be different from other instances.

iv) Inheritance

Objects may often be thought as specializations of other objects. For example, precious metals are specializations of metals, sport cars are specializations of cars. Extending this notion, we can view one class of objects as a subclass of another. A new class can thus be defined in terms of an existing class but with modifications or extensions to meet the requirements of the new class. The new class therefore shares (or **inherits**) the behaviour of the old class but has modified or additional behaviour. A class which inherits from another class inherits all the methods and attributes of that class. It can also add new methods and attributes at will. The new class is said to be a **subclass** of the old class, and the old class is the **superclass** of the new class. Because the new class has only to describe how it is different from the superclass, it has the following advantages:

logically, a brevity of expression is achieved;

physically, this permits a sharing of operation — an operation provided in one class is also applicable to every subclass, this facilitates code reuse.

Some OOP systems support **multiple inheritance,** i.e. a class can inherit behaviour and attributes from more than one class.

Class hierarchy

As an OOP system develops, subclasses are constructed out of existing classes until the appropriate functionality is developed. As a result, a class hierarchy is formed. The hierarchy is normally rooted by a special class, often referred to as *Object,* which contains a minimal set of behaviour common to all classes.

v) Polymorphism

Polymorphism can be defined as the ability of behaviour to have an interpretation over more than one class. For example, a message 'print' can be sent to many objects and different objects would interpret the message in their own way. An object representing a text document receiving the message would print itself in a way which is different from that of an object representing a graphical document. With polymorphism, the same name can be used throughout a system to denote a commonly used and well-understood operation. This consistency in operation naming across class boundaries helps significantly reduce the name space in large systems.

d) Dynamic binding

Dynamic binding is a technique by which the mapping of method name to implementation is carried out on every method invocation (as opposed to the static binding where the mapping is carried out at compile time). Each time a method is invoked, the class hierarchy will be searched to find the appropriate implementation. This technique supports the evolution of a program since changes to the program can be made without requiring a re-compilation of the whole program.

1.1.2 Several object-oriented programming systems

Of all the OOP systems, Smalltalk is the most consistent with definitions and properties of the object-oriented paradigm. It was originally developed at Xerox, Palo Alto Research Center in the early seventies and is now owned and marketed by a company named ParcPlace Systems. There have been five versions of Smalltalk: Smalltalk-72; Smalltalk-74; Smalltalk-76; Smalltalk-78; Smalltalk-80*. There is another dialect of of Smalltalk-80: Smalltalk/V which is marketed by the company Digitalk, for Macintosh and IBM PC. Apart from Smalltalk, other OOP systems include:

C++ (Stroustrup, 1986) and Objective-C (Cox, 1986) which are object-oriented extensions of the C language;

Eiffel (Meyer, 1988) which is a strongly typed object-oriented language;

Flavor (Moon, 1986), LOOPS (Bobrow et al., 1986) and CLOS (Keene, 1989) which are object-oriented extensions to Lisp language;

Object Pascal (Tesler, 1985) which is an extension to the Pascal language.

In this section, we have introduced several basic concepts of object-oriented programming and listed a number of frequently used object-oriented systems. In the next section, we will describe the problem this research addresses.

1.2 The problem

The object-oriented programming paradigm promotes the 'programming by reuse' approach (Meyer, 1987, Fischer, 1987, Rubin, 1990). In this thesis, only

* The Smalltalk-80 system is now marketed by ParcPlace Systems under the name Objectworks for Smalltalk-80.

the code reuse concerns us, little attention has been placed on 'design reuse'. With this approach, programmers do not always need to code from the beginning, they use existing components as the basis of new programs. In Smalltalk-80, for example, there is a large library of components, i.e. classes and methods associated with these classes, and programming is mainly undertaken by reuse of the existing classes and associated methods. The advantages of reusing software are evident: it would save the effort made in repeatedly developing certain software components, hence this approach has the potential of increasing code productivity; it is also relatively 'safer' to reuse software components which have been used and tested in previous applications. However, reuse is not without cost. Biggerstaff et al. (1989) suggest that, in order to operate successfully, a reusability system must address the following four fundamental problems:

— Finding components suitable for reuse;

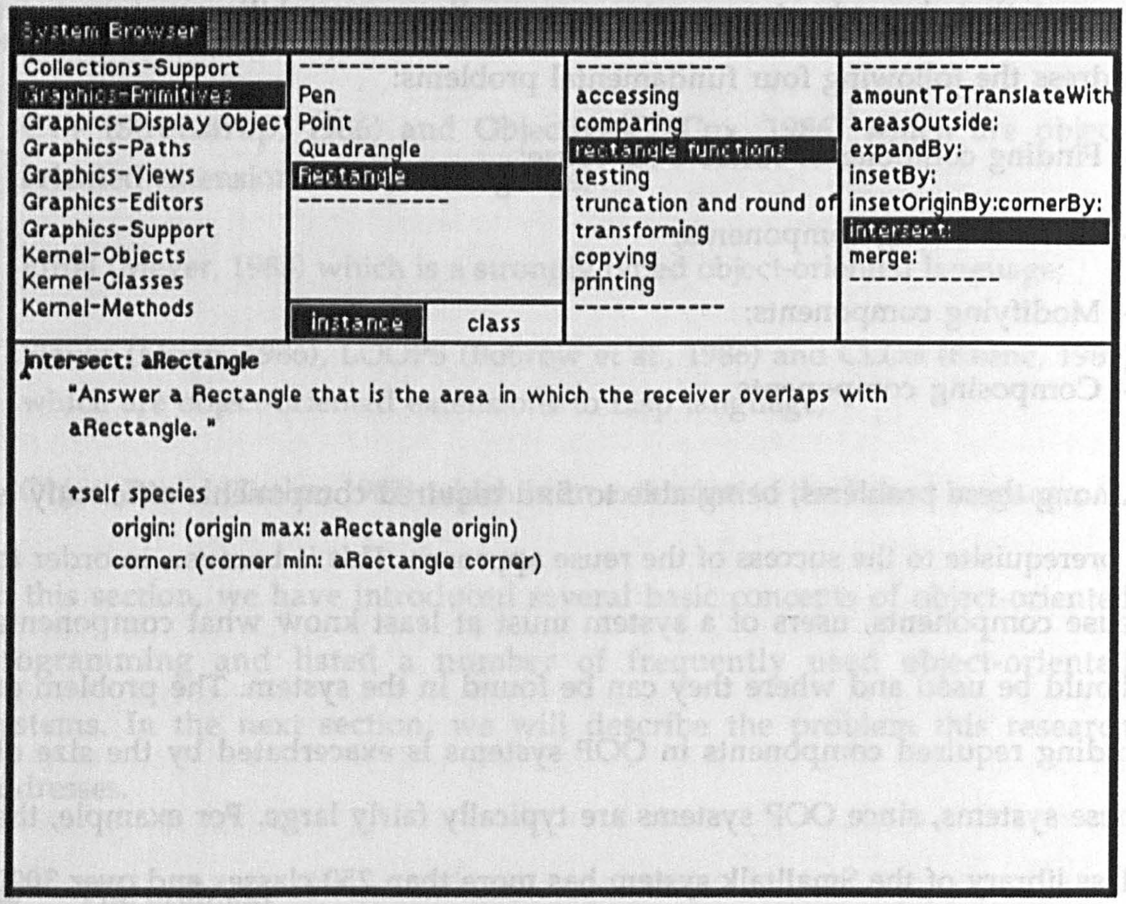
— Understanding components;

— Modifying components;

— Composing components.

Among these problems, being able to find required components efficiently is a prerequisite to the success of the reuse approach. This is because, in order to reuse components, users of a system must at least know what components should be used and where they can be found in the system. The problem of finding required components in OOP systems is exacerbated by the size of these systems, since OOP systems are typically fairly large. For example, the class library of the Smalltalk system has more than 250 classes and over 3000 methods. In such large systems, it is not always an easy task for non-expert users to find required classes or methods. It is necessary for systems to provide users with supporting facilities to help them look for necessary information. Otherwise, if users have a lot of difficulties in getting required components and spend a large amount of time and effort on it, they would rather write their own programs. Currently, the main type of help tool provided by OOP systems is some kind of 'browser' with which users can inspect the software components in a system to access the required information. The System Browser of the Smalltalk system, for example, is representative of this kind of tool. While such browsing tools provide users with some help, this kind of browser is not adequate. A briefly analysis of the Smalltalk's System Browser below will help to illustrate the drawbacks of this approach.

1.2.1 An introduction to the System Browser of Smalltalk



```
intersect: aRectangle
    "Answer a Rectangle that is the area in which the receiver overlaps with
    aRectangle. "

    ↑self species
        origin: (origin max: aRectangle origin)
        corner: (corner min: aRectangle corner)
```

Figure 1.1. System Browser window.

The System Browser provides access to the entire Smalltalk class library. Its window is divided into five scrollable panes and two switch panes labelled **class** and **instance**. The top four panes are termed **list panes**, while the bottom pane is a **text pane** (see figure 1.1). List panes contain fixed lists of menu selectors. Each item in the list is selectable but cannot be edited directly. These panes are scrollable. To view all the available items within a list pane, it may be necessary to scroll through all the contents of the list pane. Text within a text pane may be scrolled, selected, edited and evaluated. To help programmers move around in the library, it is indexed. Related classes are grouped together into class categories, and related methods within individual classes are grouped into message categories. The four list panes, therefore, provide four levels of indexing into the class library. From left to right, these panes are termed the **class categories pane, class names pane, message categories pane**, and **message selectors pane** respectively (see figure 1.1).

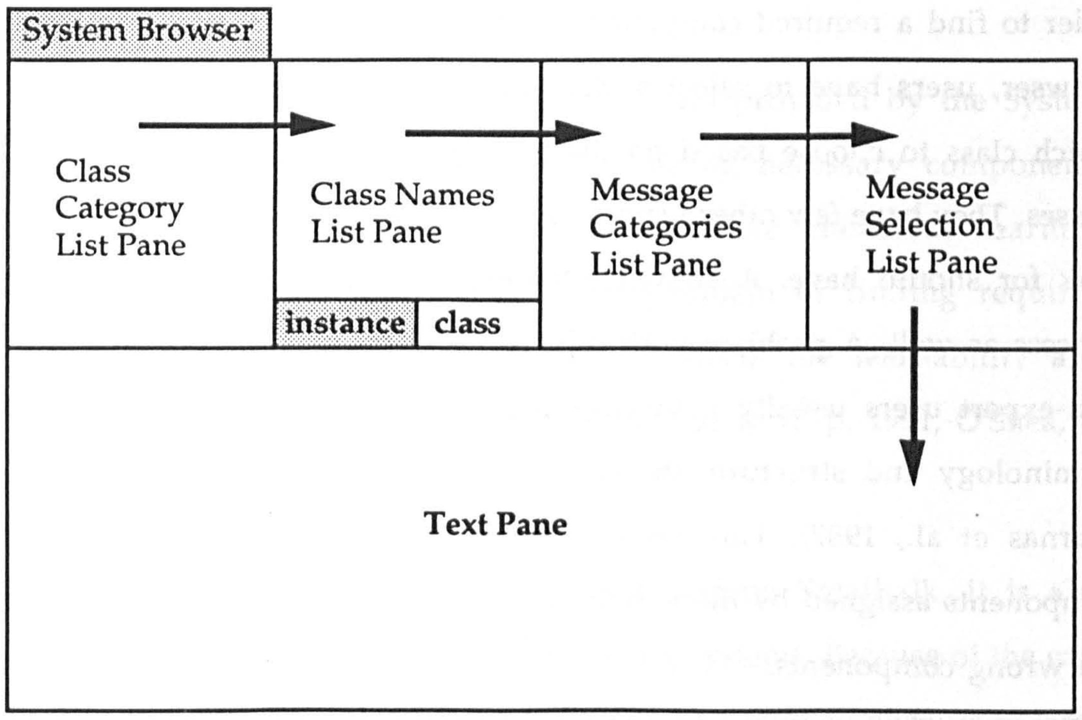


Figure 1.2. Structure of a System Browser.

To look for a class needed, a user first needs to select a class category which contains the class in the **class categories pane**. All classes in that category are then displayed in the **class names pane**. The user can then choose to view a class in the class names pane. Once a class is selected, all message categories in the class will be shown in the **message categories pane.** And, in order to see individual methods, the user chooses a method category. All messages in that category are then presented in the **message selectors pane.** The user can select a message in the message selectors pane, the implementation code of the message is then displayed in the **text pane** on the bottom. The dependencies between the panes of the browser is illustrated in figure 1.2.

1.2.2 A limitation of the System Browser

The System Browser has the advantage that users can move around quickly in the system and inspect any piece of code they want. However, in terms of which components the users should look at in such a large system, it is not problem-free. From the introduction in the last section, it can be seen that in order to find a required component (a class or a method) with the System Browser, users have to select a class category. They then need to decide which class to choose based on their understanding of the names of the classes. They have few other ways to specify what properties the classes they look for should have. A similar situation applies to the method-finding process as well. A problem with this 'retrieval by names' approach is that non-expert users usually have only a limited knowledge of the system's terminology and structure, therefore there is a great vocabulary barrier (Furnas et al., 1987). The users often misunderstand the names of the components assigned by the system designers and consequently may select the wrong components.

Since names of the components do not always convey enough information to users, they experience difficulties in finding the required components. They may have to choose a wider range of components, then study their functions in detail, often by reading the implementation code. The distributed nature of Smalltalk code makes this understanding process fraught with difficulties (Nielsen et al., 1989, Taenzer et al., 1989). Since the code to perform certain functions is usually distributed among many classes, users need to gather all pieces of relevant code from different classes in order to form a complete picture about how a function is performed. Therefore, to understand a function, sometimes they may need to understand the functions of many related components. In addition, the task of understanding the execution of a given function is exacerbated by the polymorphism and dynamic binding of Smalltalk, which, although are very useful techniques in system development itself, make it hard to understand which method is being executed during a computation. As a result of these difficulties, users sometimes find themselves in a situation where they either overlook the relevant components or after making considerable efforts in studying a particular set of components, realize that those components are in fact not relevant to their tasks.

From this analysis, we can see that the facilities provided by the System Browser make non-expert users' task of finding necessary components (classes or methods) difficult and that there are enormous learning overheads. It has been reported that the problem of finding required components is an important factor which affects the learnability and usability of this object-oriented programming system (Esp, 1991, O'Shea, in press).

Although the above discussion is mainly based on Smalltalk, it is also applicable to other object-oriented programming systems. Because of the ease of adding new objects into systems, OOP systems tend to encourage large libraries of components (Halbert, 1986). It would not be difficult to understand that at a certain stage, users would have similar problems to those we have discussed in relation to Smalltalk. It is essential therefore for OOP systems to provide supporting tools to help users find reusable components. Existing tools, such as the System Browser in Smalltalk, are useful but not sufficient. It is necessary to develop other tools.

1.3 The proposed solution

The proposed solution to the problem we have discussed above is to augment browsing tools like Smalltalk's System Browser with a query tool. When users give a description of their target components, the tool will search the library to provide a list of candidate components whose functions match the users' descriptions. Users can then choose the components they require from this list. Such a tool should reduces users' learning overheads, since they then need only study the relevant components. In this way, users are saved time and resources which would otherwise be wasted on studying irrelevant components, and non-expert users should find that their overall difficulties are significantly reduced.

Guided by this motivation, we have developed a tool to help users find reusable components. The design of the system is based on the paradigm known as 'retrieval by reformulation' originally used successfully in the domain of information retrieval. We have chosen Smalltalk as the target system since it has a large component library and it is known that users of this very system have difficulties in finding required information. In addition, Smalltalk is one of the most important OOP systems. It adheres consistently to the object-oriented programming paradigm and it is often recommended to newcomers to the object-oriented programming community as the best system to use in learning OOP-related concepts (Saunders 1989). The results of the research thus have important implications for other object-oriented systems.

One other point to note is that, although the tool discussed here is aimed mainly at helping non-expert users, it also has the potential to help expert users. Object-oriented programming systems are usually very large. It has been suggested that with very large software system, there are few experts who have complete mastery of all components (Draper, 1984). As an example, in an empirical study on the learnability of Smalltalk, more than 30 experts associated with the design and implementation of the Smalltalk family of languages, nevertheless identified parts of the Smalltalk-80 class hierarchy (O'Shea, in press) with which they personally were unfamiliar.

1.4 Methodology of this research

In outline, the methodology of carrying out this research was as follows: First of all, a prototype system named BRRR1 was built, which was then tested empirically on a group of users. From this empirical evaluation, a number of problems with this first prototype were identified. Based on the evaluation results, a second prototype system called BRRR2 was developed. This system version incorporated changes addressing the problems found in the evaluation of BRRR1. After BRRR2 was implemented, it too was empirically evaluated with a group of Smalltalk users. In the light of this second empirical study, the strengths and weaknesses of this tool were evaluated and the implications of this work for research in this area is discussed. In the next section, we briefly outline the organization of the description of this work.

1.5 The organization of the thesis

The thesis consists of seven chapters. In chapter two, we survey work relevant to this research, setting it within the context of other research previously carried out in this area. We outline work done in the area of helping users learn Smalltalk, in the area of browser design and in the area of database querying. In the final part of this chapter, we describe in detail the 'retrieval by reformulation' paradigm upon which this research is based.

In chapter three, the first prototype system we developed, BRRR1, is described. An example of using BRRR1 to retrieve required components is used to give an overview of how the system works. After that, we describe the underlying design principles of the system and show how the 'retrieval by reformulation' principle is reflected in the system. We then present the design details of the system, which is described in terms of the components which constitute the system: the software component library and the interface of the system. In the final part of this chapter, an overview of the implementation of the system is given.

In chapter four, we report on an empirical evaluation of BRRR1. This was a formative evaluation undertaken to expose possible problems in our initial design. The organization of the study and the data analysis process is then described. This is followed by a discussion of the problems found in the study. This study revealed problems on both the system design and the organization of the study itself. In concluding this chapter, we outline the changes which it was felt would address problems identified in the evaluation.

In chapter five, we describe the second prototype system developed: BRRR2. This is an improved version of BRRR1 and it incorporates a number of changes, based on findings obtained in the evaluation of BRRR1. In this chapter, again, an example is used to illustrate how this system works. We then introduce the design details of the system with emphasis on the differences between the two systems, explaining the benefits that are expected to be gained from the improvements incorporated in the second version. In the final section of this chapter, an overview of the implementation of the system is presented.

In chapter six, we report on an empirical study conducted to evaluate BRRR2. This study also took the form of formative evaluation. It was conducted with a group of ten Smalltalk users. The organization of this study was similar to that of BRRR1. Nonetheless, some changes were incorporated, based on experience gained in the evaluation of BRRR1 earlier. We report the results of the study and analyse the strengths and weaknesses of BRRR2.

Finally, in chapter seven, we summarize our research and draw conclusions regarding the work undertaken. We outline the main contributions of this research to the area of helping non-expert users learn and use the Smalltalk system; and at a more global level to the field of facilitating software reuse in object-oriented programming systems; and furthermore to the area of interface design in general. In the last part of this chapter, we suggest directions in which this research could usefully be extended.

Chapter 2 Related research

In this chapter, the work relevant to this thesis is reviewed. The purpose of this review is to provide a context for the research reported here and outline the approaches to the problem discussed in the previous chapter. The chapter is organized as follows: first of all, work done in the area of helping people learn and reuse object-oriented systems, particularly Smalltalk are described. Secondly, systems employing various browsing approaches are discussed. Following this, systems using information retrieval techniques are examined, and finally the approach on which this thesis is based, the retrieval by reformulation paradigm, is described in detail.

2.1 Help available for users learning Smalltalk

In this section, we look at briefly the work in the particular area of helping people learn Smalltalk. This includes short tutorials and graphical trace tools. Their benefits and limitations are also discussed.

2.1.1 Short tutorials

Short tutorials consist of classroom teaching and usually hands-on experience. Normally, pre-designed written materials are used to show users basic concepts and principles of Smalltalk (for example, Kaehler et al., 1986, Gray et al., 1990). Embedded in the written materials are a series of successive, carefully designed example programs used to illustrate the concepts and typical usage of core components of the system. The series of programs often in the end leads to a complete, small scale application (for example, the example and exercise series in Kaehler's book "A taste of Smalltalk" results in an object-oriented style solution to the problem: 'Tower of Hanoi'). After reading the written materials, learners typically are asked to complete some exercises based on the example program learned. During exercise sessions, help from human tutors is usually available.

Rosson et al. (1990a, 1990b) went a step further. They designed a Smalltalk tutorial which included: written instructions, on-line example programs and a software tool. They designed several example application programs which illustrate the basic concepts of the Smalltalk system (e.g. message passing; inheritance). These examples also show users the interface construction paradigm in Smalltalk: the *'Model-View-Controller'* model (Krasner et al., 1988) which is the part widely recognized as one of the most complex and difficult parts in the system. To help learners understand how a user interface program works, they developed a software tool called View Matcher (Carroll et al., 1990). View Matcher consists of several coordinated windows. One type of window incorporated in View Matcher is a modified 'System Browser' of Smalltalk. In this altered browser, only components pertaining to the application examples are shown to reduce users' learning overheads. View Matcher also has a window to show a method stack containing some methods of the application program waiting to execute. The selection of the methods in the stack is done by the system designers, so that the important methods for the application example would be shown in the stack. When an application program is run, through the method stack, users can see at certain points which method is executed (i.e. which message is sent to which object) and see the effect of the execution immediately. Learners can also select any method in the stack and the system would present in its explanation window a text explanation message about the functional role of the method in the example program. After users have learned the examples, they are asked to make changes to the example programs and then use the View Matcher to see the effect of their changes.

The purpose of short tutorials of this type is to provide users with a starting point from which to explore the system. When users finish the tutorial, they are expected to have basic ideas about the purposes of different system components. They thus have the background to investigate more complex and advanced features of the system.

Though these kinds of tutorials cover the basic components of Smalltalk, they do not directly help users find components according to their particular requirements; therefore, this approach does not address directly the problem discussed in chapter one of aiding users to find reusable components.

2.1.2 Graphical trace tools

A program written in an object-oriented program language usually consists of a set of objects. These objects send messages among each other to complete a computation. Sometimes, the message passing thread can be too complex for users to remember or to follow and can cause difficulties for users in understanding the functions of certain objects. To help users understand the functions of components and to enhance their comprehension of concepts of the object-oriented paradigm, various graphical trace tools have been developed. The main idea is to show users explicitly and dynamically how different objects interact with each other by sending messages. Two such tools are outlined below:

a) Diagram system

Cunningham et al. (1986) developed a diagram tool for the Smalltalk system which generates diagrams to illustrate the message sending dialogue that takes place between objects participating in an object-oriented computation. In this system, objects in a diagram are represented by boxes, labelled by the object's class and possibly its super classes. Messages are represented by directed arcs from the sending object to the receiving object. The diagrams are automatically constructed from Smalltalk code. While an application program is running, users are shown a diagram illustrating explicitly the objects taking part in the computation and the messages sent between objects. Thus, users can see how an object participates in the computation and what messages are invoked by a particular message. For example, in figure 2.1, an 'add: anElement' (note that the argument 'anElement' is not shown in the figure) message is sent to an OrderedCollection. An OrderedCollection implements an array of variable size. It responds to the message 'add: anElement' by adding the object represented by the argument: 'anElement' at the next available location within itself. It can be seen from figure 2.1 that the *'add:'* method makes use of two more elementary methods, *'size'* (in its own class) and *'at:put:'* (inherited from its superclass).

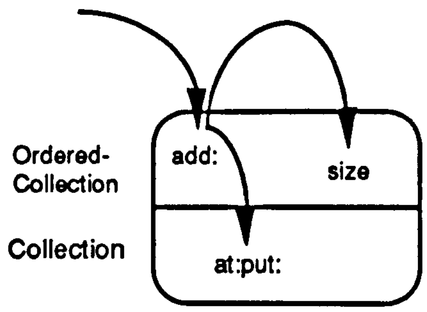


Figure 2.1. A message sending diagram.

b) TRACK

Böcker at al. (1990) built a graphical trace system called TRACK for Smalltalk based on a similar idea. TRACK is a kit to construct interactively environments that trace the execution of methods and the flow of messages between objects. It enables the user to set up traces by means of direct manipulation. Users can interactively select objects represented by icons and put obstacles between them much in the way a jumping course is set up. An obstacle is a kind of filter, with which users are able to specify the type of messages to be traced. As a program is running, the traced methods or messages are graphically animated as a little ball moving among objects.

These messages are also simultaneously presented to users in a text form. Thus, TRACK allows users to project visually the behaviour of programs.

This kind of graphical tool makes the relationships between objects more explicit and consequently to a certain extent enhances users' comprehension of functions of some complex components in the system. It is therefore helpful to users in deciding whether or not to use a component in their later programming. These tools, however, only work after users have chosen some components and constructed an example with them. They do not directly show users how and where to find the candidate components in the first place. Therefore, in the context of helping users find required components, the support provided by this kind of tools is limited.

In the following sections of this chapter, work more directly aimed at helping users retrieve information in information systems is surveyed. This work can be divided into two main streams: browsing methods and database querying methods. While the emphasis here is on finding software components, since both are relevant, techniques used in the domains of hypertext and information retrieval are also reviewed. The discussion begins with browsing tools.

2.2 Browsers

Smalltalk's System Browser is one of the best known browsers in practical use in object-oriented programming systems. However, there have been some other browsers proposed which aim to facilitate users' access to information. Several of these are reviewed below.

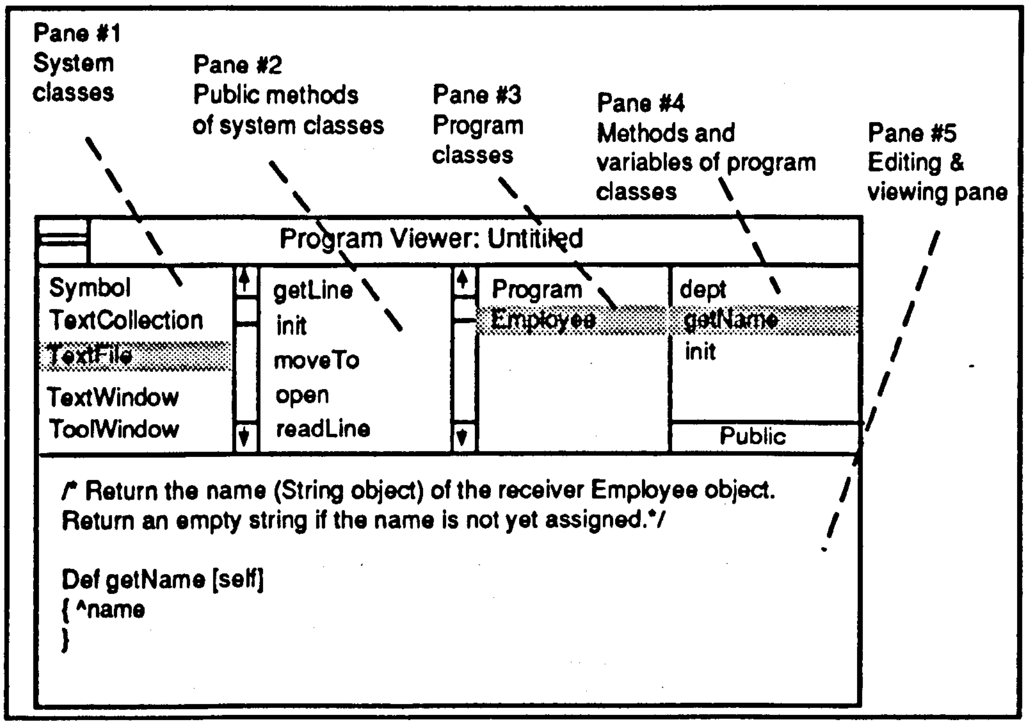
2.2.1 Program Viewer

Wu (1990) argues that a main cause of difficulties encountered by beginners in learning object-oriented programming is that they cannot visualize the entry points of their application programs. In the traditional programming approach, there is a main program that has a flow of control — initialization, input, processing and output — that users can follow and understand. There is nothing comparable in the object-oriented programming environments. The current browsers in object-oriented programming environments show all the system's classes, and beginners are lost as to where to begin. In addition, with those browsers, users cannot get the whole picture of what are and what are not parts of their program.

Wu therefore suggests using a pre-defined class that can serve as an entry point to the program, i.e. the role of the main program in traditional languages. He also indicates that a browser should enhance the concept of abstract data types promoted by the object-oriented paradigm. This should be done by showing users only the public interface of a class (i.e. the messages which are available to other objects). Meanwhile, the implementation code of a method of a system class should initially be hidden from users.

Based on these ideas, he proposes a new browser for object-oriented systems named 'Program Viewer'. It is intended to help non-expert users. In Program Viewer, classes are divided into two groups: *system classes* and *program classes* and are shown to users in two separate panes (see figure 2.2). The *system classes* are the generic classes available to all programs. The *program classes* are the classes specific to the application program a user is working on. For a system class, only its public methods are shown to users. In addition, for any method of a system class, only the definition of the method is presented; its implementation code is hidden. The system classes can only be used by users and cannot be modified by them. Furthermore, for any class, all its methods (including the inherited ones) are shown; this saves users having to navigate through the class hierarchy to see if a class responds to a particular message.

In the 'Program classes' pane, a class called 'Program' is included automatically. It serves the role of the main program in a traditional language. This class provides with templates for methods such as 'initialize' and 'close' to show users the start and finish points of a program. Users can start programming from the 'initialize' method. Wu feels that as classes shown in 'Program classes' pane are those which are specific to the program being developed, it gives the user a sense of security because these are the classes to which modifications can be done as desired since they are the individual's classes. This browser also uses different colours to represent different type of methods and instance variables.



```
/* Return the name (String object) of the receiver Employee object.
Return an empty string if the name is not yet assigned.*/

Def getName [self]
{ ^name
}
```

Figure 2.2. The 'Program Viewer' browser.

This system has some advantage over the Smalltalk browser since it hides the implementation details of a system component from beginners and provides a more complete image about the function of each class. This reduces the users' overhead in searching for information. However, with this browser, users still need to find components by name-based browsing, as in the conventional Smalltalk browser.

2.2.2 Multi-dimensional browser

Traditionally, class or type based object-oriented programming systems group methods according to one dimension: classes or types. Types are arranged into a type hierarchy according to inheritance. In order to find a method, users usually need to navigate through the type hierarchy with the hierarchy browser. Ossher (1990), however, argued that it would be useful to group methods according to two dimensions: both types and generic functions*, because each method is an implementation of a particular generic function for a particular object type. He suggests that organizing methods by generic functions would be helpful for browsing in that:

i) Programmers wishing to write an implementation for a generic function for some type can conveniently examine other implementations of the same generic function.

ii) Programmers are able to find all types that support a generic function.

iii) Once programmers want to change the meaning of a generic function, they can find and change all methods implementing that function.

Ossher therefore proposes a two dimensional browser for an object-oriented extension of Pascal, embedded in a system known as RPDE[3] (Harrison, 1987). With this browser, components are displayed as a two dimensional function/type matrix. In the matrix, each row corresponds to a generic function, and each column corresponds to a type (see figure 2.3). Each row has a header, shown as a shaded box in the figure, which stores information

* A generic function provides an interface to a number of methods, and the method to be invoked is determined by the classes of the arguments to the function.

about the associated generic function, such as its name, its parameters and its semantics. Each column also has a header which stores information about the associated type, such as its name, its instance variables, documentation and a semantic specification.

When they start using the browser, users indicate types and/or generic functions of interest, the system then presents users with a display which includes those types and generic functions together with as much surrounding context (i.e. other types or functions considered useful) as will fit in the display window.

If a type implements the method for a generic function, then the corresponding position in the matrix is marked with a asterisk. Users can directly click on an asterisk to examine the code of the method it represents. If a method in a type for a generic function is inherited from another type, then the corresponding position in the matrix is marked with an upward arrow. Users can also click on the arrow to see directly the code of the method; they don't need to navigate through the inheritance hierarchy to search the code as they must do with Smalltalk-like browsers.

This browser emphasizes the equal importance of both type and generic functions in terms of browsing. It adds one more dimension, generic functions, for users to browse information. Furthermore, it visually displays the relationships between generic functions and associated types. Therefore, it facilitates users' examinations of components to find information. Nevertheless, it does not tell users which functions or types they should be interested in or which they should examine; thus it is expected that users would encounter problems similar to those experienced by users of the Smalltalk browser.

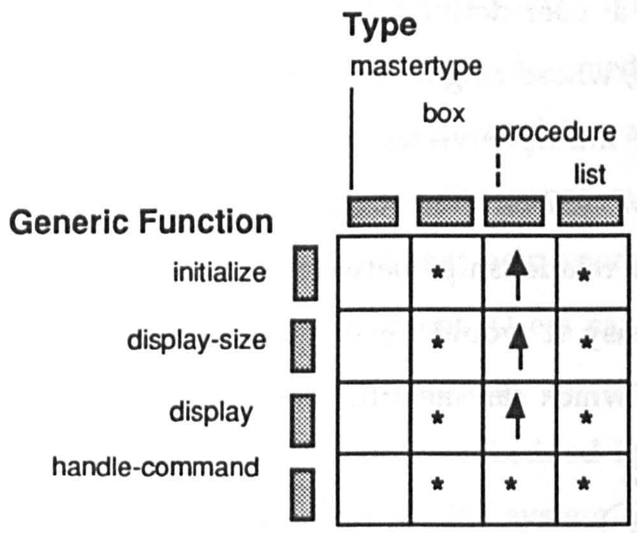


Figure 2.3. An illustration of the multi-dimensional browser.

2.2.3 Affinity Browser

Affinity Browser (Pintado, 1990; Gibbs et al., 1990) is a browser designed to help users select required objects (classes) and explore relationships among the objects in very large object-oriented systems. It is based on the notion of 'affinity' between objects. Affinity between two objects is defined as a relationship with an associated intensity (i.e. a value specifies the 'weight' of the relationship). An object has affinity to another object. The same group of objects may be viewed from several different contexts, called 'views'. Each view has associated with it an affinity function defined by system designers or users, which defines the intensity of a relationship. Moreover, the affinity between a pair of objects may be different in one view from that in another view. The browser displays a group of objects in a two dimensional display according to a view selected by users. The affinity between objects is transformed into a distance relationship so that objects with strong affinity are displayed close together, while those with less affinity lie further apart. Users can decide the similarities between objects based on the closeness of these objects (see figure 2.4 and 2.5). Once users select an object to examine, the browser also displays the objects that are within a user-defined affinity neighbourhood (i.e. those that have an affinity with the current object which is greater than a user-defined limit). Users then can examine those neighbour objects, whose neighbours may in turn be displayed. In addition, users can create multiple views to see different relationships between objects. This approach is interesting in that it attempts to use graphical display to convey relationships between objects. However, it is not clear at this stage how easy it would be to define comprehensible and effective affinity functions which can meet the non-expert users' requirements.

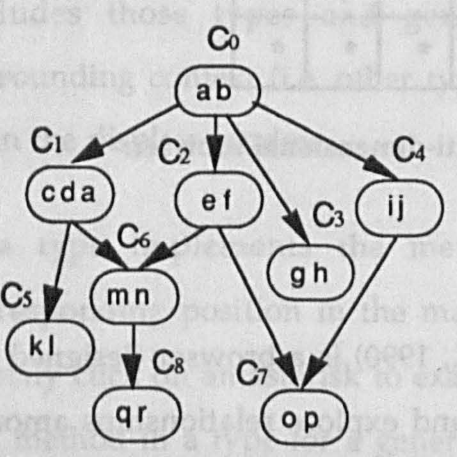


Figure 2.4. Inheritance structure of a set of classes. Ci is a class name and a, b, c, ... represent methods of a class.

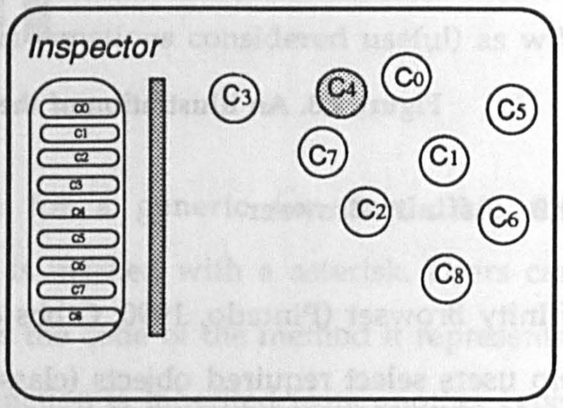


Figure 2.5. The Affinity Browser Display of the objects shown in figure 2.4.

Besides the programming systems, many hypertext systems (Conklin, 1987) also use browsers to access information stored in the systems. Described below are two hypertext browsers: 'Document Examiner' and 'SuperBook'. They are somewhat different from more general hypertext systems in that they are not designed to support the more creative writing applications. Instead, they are used to present users with online information. The information to be presented is organized into very large documents which are highly structured and not frequently modified.

2.2.4 Document Examiner

Document Examiner (Walker, 1987) is a hypertext system. It is a delivery interface which provides online access to all the documentation to

Symbolics computers (approximately 8000 printed pages). It has been a part of the release of Symbolics software since 1985. It was designed to provide faster and richer access to documentation initially created in paper form, and it makes extensive use of the organization of the paper materials. It has a window consisting of several subwindows that help users manage various aspects of their interaction with the document. Users can directly ask the system to present materials about a specific topic by entering the name of it. The text presented to users has various links embedded in it, so that users can follow these links for cross-referencing. The system also has a simple query capability. Users can enter a group of keywords, the system then performs a keyword or substring search (i.e. find the words which contain the string input by users as part of the word) and shows users in a subwindow all retrieved sections which contain those words in their titles or *keywords* field. Users can select a record presented by the system and use the 'table of contents' command to see the subsections of it. The system can also present an 'overview' on a user selected record. An overview contains contextual information, i.e. related records, and is displayed in the form of a graph in a temporary window. This mechanism enables the user to determine whether or not the selected section is relevant and, if it is relevant, whether it or something related to it is more appropriate. In addition, the system has a 'bookmarks' window to record the history or state of users' interaction with a document for their later reference. This tool allows users to access the documents in multiple ways and this to a certain degree helps them explore the information base to find required data. However, as we will discuss later after section 2.2.5, this tool has some limitations.

2.2.5 SuperBook

The SuperBook system (Remde et al., 1987) is a text browser intended to provide improved access to existing online information by employing several cognitive tools suggested by research on human-computer interaction. SuperBook uses a version of the 'fisheye view' (Furnas, 1986) technique. Initially, users are presented a hierarchically organized 'table of contents' of each document in the system. Users can open up selected parts of the hierarchy to the desired depth, leaving the coordinate and direct ancestor nodes of the focal point visible for contextual orientation.

It also uses a rich indexing technique. All words in the documents can be used to search for information of interest. In addition, users can specify their own synonyms to the words appearing in any document. These synonyms are also used by the system to perform a search. This technique to some extent overcomes the problem that users fail to find desired information in a textual database because they often describe what they are looking for in terms different from those by which the material is indexed (Furnas et al., 1983; Gomez et al., 1984).

These techniques may be used in combination. For example, users can first provide some words to ask the system to show all documents containing those words. After the SuperBook displays a 'table of contents' which shows all sections incorporating the words, users can use the 'fisheye viewer' to examine some sections in more detail. They can also add their own aliases to some of the words to facilitate their later retrievals. The use of the 'fisheye view' technique helps users focus on the relevant information. However it still requires users to recognize the names of the topics they need. Its query capacity, on the other hand, is limited as we will discuss in section 2.3.

So far, we have reviewed various browsing approaches used to help users access information in programming systems and hypertext systems. Although those techniques help users in various way and to a certain extent facilitate users' information seeking process, most of them still suffer from the same problem as that of the Smalltalk browser we discussed in the previous chapter. They mainly allow users to access information according to their names (with the exception of the Affinity Browser) and do not provide adequate mechanisms to allow users to directly specify the information they are interested in. The Document Examiner and SuperBook do have some query capacities, but they are based mainly on keyword match techniques. It will be shown in the following sections that these techniques are not satisfactory. As suggested in the previous chapter, it is necessary to augment the browsing approach with a query mechanism. In section 2.3, we survey systems based mainly on database query techniques, while in section 2.4 we describe the 'retrieval by reformulation' approach on which this research is based.

2.3 Approaches based on querying techniques

With the query approach, users input their requirements to the system in some system-acceptable forms, the system then performs a search and presents users with answers. The following systems illustrate this approach.

2.3.1 Keyword based retrieval systems

This approach is identical to the one used in the domain of bibliographic information retrieval (Salton et al., 1983). The basic idea is to index the software components in a database with a set of keywords assigned by the system designers or users. Users' requirements (i.e. the queries) are also represented as a set of words or phases which they think characterize their requests. The system then searches the database to find the components whose descriptions 'best' match against the users' queries.

a) The simplest match method of the keyword based approach is word to word match. If the set of keywords used to index a component includes those input by users, then the component matches the query. The query mechanism in Document Examiner described above is such an example.

This approach is also adopted by the REUSE system (Arnold et al., 1988). In REUSE, all software components are classified into four types: *Template, Module, Package* and *Program*. Components within each type are assigned keywords appropriate to that type. On retrieval, users are prompted to first specify the type of the required component. They then provide some keywords or phrases which characterize their requirements. The system searches the database in the same way as was mentioned in last paragraph. To improve retrieval, REUSE incorporates a system thesaurus, which is a dictionary containing keywords together with their synonyms. This increases the possibility of user-input words matching those used to index the components.

b) Other systems use automatic indexing techniques (Salton et al., 1983) to classify the components. In those system, instead of assigning a set of keywords to each component manually, the indexing is done by a computer software. The system analyses the text description for each component and automatically extracts the important words as indices to the component. The CATALOG (Frake et al., 1987) system employs such a technique. It is an information retrieval system for storing and retrieving C software components. In CATALOG, each component is characterized by a set of single-term indices that are automatically extracted from the natural-language headers of C program. In addition, the query interface of CATALOG allows full boolean combinations of search terms. That is, search terms can be connected by the logic operator: *and*; *not*; and *or*. For example, users can enter a query such as:

((sorting *and* routines) *or* quicksort) *not* heapsort.

This query would retrieve components about sorting routines or quicksort, which are not about heapsort. The system also supports partial string match techniques such as automatic stemming and the use of wild card characters.

A more complex automatic indexing and retrieval technique has been used by Helm et al. (1991) in a tool developed for reusing a library of C++ user interface classes. This tool has both querying and browsing capacities and in it each index generated by the system has associated with it a numerical weight. The indexing process is as follows. First of all, a document is built for each class in the library considered important by the system designers. The document consists of a natural language description of the class taken from the manual of the system. This description specifies the function of the class as well as the function of each public method of the class. An automatic analysis is then performed for each document from all documents in the system to extract the indices which best characterize the document. The indexing scheme they used is to consider an index as a word or a set of words that occurs with a significant frequency in the document. Through a statistical analysis, a numerical weight that represents the relative importance of the index to a document can then be generated and associated with each index. The set of indices together with their weights for a document forms a *profile* of the document. From the document profiles, an inverted index is produced to allow storage and subsequent retrieval of documents. To retrieve components, users specify a query in natural language. The query is then treated as if it is just another document and subsequently indexed using the same indexing technique used for the original documents. The profile extracted from the query is then compared to those stored in the system using a specially defined similarity measure. Based on their similarities to the query, retrieved classes are ranked for presentation to the users. This system also makes use of domain knowledge of object-oriented programming systems. For example, if a class matches a user's query, all its subclasses would also be presented.

This tool also allows users to browse classes. To support browsing, profiles of the documents for classes can be organized into browsing hierarchies using a numerical clustering technique (van Rijsbergen, 1979) based on similarities between classes. A group of classes which are similar measured against a similarity threshold are organized into a cluster, then several clusters are grouped again to form a higher level cluster. The newly formed clusters can form even higher level clusters until a hierarchy is created. The hierarchy is produced purely from the documents of classes rather than from the class structures (e.g. the inheritance relationship). Thus its designers claim that the browsing hierarchy built in this way provides a means to browse among functionally rather than structurally related components.

The GURU software library system (Maarek et al. 1991) uses the same approach.

c) A final example of the keyword based approach is the RSL system (Burton et al, 1987), which provides an interesting interactive scoring mechanism to help users select components. In the RSL system, components are classified into categories. Each component belongs to a category which is identified by a category code. Each component also has a set of ten attributes describing its characteristics. Some attributes are: *overview* (i.e. a text description about the function of it); *author* (i.e. the person who wrote the component); *keyword* (i.e. a set of up to five keywords used to index it); *algorithm* (i.e. the algorithm used to implement it); etc. For a retrieval in RSL, a user can target his/her query on a particular attribute, for example, asking for components written by a specific author; or asking for components belonging to a particular category by entering the category code. Additionally, a user can input keywords, in which case, the system would do a keyword match. Although it is said by the designers that RSL can also handle natural language queries, no further details are given about the techniques used.

If there are large number of components matching a query, the user can use the scoring mechanism of RSL to help make decisions. Once invoked, the scoring mechanism would prompt the user to specify the required application domain of the components. Based on this information, the system would show the user a group of attributes characterizing components in that domain (e.g. the complexity of the components, the operation of the components and the language used to implement them) and request the user to specify the importance of each attribute by interactively adjusting a group of barometers corresponding to the attributes. From the user's selections, the scoring mechanism searches the library for candidate components, evaluates them against the user's requirements and rates them according to a scoring algorithm. A user can repeatedly adjust those barometers to request the system to retrieve components according to adjusted criteria until they are satisfied with the retrieval result.

While it is useful to let the users interactively and explicitly specify the importance of the attributes, this scoring mechanism is only implemented for a small set of components. The system designers also acknowledge the difficulties of finding a rigourous evaluation method to rate components in the library. In addition, it is not clear how the system can help users identify in the first place the software application domain (i.e. the type of the software) to start the scoring process.

The advantage of the keyword based retrieval approach is that it is straightforward to implement. In particular, the automatic indexing technique offers a cheap way to index and retrieve components. Thus, this technique is normally not difficult to be scaled up and extended in the sense that new classes can be added to the library without major effort. However, this approach is far from perfect. The studies in bibliographic information retrieval where this approach originated indicate that it is often the case that users retrieve much unwanted information and conversely, that they fail to find relevant materials (Dumais, 1988). Furnas et al. (1983, 1987) showed that there is tremendous diversity in the words that people use to describe an object or concept and that this places strict limits on the expected performance of keyword-based systems. If a requester uses different words from the author or organizer of the information, relevant materials will be missed. Conversely, the same word can have more than one meaning which leads to irrelevant materials. As Dumais (1988) rightly points out, since human word use is characterized by extensive synonymy and polysemy*, straightforward term-matching schemes are seriously deficient. The basic problem is that people often want to access information based on conceptual meaning, and there is not a one-to-one-relationship between word choice and meaning.

2.3.2 Prieto-Díaz's faceted classification scheme

The REUSE and CATALOG systems described above use the hierarchical classification scheme to classify software components. Many software catalogues use this scheme as well. With this scheme, a universe of knowledge is divided into successively narrower categories and categories are arranged into a hierarchical structure. Each component to be classified is assigned to one of the categories. Prieto-Díaz et al. (1987, 1991) and Jones et al. (1988) argue that this classification scheme does not facilitate reuse. In this scheme, it is difficult to determine a single category to assign a component because some components can belong to more than one category. In addition, with this scheme, maintaining and expanding the existing hierarchy is a difficult task because the adding of new components may cause rearrangement of the hierarchy or the generation of many cross-references from related classes.

Prieto-Díaz et al. proposed a software classification scheme based on the faceted classification scheme used in library science. In the faceted scheme, subject statements are analysed into their component elemental classes

* Polysemy: existence of many meanings (of words etc.).

called facets. Each facet contains a group of descriptors called terms. A component is classified by selecting the most appropriate term from each facet to best describe the component. This results in a new, synthesized class that is tailored to the individual component. In their scheme, a software component is classified with six facets: three related to the functionality of the component and three related to its environment. The three functionality facets concern the actual computation function performed, the object on which the function would be performed, and the medium through which the function would be applied to the object. The environmental facets have to do with the role of the function within the system, the functional area of the application, and the external setting in which the function will be used. Within these facets are numerous terms describing how the system could be classified on that facet. The search for a reusable component is accomplished by entering a query with six terms into a relational database that contains components, documentation and so forth. Synonyms for a term are not allowed in a query, but a thesaurus is provided to aid users' search for the correct term.

Within a facet, terms are structured around certain supertypes that represent organizing concepts. The conceptual distance between two terms can be measured as the cumulative distance between the two terms and the supertype to which they both belong. These distances are assigned by the user, and the conceptual graph can be used during a component search session to find a reasonable alternative component description to search for if the original query failed to produce a component. The development of individually weighted conceptual graphs relies on users' willingness to input the data needed to construct them.

This classification scheme may be successful when users are familiar with the meaning of the terms assigned by the classifier. In this case, they can select the terms from different facets to construct the query. However, for non-expert users, who are not very familiar with the domain, there may be difficulties in understanding these terms and consequently difficulties in using them.

2.3.3 The 'conceptual dependency' approach

Wood et al. (1986, 1988) developed a retrieval system for software components (mainly Unix commands) based on the approach of 'conceptual dependency' originally developed in the domain of natural language understanding (Schank, 1972, Waltz, 1977, 1978). Conceptual dependency is a representational system that encodes the meaning of sentences by decomposition into a small set of primitive actions. The core of such an approach is a number of fundamental concepts which are sufficient to capture the semantics of any domain of interest. The concept categories can relate to each other in specified ways. These relations are called *dependencies*. Wood et al. suggest that software components can be described by three fundamental types of concepts: actions, nominals and modifiers. Actions correspond to the basic, fundamental functions that software components perform. Nominals correspond to the objects that perform the function (i.e. the software component itself), objects that the function manipulates, objects produced as a result of the function and objects that provide a context for the action. Modifiers describe actions and nominals. Wood et al. analysed the software component domain and obtained a set of basic functions for software. A set of conceptually similar verbs is then identified for each basic function. In addition, all objects manipulated by software components are classified into classes or 'nominals'. They then developed for each basic function a 'component descriptor frame' to capture the relationships between the 'action' and the 'nominals'. Such a frame is based around the function and has slots for the objects manipulated by the component. All components are then classified based on the basic functions by filling the slots of each 'component descriptor frame' with values.

During the retrieval, initially the user is prompted by the system to input either a verb describing the action the component performs, or a noun representing an object manipulated by the component. The system finds a skeleton frame corresponding to the action (or an object) which conceptualises the verb (or noun). It then continually prompts the user to input words to fill the remaining slots of the frame. Based on the information input, the system performs a search to find the relevant components.

This approach attempts to capture more semantics of the software component descriptions than the keywords representation scheme, and therefore to a certain extent overcome some of the problems of the keywords system. However, this approach has not yet been evaluated other than in the original context. It does raise the question as how large a task it would be to generalize this approach into other contexts.

2.3.4 Structured database systems

The structured database is intended to manage a large quantity of information. Unlike the keywords based retrieval approach on which most systems mentioned above are based, the structured database systems are used to manage highly structured information. The information stored in these databases is typically organized in terms of relations, but sometimes involves data structured in trees or more general networks. The relationships between objects or attributes need to be explicitly specified. Users use specially designed query languages to generate queries to retrieve information stored in these systems. However, many of the query languages associated with the structured database systems are not easy to learn for non-expert users. Some of them like SQL (Elmasri et al., 1989) and SQUARE (Boyce et al., 1975) require many hours of instruction to learn, and others have a syntax which users find difficult to use and understand (Tou, 1982).

Take SQL as an example. SQL is a widely used query language for databases based on the relational data model. In order to generate queries, SQL requires users to know in advance which tables (relations) and attributes they will be needing. For example, for the query "Find the names of employees in department 50", its SQL expression is:

```
Select Name
From Emp
Where DeptNo=50.
```

For this example, users have to know from which relation (here the relation: Emp) to choose which attributes (the attribute: Name) based on certain criteria (the DeptNo=50). This is not always an easy task for non-expert users. They often are not familiar with the terms used to specify the attributes, and they have difficulties in specifying the values corresponding to those attributes (Tou, 1982). In addition, this language presumes that users can articulate a query precisely in advance. Users, especially the non-expert ones, however, often have only a vague idea about what they need and consequently have difficulties in articulating the query exactly beforehand (Fischer, 1989).

Another database approach which seems promising in terms of facilitating non-expert users making a query is the 'retrieval by reformulation' paradigm exemplified by the system RABBIT. We describe it in detail in the following section.

2.4 Retrieval by reformulation

'Retrieval by reformulation' is a paradigm based on a theory of human remembering and is used in the domain of information retrieval. The core of this paradigm is two techniques of human remembering: descriptive retrieval and retrieval by instantiation, which are introduced below.

2.4.1 Descriptive retrieval

A theory of human remembering (Norman et al., 1979) postulates that people retrieve information from memory by iteratively constructing a description of the target item. Norman et al. propose that the retrieval starts with a description of the desired information as an initial specification of the records sought from memory. This retrieval description guides the memory search process and helps determine the suitability of retrieved records for the purpose of the retrieval. The initial description can be modified as intermediate information becomes available during the retrieval cycle. This idea of accessing memory through descriptions is extended by Williams et al. (1981) to include the notions of iteration and reconstruction. Williams et al. suggest (1981, p. 118):

> Information about the target item is used to construct a description of some aspects of the item. This description is used to recover a fragment of information about the item which is added to what is known. From this information, a new description is formed to retrieve still more information, until the particular piece of information sought can be recovered.

They also suggest that the retrieval process has three stages: *find a context* in which a proper environment for conducting a search is recovered; *search* in which bits and pieces of information appropriate to the context are recovered until an adequate description can be formed within the search context; and *verify* in which the information recovered is checked against the original query. If the information retrieved satisfies the original query, the retrieval terminates at this point. Otherwise, the retrieved information is used to reformulate the description and a new cycle is initiated.

In addition to being iterative, the retrieval process is also recursive (Norman et al., 1979, Williams et al., 1981). Each of the three stages can have within it one or more recursive calls to the retrieval process. The establishing of a context, for example, may itself require a retrieval cycle, involving the finding of a context, search and verification. Similarly, during the course of a search, information may be found which is incomplete and thus, requires further search before it can be understood. Finally, the verification stage may also require its own retrieval cycles for the purpose of certifying the accuracy of the information provided by the preceding search phase.

In summary, this theory postulates that humans retrieve information from memory by iteratively constructing partial descriptions of the desired target item. After a partial description has been constructed, a search is conducted to find the information matching the description. If the retrieved information does not satisfy the original query, then that information is used to reformulate the description, and the retrieval cycle is repeated. Moreover, each stage of the cycle can itself be recursive.

2.4.2 Retrieval by instantiation

The technique of descriptive retrieval described above is a general paradigm for retrieving information from memory. However, it does not specify what the fragments of the information retrieved on each cycle are or how they are incorporated into the partial description to make a new description. Williams (quoted from Tou, 1982) postulates that the information retrieved on each retrieval cycle is in the form of an instantiation, i.e. a description of an example item suggested by the partial description. This idea is based on the observation that when people are trying to recall something, they frequently are reminded of items which are similar to or related to their target items. The instantiation serves as a template for the description of the target item by providing a set of descriptors which can be incorporated into the partial description.

2.4.3 Retrieval by reformulation and the RABBIT system

The 'retrieval by reformulation' paradigm (Williams, 1984) was a combination of the two human information retrieval techniques mentioned above and some other ideas which will be described below. It was used as a base for the development of the RABBIT system — an interface for information retrieval. The idea is to let a user retrieve information by incrementally constructing a description of his/her target item using an instantiation provided by the computer. The hypothesis of using the human information retrieval techniques was that the methods used by people in remembering could also be applied to the task of retrieving information from electronic databases. In addition, an interface employing those methods would be in some sense 'natural' to use since the interface would be relying on techniques which people use in recalling thoughts from their own memories.

In RABBIT, users retrieve information by iteratively reformulating their queries. When a retrieval starts, the user inputs an initial description (a partial query) of the desired items, and RABBIT presents him/her with a list of items matching the initial query. One individual of the matched items is presented to the user as an example instance. The user can then select information incorporated in the example instance — the attributes; values; etc. to reformulate the partial query. This new query is then used to retrieve a set of new items and possibly a new example. This process may be repeated until the user is satisfied with the result or it is established that the required items are not in the database.

More specifically, the retrieval by reformulation paradigm is characterized as follows (Williams, 1984, Tou, 1982):

i) *Retrieval by reconstructed descriptions*

The user makes a query by describing the object he/she is seeking.

ii) *Interactive construction of queries*

The query is constructed in an interaction between the computer and the user. The user creates an initial, partial query and then gradually refines it into a better, more complete one using the information provided by the computer. He/she does not have to compose a precise query before using the system. In some sense, there is no query language as is the case with the traditional relational query languages such as SQUARE and SQL. The user does not need to learn a formal query language. All he/she needs to master is the set of several commands (see the following paragraph) used to critique a portion of the example instance presented.

iii) *Critique of example instances*

This is the core of the retrieval by reformulation paradigm. A user employs a set of commands to manipulate information incorporated in examples provided by the system. This aspect of the system results in the user getting a template for the type of object he/she wants to describe, a vocabulary he/she can be certain that the system understands, and access to additional information and terminology within the database. The access to additional information in the database can be achieved by the user selecting a descriptor in the example instance description and asking RABBIT for alternative values or further descriptions.

iv) *Dynamic perspectives*

The information presented to the user (i.e. the examples) should be based on what view users take of the information, that is, it should be based on information in which the users have shown interest. Unnecessary attributes and values should be filtered out. In addition, the view should change with the users' changing queries.

Since RABBIT, there have been other systems which have used the retrieval by reformulation approach. One is called ARGON (Patel-Schneider et al., 1984), which is a system used to store and retrieve personnel information. Another system is HELGON (Fischer et al., 1989) which is used to manage database stored information about literature. It has been reported (Fischer et al., 1989) that this approach is effective in helping non-expert users retrieving information.

In the process of the research reported in this thesis, we decided to use the retrieval by reformulation paradigm as our base to design the query tool for helping users find reusable components in Smalltalk. The main consideration for choosing this paradigm was as follows.

The users of this tool are non-expert users. Faced with Smalltalk, they have already had a considerable amount of learning overheads. It would be inappropriate to design a formal query language that requires a lot of effort to learn before it can be used. The query language designed based on the retrieval by reformulation paradigm aims to help non-expert users make a query and does not seem to impose much extra learning load on users. It thus seems suitable to our purpose.

2.5 Summary

In this chapter, work relevant to this project has been surveyed. This includes the particular help provided for Smalltalk users — short tutorials and graphic trace tools; various browsers for users to access information in programming environments and hypertext systems; systems using information retrieval techniques and the retrieval by reformulation paradigm. The tutorials provide starting points for users to explore the Smalltalk systems, but the help from this approach is very limited. The graphic trace tools make more explicit relationships between objects and enhance users' comprehensions to the functions of software components. However, they do not address directly the problem of how to find reusable components. The various proposals for the designs of browsers provide useful techniques in terms of facilitating users' access to information. However, most of them suffer from problems similar to that of Smalltalk's System Browser. The majority of systems based on query approaches use keyword based retrieval techniques. This kind of technique is easy to implement but is not very effective for helping users to find required information. The retrieval by reformulation paradigm exemplified by the RABBIT system seems to be a promising solution, and was thus selected as a base for this project. In the next chapter, its use in the design of this project is described.

Chapter 3 BRRR1 — the design and application

3.1 Introduction

In the last chapter, we introduced the paradigm on which this project is based: retrieval by reformulation. Described in this chapter is the first prototype tool we developed — BRRR1 (BRowser for Retrieval by Reformulation). The principal idea guiding the design of this prototype is to allow the users to query the system (i.e. give the system a description) about the components they need. The system would recommend them with a list of candidate components according to the description. The users then examine the candidates offered and choose the appropriate ones. A query is constructed iteratively, using the information presented by the system.

The chapter is organized as follows: first of all, an example is shown to illustrate how a query is carried out in BRRR1. Then discussed is how the retrieval by reformulation paradigm is reflected in BRRR1 and some general considerations about the design of the system. Thirdly, we examine the component library and component organization of BRRR1. What follows is an overview of the implementation of BRRR1 and finally, the interface of BRRR1 is described.

3.2 Finding components in BRRR1 — an example

In this section, an example is used to show how users use BRRR1 to find the required components. To facilitate the comprehension of the example, it is first necessary to introduce briefly the interface of BRRR1.

3.2.1 The interface — an overview

The interface of BRRR1 consists of two windows, the Main window and the Method Examination window. The Main window consists of the following five panes (see figure 3.1):

i) Class Descriptors pane (leftmost):

Presented in the Class Descriptors pane is a group of words describing characteristics of the classes, each word is called a *class descriptor.*

ii) Class Query pane (upper to the right of the Class Descriptor pane):

The query constructed by users for retrieving classes is shown in this pane. A class query comprises a set of class descriptors similar to those displayed in the Class Descriptors pane.

iii) Method Query pane (right below the Class Query pane):

The query constructed by users for retrieving methods is presented in this pane. A method query consists of a set of method descriptors which we will describe later.

iv) Matched Items pane (top to the right of the Class Query pane):

Shown in this pane is a list of classes or methods matching a user's query on class or method.

v) Example pane (right below the Matched Items pane):

This pane contains a description of a class or a method included in the matching list in the Matched Items pane mentioned above. Shown here is either the description of a class or a method depending on whether the user is retrieving classes or retrieving methods. This will be explained fully later in section 3.5.1.

The 'Method Examination' window is not shown in figure 3.1. A Method Examination window would be open when a user starts examining the methods in a class. It is from this window that an initial method query is created. This can be seen in the next section where an example of using BRRR1 is presented. In each individual pane except the Matched Items pane, there is a pop up menu which shows a set of commands applicable to that pane and with which users interact with the system. Generally, we tried to design BRRR1's interface similar to that of Smalltalk, so that users can feel more familiar with it. This is to reduce their learning overload, since they should not be distracted too much by learning the operations in BRRR1.

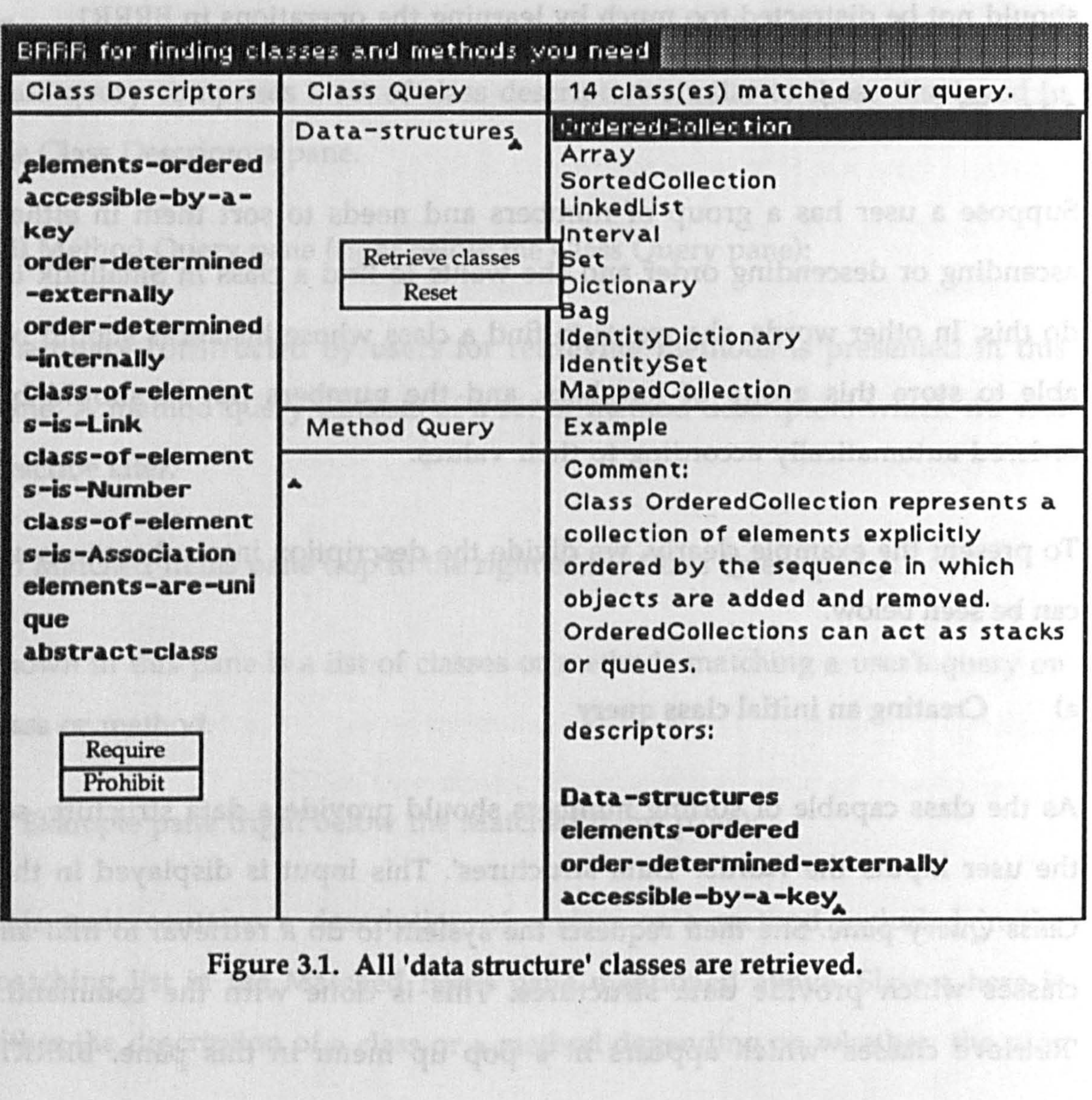
3.2.2 The example

Suppose a user has a group of numbers and needs to sort them in either ascending or descending order and she wants to find a class in Smalltalk to do this. In other words, she wants to find a class whose instances should be able to store this group of numbers, and the numbers put in should be ordered automatically according to their values.

To present the example clearly, we divide the description into subsections as can be seen below.

a) Creating an initial class query

As the class capable of sorting numbers should provide a data structure, so the user inputs the words: 'Data structures'. This input is displayed in the Class Query pane. She then requests the system to do a retrieval to find all classes which provide data structures. This is done with the command: 'Retrieve classes' which appears in a pop up menu in this pane. BRRR1 performs a search and presents in the Matched Items pane all classes which provide some kind of data structures, which are in fact all the Collection classes of Smalltalk. The first class in this pane is highlighted automatically. Meanwhile, a description of the class highlighted in the Matched Items pane is displayed as an example instance in the Example pane (see figure 3.1). The description of the class consists of a text message which explains the function of the highlighted class — the class OrderedCollection, and several class descriptors (the boldfaced text) which are under the heading: 'descriptors'. These descriptors specify the properties of the example class and can be used by users to construct a further query for class.



Comment:
Class OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. OrderedCollections can act as stacks or queues.

descriptors:

Data-structures
elements-ordered
order-determined-externally
accessible-by-a-key

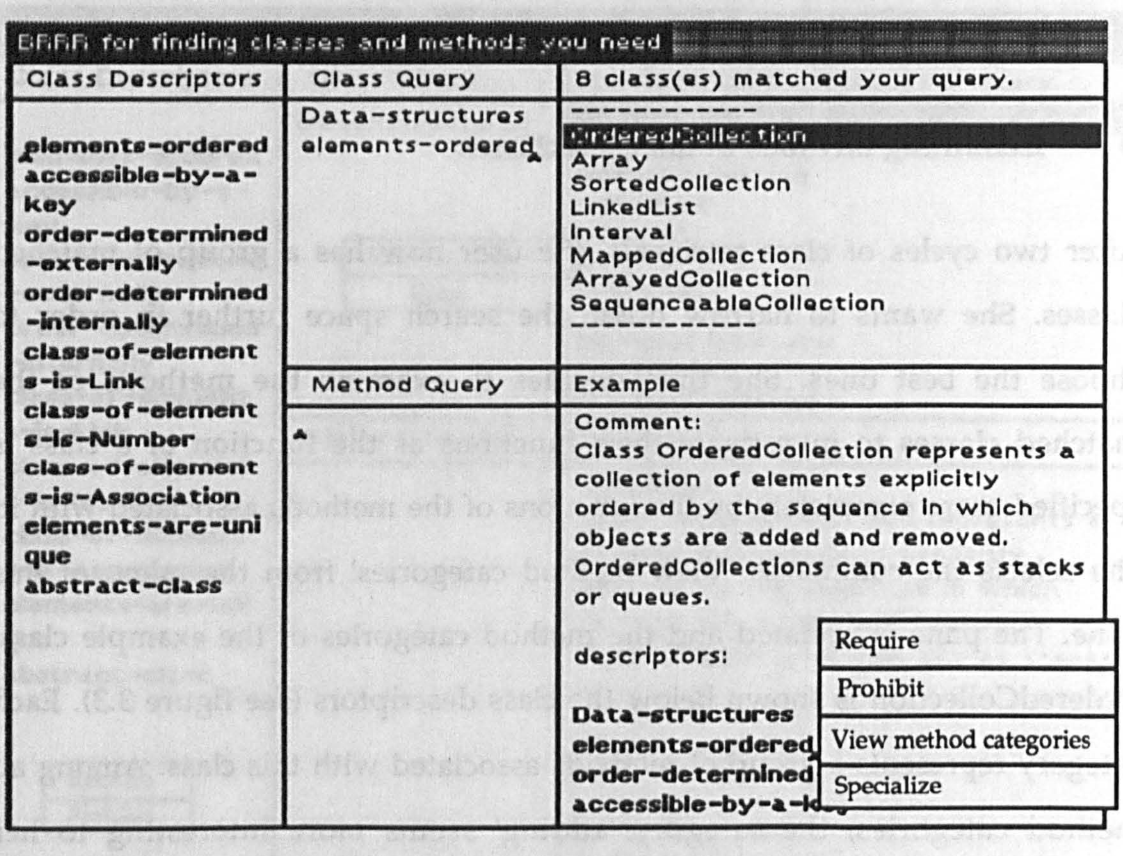
Figure 3.1. All 'data structure' classes are retrieved.

b) Reformulating the class query and retrieving classes

Now, the user examines the class descriptors in the Example pane. She thinks that the instances of the class for which she is looking should be able to keep its elements in some order. She thus selects the descriptor: *elements-ordered* and then chooses the command: 'Require' which appeared in the pop up menu of this pane. After the menu command is executed, this class descriptor is added to the Class Query pane. At this stage, she is not sure which other class descriptors to choose to specify her requirements further. She therefore requests BRRR1 to do a class retrieval. BRRR1 now presents her with all such classes whose instances provide some kind of data structure and can keep their elements in certain order. Note that the Matched Items pane is updated again (see figure 3.2).

c) Examining methods in matched classes

After two cycles of class retrievals, the user now has a group of matched classes. She wants to narrow down the search space further in order to choose the best ones. She thus decides to examine the methods of the matched classes to investigate their functions as the function of a class is specified more completely by the functions of the methods associated with it. She selects the command: 'View method categories' from the menu of this pane. The pane is updated and the method categories of the example class: OrderedCollection is shown below the class descriptors (see figure 3.3). Each category represents a group of methods associated with this class. Among all method categories, the category 'adding' seems more interesting to her because all numbers need to be put into a collection and ordered automatically. She thus wants to see more details about the 'adding' methods. She chooses the category and uses the menu command: 'Specialize'. Once this command is executed, a new window — the Method Examination window is open (see figure 3.4). This window has two panes, the top one contains all methods associated with the class OrderedCollection and which are used to add new objects into a collection. The bottom pane contains the description of the method highlighted in the top pane — the method: 'addAll:'. The description comprises two parts: the first part (plain text) is a text message explaining the function of the method; the second part (the boldfaced text) is called *method descriptors.* Each method descriptor is also named: an *attribute-value pair* with the part before the ':' an *attribute* and the part after it the corresponding *value* of the attribute. The method descriptors specify the function of the method and are used by the user to construct a query for methods.


BRRR for finding classes and methods you need

Class Descriptors

elements-ordered
accessible-by-a-key
order-determined-externally
order-determined-internally
class-of-elements-is-Link
class-of-elements-is-Number
class-of-elements-is-Association
elements-are-unique
abstract-class

Class Query

Data-structures
elements-ordered

8 class(es) matched your query.

OrderedCollection
Array
SortedCollection
LinkedList
Interval
MappedCollection
ArrayedCollection
SequenceableCollection

Method Query

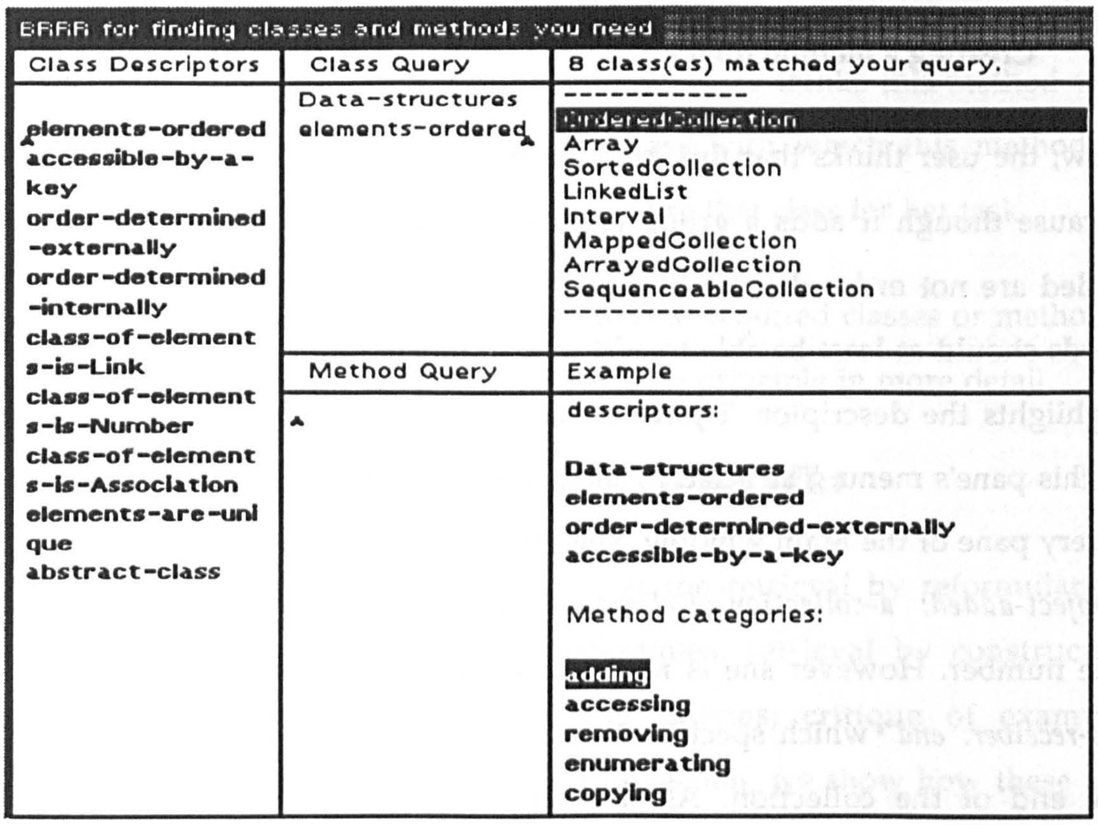
Example

Comment:
Class OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. OrderedCollections can act as stacks or queues.

descriptors:

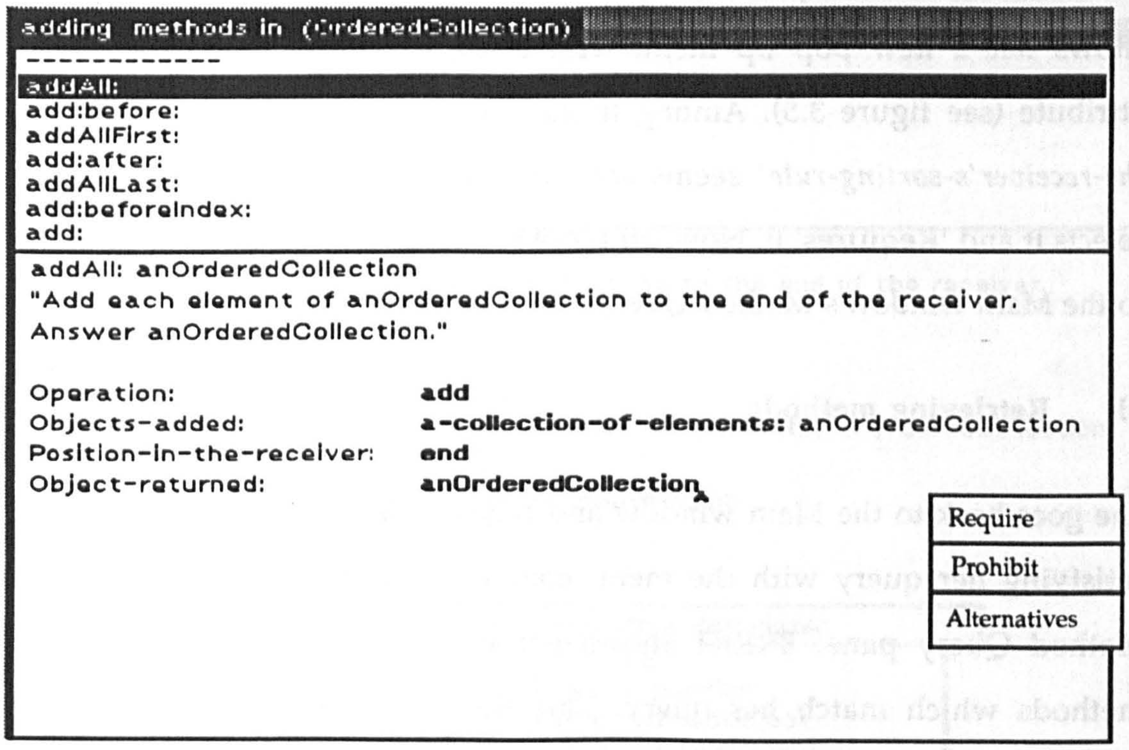
Data-structures
elements-ordered
order-determined
accessible-by-a-key

Figure 3.2. All 'ordered' classes.


BRRR for finding classes and methods you need

Class Descriptors	Class Query	8 class(es) matched your query.
elements-ordered accessible-by-a-key order-determined-externally order-determined-internally class-of-elements-is-Link class-of-elements-is-Number class-of-elements-is-Association elements-are-unique abstract-class	Data-structures elements-ordered	------------ OrderedCollection Array SortedCollection LinkedList Interval MappedCollection ArrayedCollection SequenceableCollection ------------
	Method Query	Example
		descriptors: Data-structures elements-ordered order-determined-externally accessible-by-a-key Method categories: adding accessing removing enumerating copying

Figure 3.3. Method categories of OrderedCollection are shown in the Example pane.


adding methods in (OrderedCollection)

```
------------
addAll:
add:before:
addAllFirst:
add:after:
addAllLast:
add:beforeIndex:
add:
```

```
addAll: anOrderedCollection
"Add each element of anOrderedCollection to the end of the receiver.
Answer anOrderedCollection."

Operation:                  add
Objects-added:              a-collection-of-elements: anOrderedCollection
Position-in-the-receiver:   end
Object-returned:            anOrderedCollection
```

Figure 3.4. The Method Examination window.

d) Creating a method query

Now, the user thinks that this example method is not quite what she needs because though it adds a group of elements into a collection, all elements added are not ordered according to their values. However, the method she needs should at least be able to add new numbers into a collection. She thus highlights the descriptor: '*Operation: add*' and uses the command: '**Require**' in this pane's menu. The selected method descriptor is shown in the Method Query pane of the Main window. She then '**Requires**' the method descriptor: *'Object-added: a-collection-of-elements'* since she needs to add more than one number. However she is not satisfied with the descriptor *'Position-in-the-receiver: end'* which specifies that the objects added in would be put at the end of the collection. As she needs to sort numbers, therefore the numbers added in should be put in positions based on their values. She therefore selects the descriptor and chooses the command 'Alternatives' from the menu of this pane to request other values for this attribute. BRRR1 shows her a new pop up menu with a list of values it knows for this attribute (see figure 3.5). Among those, the value: *'postion-determined-by-the-receiver's-sorting-rule'* seems satisfying her requirement. She therefore selects it and '**Requires**' it. Now, all those required descriptors have been sent to the Main window's Method Query pane (see figure 3.6).

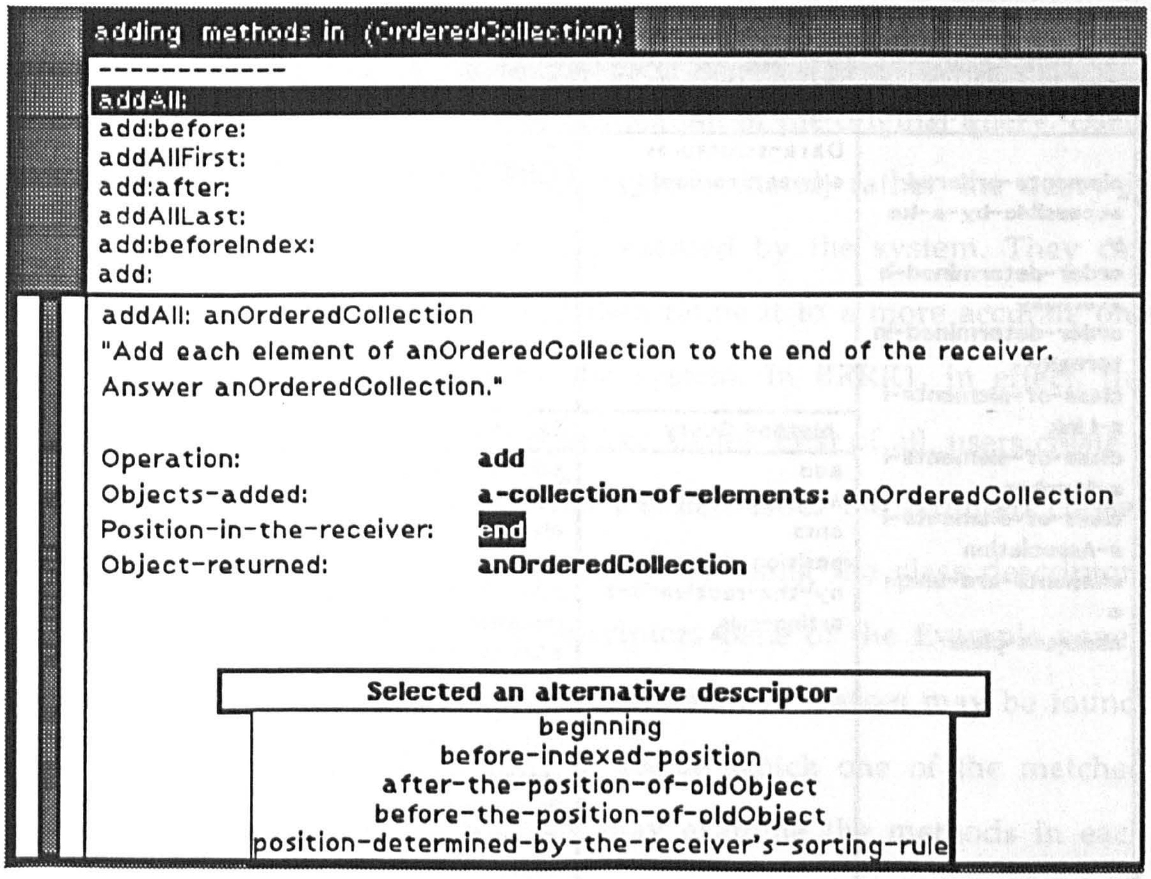
e) Retrieving methods

She goes back to the Main window and requests BRRR1 to find all methods satisfying her query with the menu command: '**Retrieve methods**' in the Method Query pane. BRRR1 shows her in the Matched Items pane the methods which match her query. This time, only one method: 'addAll: (SortedCollection)' matches the query (see figure 3.7). The text in the bracket: SortedCollection is the name of the class with which the method is associated. Note that, after the retrieval, the Example pane is updated again, the description of the method 'addAll: (SortedCollection)' is now displayed in this pane. After reading the description, the user thinks this method will perform the function she needs. As the class with which this method is associated is SortedCollection, she decides to use that class for her task.

This example shows how BRRR1 is used to find required classes or methods. In the following sections, we describe its design principle in more detail.

3.3 The retrieval by reformulation paradigm in BRRR1

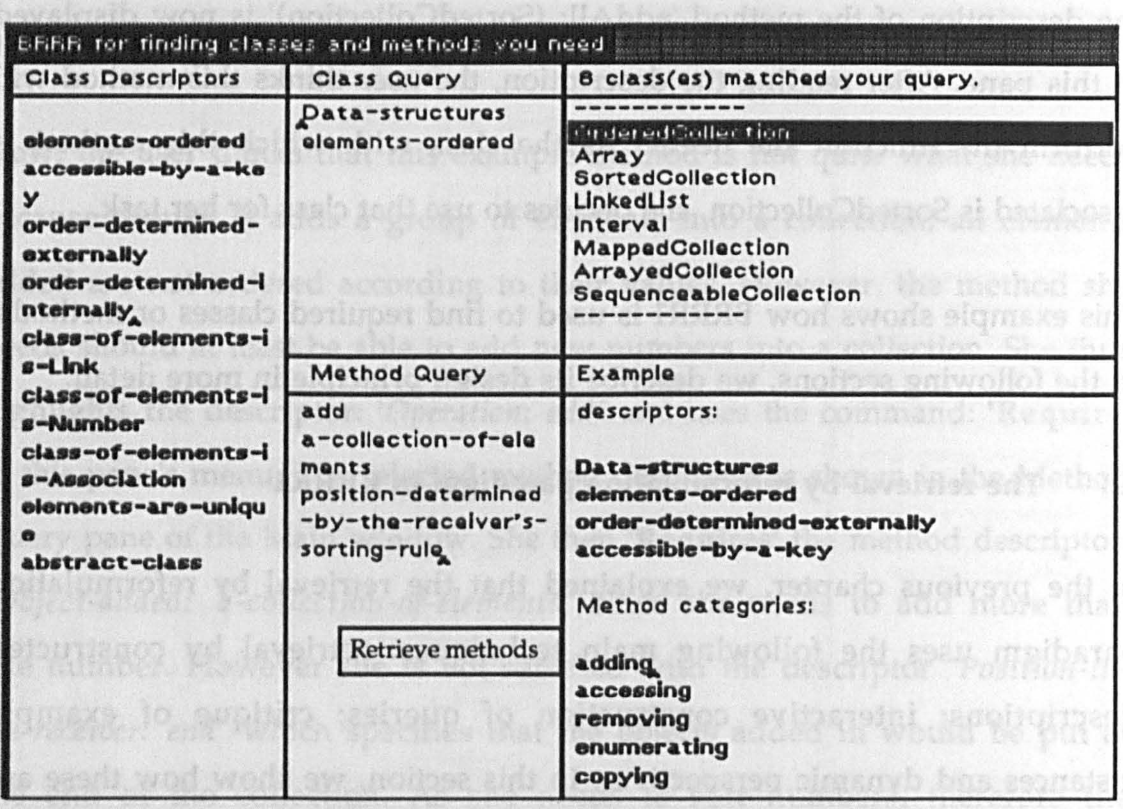
In the previous chapter, we explained that the retrieval by reformulation paradigm uses the following main techniques: retrieval by constructed descriptions; interactive construction of queries; critique of example instances and dynamic perspectives. In this section, we show how these are reflected in the design of BRRR1.


adding methods in (OrderedCollection)
----
addAll:
add:before:
addAllFirst:
add:after:
addAllLast:
add:beforeIndex:
add:
Selected an alternative descriptor
beginning
before-indexed-position
after-the-position-of-oldObject
before-the-position-of-oldObject
position-determined-by-the-receiver's-sorting-rule

```
addAll: anOrderedCollection
"Add each element of anOrderedCollection to the end of the receiver.
Answer anOrderedCollection."

Operation:                    add
Objects-added:                a-collection-of-elements: anOrderedCollection
Position-in-the-receiver:     end
Object-returned:              anOrderedCollection
```

Figure 3.5. Alternative values to the value: 'end' of the attribute: 'Position-in-the-receiver' are shown in a pop up menu.



BRRR for finding classes and methods you need

Class Descriptors

elements-ordered
accessible-by-a-key
order-determined-externally
order-determined-internally
class-of-elements-is-Link
class-of-elements-is-Number
class-of-elements-is-Association
elements-are-unique
abstract-class

Class Query

Data-structures
elements-ordered

8 class(es) matched your query.

OrderedCollection
Array
SortedCollection
LinkedList
Interval
MappedCollection
ArrayedCollection
SequenceableCollection

Method Query

add
a-collection-of-elements
position-determined-by-the-receiver's-sorting-rule

Retrieve methods

Example

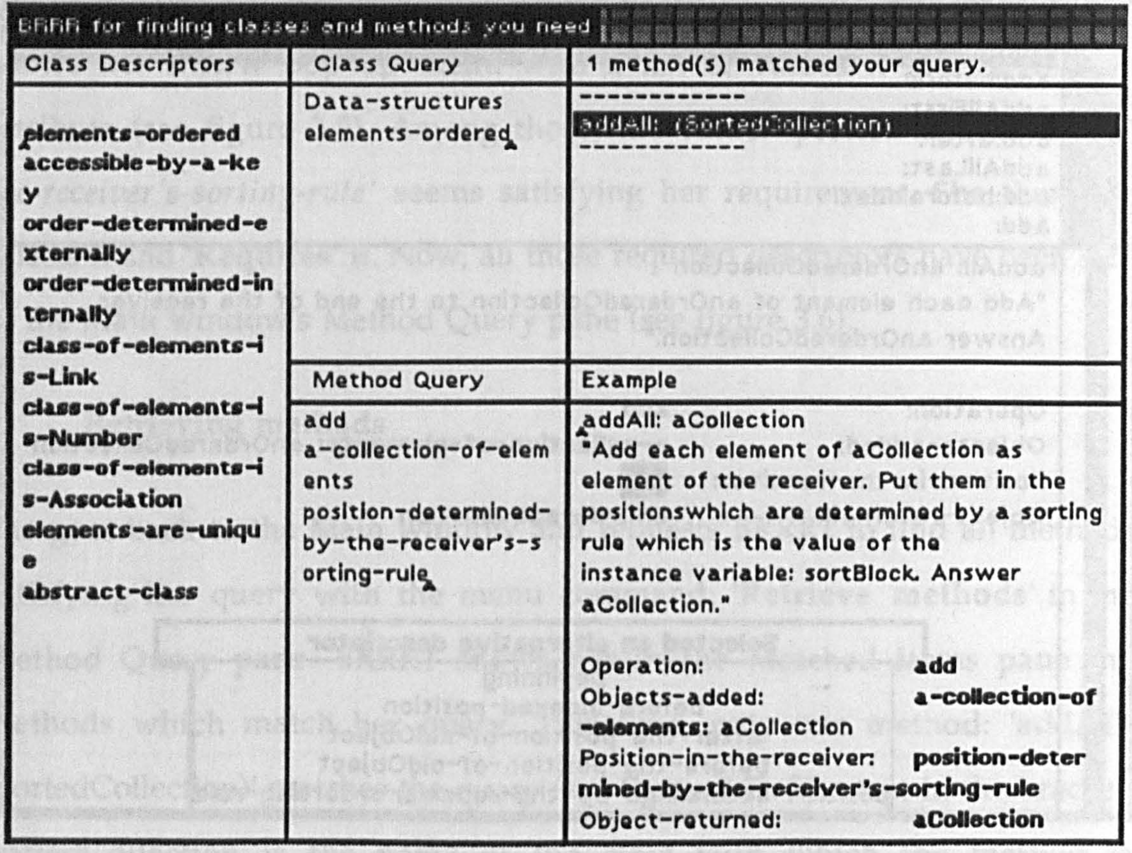
descriptors:

Data-structures
elements-ordered
order-determined-externally
accessible-by-a-key

Method categories:

adding
accessing
removing
enumerating
copying

Figure 3.6. The method query constructed so far.



BRRR for finding classes and methods you need

Class Descriptors

elements-ordered
accessible-by-a-key
order-determined-externally
order-determined-internally
class-of-elements-is-Link
class-of-elements-is-Number
class-of-elements-is-Association
elements-are-unique
abstract-class

Class Query

Data-structures
elements-ordered

1 method(s) matched your query.

----
addAll: (SortedCollection)

Method Query

add
a-collection-of-elements
position-determined-by-the-receiver's-sorting-rule

Example

addAll: aCollection
"Add each element of aCollection as element of the receiver. Put them in the positionswhich are determined by a sorting rule which is the value of the instance variable: sortBlock. Answer aCollection."

Operation:	add
Objects-added:	a-collection-of-elements: aCollection
Position-in-the-receiver:	position-determined-by-the-receiver's-sorting-rule
Object-returned:	aCollection

Figure 3.7. The result of the method query.

3.3.1 Retrieval by reconstructed descriptions

In BRRR1, users make a retrieval by constructing a description of the desired components (classes or methods). The components in BRRR1 are the basic reusable components of Smalltalk, i.e. classes. Each class has associated with it a group of methods. Therefore, in BRRR1, the descriptions are of classes and methods. Users describe the required classes with the class descriptors and the required methods with the attribute-value pairs. Each descriptor represents an aspect of the function of a class or a method, users can therefore retrieve a class or a method based on several functional aspects of it. The descriptors provide more information about the function of the component than just the name of the component. This should be more helpful for users in finding the required components than the primarily name-oriented scheme used in the original Smalltalk browser.

3.3.2 Interactive construction of queries

Retrieval in BRRR1 is a process of reformulation of the original query. Users do not have to compose an exact query beforehand, rather the query is created gradually using information presented by the system. They can initially construct a partial query and then refine it to a more accurate one using the information fed back by the system. In BRRR1, in effect, the retrieval of components is completed in two stages. First of all, users create a query for classes which specifies what characteristics the required classes should have. The class query is composed by using the class descriptors (select them from either the Class Descriptors pane or the Example pane). After several cycles of class retrievals, a number of classes may be found. However, the users may still need to decide which one of the matched classes to choose. At this point, they may examine the methods in each matched class, or if there are many matched classes, they can create a method query to retrieve required methods. After finding the required methods, they would be able to know with which classes those methods are associated and then choose the most suitable ones.

The reason for finding methods of classes is that the methods of a class specify the function of the class in more detail. Therefore, it is necessary for the users to investigate the methods of a class to understand its function better. An important concept supported by object-oriented programming systems is data encapsulation (Pinson et al., 1988; Blair et al., 1991). An object encapsulates both data and a set of operations (i.e. the methods) which are allowable on the object. The representation of the data is protected and users can only access the object through the set of operations which is referred to as the interface of the object. Therefore, from an external point of view, it is the set of operations or methods of an object which collectively defines the function of the object. In Smalltalk, a class represents a group of similar objects — its instances, the methods of a class specify the interface of all its instances. To understand fully the function of a class, it is thus necessary to inspect and understand the methods of the class. Furthermore, for users, to determine if a class is required, an important criterion should be that the class has the kind of methods which they need. This process is equivalent to asking the system to find the class which satisfies the conditions specified by the class descriptors; and which should have such methods that they satisfy the conditions specified by the method descriptors. To find the required methods from the matched classes, users can construct a method query — this is the second stage of the query process in BRRR1. It is through the method query that the users further specify the requirements to the class they need.

From the example described in the last section, it can be seen that again, a method query is created gradually. The users first select one of the BRRR1 method categories which represents the methods in which the users are interested. They can then use the example method provided by the system to create a method query. If the methods given by the system do not satisfy their requirements, the users can reformulate the query still further. This should be particularly helpful in the situation where users cannot form a complete image about the components they want, and have difficulties in creating a complete query in advance. Furthermore, users do not need to learn a formal query language, which reduces their learning overheads.

3.3.3 Critique of example instances

In BRRR1, users reformulate the partial query with the information which is generated by the system on receipt of the users' initial query. The information is in the form of a description of a component (a class or a method) which matches the users' partial query. The description consists of a comment which describes the function of the component (class or method) and a set of descriptors (class descriptors in the case that the example instance presented is a class; attribute-value pairs in the case of a method). The example component is a concrete example of the components satisfying the users' query. It is an instantiation of the abstract specification of the query description. It serves as a template for users to describe their target components. The descriptors embedded in the description of the example component can be used to reformulate the previous query. In the case of the method query, the description of an example method provides a pattern to construct a method query. The values of the attributes used in describing the example method facilitate users' understanding of the meaning of the attributes. In addition, the text comment about the function of the component provides a context in which the descriptors are used. This should also help in resolving the possible ambiguities of the descriptors. Moreover, as users only need to select the descriptors presented by the system, they are supplied with a vocabulary which is guaranteed to be recognizable by the system.

Another aspect of the example is that it provides users access to additional information within the system (this happens during the method retrieval through the uses of the **'alternative'** command). It is likely that the example method presented to users is not quite what they need. The example is however related to the target ones since it satisfies the initial query. By choosing some attributes and exploring the alternative values from the description of the example, users may find other descriptors which suitably describe their requirements.

3.3.4 The dynamic perspective

This technique is used in the RABBIT system mainly to filter out the unnecessary attributes for a data item in a conventional database because a data item there may have large amount of attributes. For example, a restaurant may have attributes: name, location, cuisine, reservation requirements, manager, turnover etc. Not all of them are interesting and necessary for all users. Some users may only be interested in the cuisine and reservation requirements while other people may be interested in its business aspects (for example, its manager, its income). Thus for different user requirements, only the relevant information (attributes) should be presented while the irrelevant ones should be hidden. Furthermore, the attributes presented to users need to be changed dynamically when users select different categories of data to examine. This is useful when the data stored in the system is heterogeneous. In BRRR1, however, a class or a method has relatively limited amount of attributes, thus in this first version of BRRR1, this technique is not implemented.

So far, we have showed an example of how BRRR1 is used to help users find components and described in general how retrieval by reformulation is reflected in BRRR1. In the following sections, we describe the design of the system. The system consists of two main parts: the component library and the interface. The component library provides a mechanism to organize (i.e. store and index) reusable components for retrieval. The interface aids users to query and examine components to find the required ones. We look at each of them in turn and the library is described first below.

3.4 The component library of BRRR1

In BRRR1's component library are components (classes) taken from Smalltalk. They are reorganized to suit our retrieval paradigm. Before we describe components organization in BRRR1, first we briefly introduce what basic reusable components are in Smalltalk and how they are organized there.

3.4.1 The components organization in Smalltalk

In Smalltalk, the basic reusable components are classes. Each class has a set of methods associated with it. For convenience, the methods associated with a class will be referred to as 'methods of the class' or 'methods in the class'. For a single class, the number of methods in the class may be large, therefore all methods of the class are classified into several method categories. Each method category contains a group of functionally similar methods. An example of this structure: the class OrderedCollection and methods associated with it are illustrated in figure 3.8.

All classes in Smalltalk are arranged into a inheritance hierarchy in which each class, except the class Object, has a superclass and may have several subclasses. The class Object is at the top of the hierarchy. All methods in a class are inherited by its subclasses. A simplified illustration of the class hierarchy of Smalltalk is shown in figure 3.9.

Another layer of organization to the classes are class categories. All classes are classified into class categories and a class category represents several functionally relevant classes. There is, however, no further structure imposed on individual class categories, all class categories being at the same level.

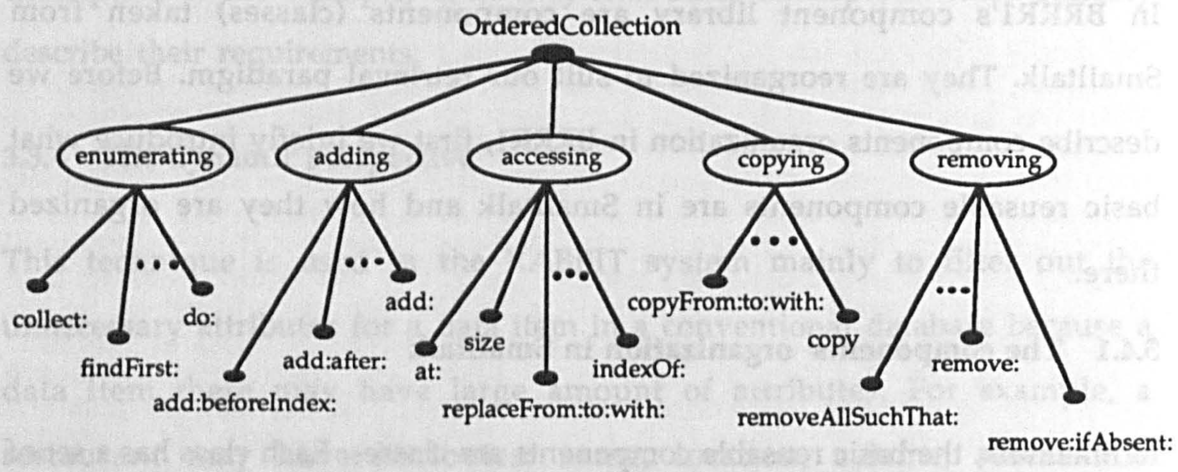


Figure 3.8. The class OrderedCollection and the methods in it. The nodes in the middle of the tree represent method categories. The leaf nodes represent methods.

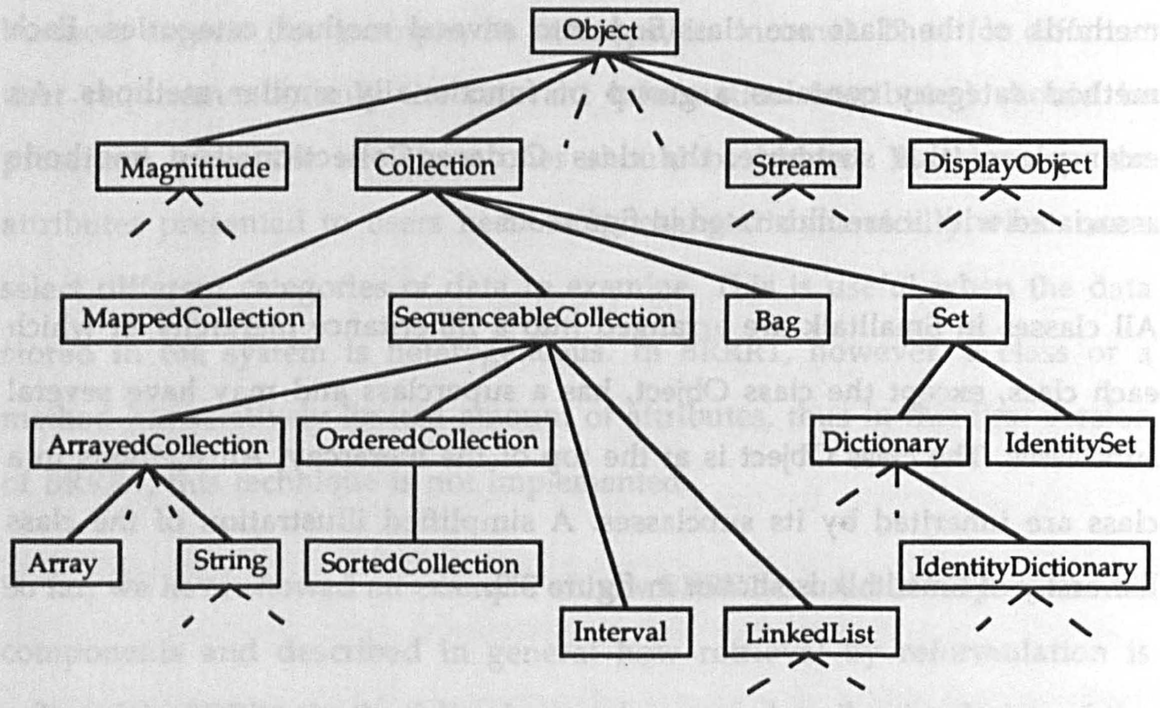


Figure 3.9. An illustration of the inheritance hierarchy of classes in Smalltalk .

3.4.2 The component organization in BRRR1

BRRR1 is a prototype system to demonstrate our approach, thus only a subset of the components in the original Smalltalk system is stored in its library. The classes we chose to store are the 'Collection' classes in Smalltalk's class hierarchy. The Collection classes are a cluster of classes which serve as containers to other objects and are often used as bases to construct various data structures. The inheritance hierarchy for some of the Collection classes can be found in figure 3.9 (i.e. all classes in the hierarchy under the node: 'Collection').

The Collection classes have similarities in that all of them are used to store other objects. However, there are also differences among them, for example, some classes are ordered, (i.e. their instances can maintain an order on elements stored in them, the class OrderedCollection and SortedCollection are such examples) while others are not (for example, the class: Set). Some of the classes are accessible by external keys (i.e. objects stored in instances of these classes can be directly accessed through indices, for example, the class Array and Dictionary). To find an appropriate class among those Collection classes to complete a task, users need to distinguish the functions of different classes. Finding the required classes in this group of classes therefore seems to represent a sufficiently complex and typical situation where non-expert users try to get access to required information. In addition, the Collection classes are so frequently used in Smalltalk programming that for non-expert users, help should be provided for practical use. It is for these reasons that the Collection classes were chosen as the test bed and used in BRRR1's component library.

In BRRR1, both classes and methods are classified. The classification scheme for classes is different from that for methods. The classes are classified according to class descriptors and the methods are classified into method categories. We explain them further in the following sections.

a) The class classification

In BRRR1, in contrast to the class organization in the original Smalltalk system, each class in the library is indexed by one or several phrases named 'class descriptors'. Each class descriptor represents a property and is associated with one or several classes which have the property. For example, the descriptor: 'elements-ordered' specifies the property that instances of a class are able to keep their elements in some order. Therefore, all Collection classes which are ordered are indexed by this descriptor. The selection of indices (i.e. the class descriptors) for all Collection classes relies on the properties used to distinguish the functions of the individual classes. After analysing the functions of all Collection classes, several criteria were chosen to classify them. These criteria together with the descriptors used to represent them are listed below:

i) Is the class ordered (i.e. do instances of a class maintain an order on their elements)? The corresponding descriptor for this property is: 'elements-ordered'.

ii) Is the order determined by the order in which the elements are put in or removed? The corresponding descriptor for this property is: 'order-determined-externally'.

iii) Is the order determined by the class' own criteria? The corresponding descriptor for this property is: 'order-determined-internally'.

iv) Is the class accessible by a key (i.e. can elements of instances of a class be accessed by external indices)? The corresponding descriptor for this property is: 'accessible-by-a-key'.

v) Can elements stored in instances of the class only be numbers? The corresponding descriptor for this property is: 'class-of-elements-is-Number'.

vi) Can elements stored in instances of the class only be instances of the class Association? The corresponding descriptor for this property is: 'class-of-elements-is-Association'.

vii) Can elements stored in instances of the class only be instances of the class Link? The corresponding descriptor for this property is: 'class-of-elements-is-Link'.

viii) Are elements in instances of the class unique (i.e. any element appears in the collection only once)? The corresponding descriptor for this property is: 'elements-are-unique'.

ix) Is the class an abstract class? The corresponding descriptor for this property is: 'abstract-class'.

The first three descriptors (i, ii, iii) deal with the order of a collection. The fourth specifies the way in which elements in a collection are accessed. The three following descriptors (v, vi, vii) deal with type restrictions on objects a collection can store. The meanings of the last two descriptors (viii, ix) are self-explanatory.

b) The method organization

In Smalltalk, each class has associated with it a group of methods. The methods in each class are classified into several method categories. Each method category represents several functionally similar methods in the class. In BRRR1, this class-method relationship is still maintained, in other words, each class knows what method categories it has and what methods are in each method category. For example, in Smalltalk, the class OrderedCollection has the following method categories:

adding: representing methods used to put new objects into the collection.

accessing: representing methods used to access elements in the collection.

removing: representing methods used to delete existing elements in the collection.

copying: representing methods used to make copies of the collection.

enumerating: representing methods used to sequence through all elements in the collection to perform a computation.

In BRRR1, the class OrderedCollection still has the same set of method categories and for each method category, the same set of methods. This ensures that when users select a class, they are able to find the same set of methods as would appear in the original Smalltalk's System Browser.

However, to support method queries in BRRR1, it is necessary to impose a new layer of structure on methods to give the system a capacity to retrieve methods across different classes. A classification method used in BRRR1 is to categorize methods. All methods in *all* Collection classes are classified into several method categories referred to as '*BRRR1 method categories*'. Each BRRR1 method category represents the methods in *all* Collection classes which have similar functions. The BRRR1 method categories have the same names as those in original Smalltalk but have a wider scope. They are listed below:

adding;

accessing;

copying;

removing;

enumerating.

The '**adding**' represents methods in all Collection classes used to put new objects into Collections*. An example method of this category is the method: 'add:' of the class: Set. This method, when used, adds a new object into an instance of the class Set. Another example of such method is the method 'addLast:' of the class: OrderedCollection, which adds a new object to the end of an OrderedCollection.

The '**accessing**' represents methods in all Collection classes used to retrieve or replace elements; access various parameters of Collections. The method: 'at:put:' of the Class 'Array' is such an example. This method uses a new object to replace an existing elements of an Array which is at a particular position.

The '**copying**' represents methods in all Collection classes used to make copies of Collections. For example, the method: 'copyFrom:to:' of an OrderedCollection which copies all elements of an OrderedCollection which are between two indices.

The '**removing**' represents methods in all Collection classes used to delete existing elements in Collections. An example of this type of method is the method: 'remove:' of the class Bag, which removes an existing element in a Bag. The method 'removeAllSuchThat:' of the class SortedCollection removes all elements of a SortedCollection which satisfy certain user-specified condition.

The '**enumerating**' represents methods in all Collection classes used to sequence through all elements in Collections to carry out certain

* 'Collections' means instances of Collection classes.

computations. For example, the method: 'do:' is supported in all Collection classes, which is used to do looping. The method: 'findFirst:' in an OrderedCollection is used to find the index of the first element in the collection which satisfies a user-specified condition.

In Smalltalk, methods in each class are also classified into method categories. A class has several method categories, and each method category represents a group of functionally similar methods in the class. In effect, BRRR1 method categories are directly derived from the method categories in Smalltalk but with an important difference. For each BRRR1 method category, we examined each Collection class and classified into the BRRR1 method category all methods in the class, which belong to the method category whose name is identical to the name of the BRRR1 method category. An example should make this clearer. In Smalltalk, the class OrderedCollection has a method category 'adding', which represents all methods in *an* OrderedCollection which are used to add new objects. The class Set also has a method category 'adding', which represents all methods in a Set which are used to add new objects. Similarly, some other Collection classes such as 'SortedCollection', 'Bag' also have the method category: 'adding' and all represent the methods in these classes which are used to add new objects. In BRRR1, all these 'adding' methods in *different* Collection classes thus are classified into the BRRR1 method category: 'adding'. Although the 'adding' methods in different classes have different ways of doing the addition, they do have the common function: 'add new objects into a collection'. It is therefore appropriate to classify them into a 'global' category representing the same meaning — 'adding', and in which methods from different classes are gathered together. (see figure 3.10 for an illustration).

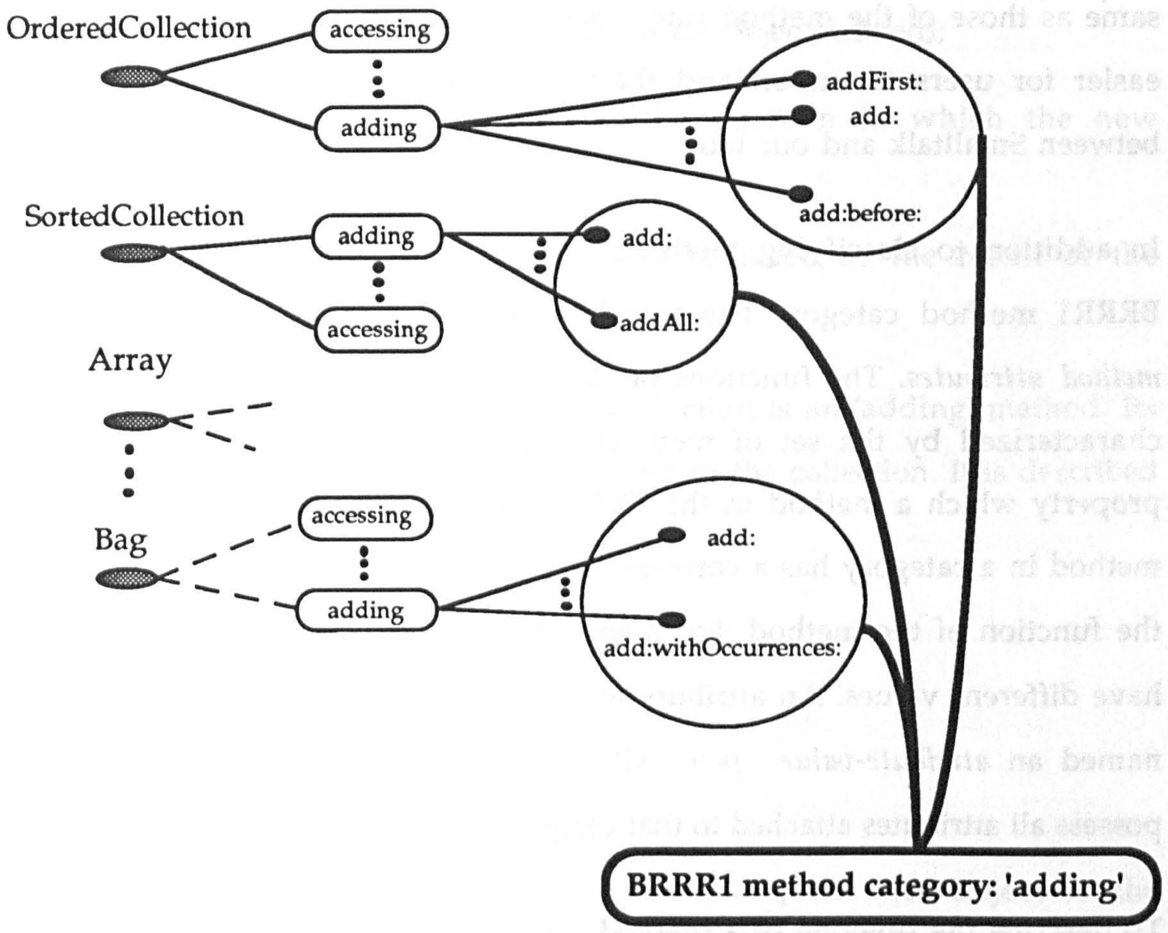


Figure 3.10. An illustration of BRRR1's method organization. Included in each circle are the methods in a class. The methods in individual classes are grouped together into a 'BRRR1 method category' which has the same name as the method category in individual classes.

In BRRR1, methods in other categories are also classified in the same way, i.e. all methods in 'accessing' ('copying'; 'removing' and 'enumerating') category in different Collection classes are put into the corresponding BRRR1 method category: 'accessing' ('copying'; 'removing'; enumerating'). This classification method provides a basis for users to find information across different classes through a method query. For example, if a user wants to find a method which performs a special 'add' operation, as it will be shown later, he/she can go to the BRRR1 method category: 'adding' and find the method through a method retrieval, which retrieves the method from all methods in different classes which perform the 'add' operation. This would save him/her from going to each class's 'adding' category and browse methods in each class. In addition, the names of the BRRR1 method categories are the same as those of the method categories in Smalltalk. It therefore should be easier for users to understand their meanings as there is a consistency between Smalltalk and our tool.

In addition to classifying methods into BRRR1 method categories, each BRRR1 method category has associated with it a set of attributes named *method attributes*. The functions of the methods in the method category is characterized by the set of method attributes. Each attribute specifies a property which a method in the BRRR1 method category must have. Each method in a category has a corresponding value to an attribute according to the function of the method. For a specific attribute, different methods may have different values. An attribute together with the corresponding value is named an *attribute-value pair*. All methods in a method category must possess all attributes attached to that category.

To describe the function of a method, two basic aspects to consider are: the operation that the method performs; and the objects manipulated by the method (for the objects manipulated, factors to consider are: the number of objects involved; constrains to the objects and the objects returned, etc.). We therefore used the following criteria to determine the attribute set for each BRRR1 method category:

i) The operation a method performs;

ii) The objects a method manipulates;

iii) The positions in which the objects are manipulated in a Collection;

iv) The possible constraints to the objects manipulated by the method;

v) The objects returned as a result of the operation.

For example, for the category: 'adding' which represents all methods used to put new objects into a collection, the attributes are:

Operation: (represents the 'add' operation);

Objects-added: (represents the number of the objects added);

Position-in-the-receiver: (represents the position in which the new objects should be put);

Object-returned: (represents the object returned as the result of the operation).

The method 'add:' of the class OrderedCollection is an 'adding' method. Its function is to add an new object into the end of the collection. It is described as:

Operation: add;

Objects-added: one-object;

Position-in-the-receiver: end;

Object-returned: newObject. (note that, here, the newObject is the argument of the method 'add:' and represents the object to be added).

Another method 'addAllFirst:' of the same collection, whose function is to add a group of objects into the end of the collection, can be described as:

Operation: add;

Objects-added: a-collection-of-objects;

Position-in-the-receiver: beginning;

Object-returned: anOrderedCollection.

Taking the method category 'accessing' as another example, the attributes are:

Operation: (represents the accessing operation);

Objects/Parameters-accessed: (represents the objects to be accessed);

Position-in-the-receiver: (represents the position of the objects to be accessed);

Object-returned: (represents the object returned as the result of the operation).

The method 'at:put:' of the class Array is a method of this category, whose function is to replace the element of an Array which is at the position specified by an integer with a new object. Its description is:

Operation: replace;

Objects/Parameters-accessed: one-element;

Position-in-the-receiver: position-indexed-by-an-integer;

Object-returned: newObject (note: the 'newObject' is the argument of the method).

3.4.3 Summary of component organization

In this section, we have described the component organization in BRRR1's library. Components are principally organized into two levels: the class level and the method level. At the class level, classes are indexed by one or several class descriptors. Each class also knows the method categories it contains and what methods are in each method category. At the method level, methods from all Collection classes are classified into different BRRR1 method categories. The BRRR1 method categories use the same names as those in individual classes. Each BRRR1 method category has associated with it a set of attributes which characterize the functions of the methods in the category. For each method, it possesses the attributes of the category and has corresponding values to the attributes according to the function of the method. In the following section, the interface of BRRR1 will be described.

3.5 The interface of BRRR1

The interface of BRRR1 consists of two windows: the Main window and the Method Examination window. We first introduce the Main window.

3.5.1 The Main window

The following five panes constitute the Main window, we describe them in turn:

a) Class Descriptors pane;

b) Class Query pane;

c) Method Query pane;

d) Matched Items pane;

e) Example pane.

a) **The Class Descriptor** pane contains all class descriptors used to index the Collection classes stored in BRRR1's library. Users start their initial query for classes from this pane. They can select any of the descriptors in this pane and manipulate it with the menu commands in this pane to construct a class query. There are two menu commands in this pane: **Require** and **Prohibit.**

Require: sends the selected class descriptor to the Class Query pane. This specifies that the classes sought must have the property characterized by the selected class descriptor.

Prohibit: sends the negated form of the selected class descriptor to the Class Query pane (i.e. prefix the descriptor with: 'not--'). It specifies that the classes sought must not have the property characterized by the selected class descriptor.

b) **The Class Query pane** holds users' query for retrieving classes. A class query consists of a set of class descriptors. Some of the descriptors may be in negated form. The descriptors are sent to this pane either from the Class Descriptors pane or from the Example pane which we will explain later. All class descriptors are connected implicitly by the logic operator: 'and'. It specifies that, for a class to match the query, all conditions expressed by the class descriptors must be satisfied. Take as an example the following query in this pane:

elements-ordered;

accessible-by-a-key;

not--abstract-class.

This query specifies that the classes sought must be ordered, [and] elements in the instances of the classes must be accessible by an index, [and] must not be abstract classes.

Currently, BRRR1 can only process the queries connected by the operator 'and', it cannot process the queries connected by another logic operator 'or'.

There are two menu commands in this pane: **Reset** and **Retrieve classes.**

Reset: resets the whole system. This includes clearing all descriptors in both the Class Query and the Method Query pane.

Retrieve classes: retrieves all classes which match the query in this pane and presents the result in the Matched Items pane. The Example pane is also updated to show an example. For a class to match a query, the class must satisfy all conditions expressed by the class descriptors which are 'required'; meanwhile, it must not satisfy any of the conditions represented by the class descriptors which are 'prohibited'.

The description of the Method Query pane is delayed until we have described how users examine methods after the initial class query.

c) **The Matched Items pane** displays a list of classes or methods which match users' query for classes or methods. The number of matched items is displayed in the label of this pane. If users are doing a class retrieval, i.e. they use the menu command: '**Retrieve classes**' in the Class Query pane, then displayed in the Matched Items pane is a list of classes which match the class query shown in the Class Query pane. Otherwise, if users are carrying out a method retrieval, i.e. they use the menu command: 'Retrieve methods' in the Method Query pane, then shown in this pane is a list of methods which match the method query displayed in the Matched Query pane.

In the case that matched methods are shown in this pane, for each method presented in this pane, the text in the bracket following the name of a method tells the class to which this method is associated. For example, if a method is shown as: 'at:put: (Array)', then the name (selector) of the method itself is: 'at:put:' and it is in the class: Array. In this way, when users get a group of matched methods, they can recognize the class of each method.

Users can select any item in this pane to examine. Once selected, the corresponding description of the selected item would be presented in the Example Item pane to be described next. This gives BRRR1 a browsing capacity, users can query and then browse to examine the matched items.

d) **The Example Item pane** is used to display a description of an item which matches a query (either a class or a method query). Users may use the information shown in this pane to reformulate the initial query. Corresponding to the situations we mentioned in the last paragraph, if users are doing a class retrieval, then shown here is the description of an example class. If a method retrieval is being performed, then displayed here would be the description of an example method. The description of an example class comprises a text message and a set of class descriptors used to index it. The class descriptors shown in the description of the example class are those which index the class. For example, the descriptors used to index the class OrderedCollection are:

elements-ordered

order-determined-externally

accessible-by-a-key.

Thus, all of them would be displayed as a part of the description of this class. Users can directly select any of the class descriptors to construct a class query (with the menu commands: 'Require' and 'Prohibit', which are similar as those in the Class Descriptors pane).

In BRRR1, users create a class query by selecting class descriptors either from the Class Descriptors pane or from the Example pane. A possible problem with selecting descriptors from the Class Descriptors pane is that the users may not fully understand the meaning of the descriptors there. Consequently they may encounter difficulties in deciding which one to choose. In the Example pane, with the set of class descriptors appears a comment which explains the function of the example class. This comment serves as a context to the use of the descriptors. It thus should help users comprehend better the meaning of the descriptors since the descriptors essentially express the meaning of the comment, i.e. they all describe the function of the component. This should facilitate users' selection of the descriptors to reformulate the initial query.

The menu commands are different when an example class is presented in the pane from those when an example method is presented. When an example method is displayed in this pane, the menu commands displayed would be fully described after we have introduced the Method Examination window. When a class is shown, the menu commands are:

Require: the same as that in the Class Descriptors pane.

Prohibit: the same as that in the Class Descriptors pane.

View method categories: updates the Example pane and attaches the method category of the example class to the original display contents. We said in section 3.3 that after several class retrievals, users may want to examine methods in the example class to further inspect its function. They use this command to find all method categories of the example class.

After the users see the method categories, if they want to investigate the methods in a particular method category further, they can select the category and use the command **Specialize** described below.

Specialize: opens a new window — the Method Examination window. Once that window is open, users can then examine in that window the methods in the selected method category of the example class; or they may start a method query from that window. We will describe the Method Examination window in the next section.

3.5.2 The Method Examination window

This window is open when users have selected a method category of an example class and used the **Specialize** command. Shown in the top pane of this window are methods of the example class that are in the selected method category. The bottom pane contains the description of the method selected by the users in the top pane (see figure 3.4 which appeared in section 3.2.2). The selected method is used here as an example method. The description of an example method consists of a comment (the plain text part) which explains the function of the method, and a set of method descriptors (or the attribute-value pairs). The users view this window to inspect methods which are in the class in the Example pane; and which belong to the selected method category. They can select any method shown in the top pane and its description is presented in the bottom pane.

a) Inherited methods are also shown in this window

One point to note is that the methods presented to the users include not only those which are explicitly defined in the example class, but also the methods which are inherited from some of its superclasses; and which are in the same named method category in those superclasses. The superclasses from which the example class inherits methods are such classes that they are in the Collection class hierarchy and that they lie between the top class of the Collection classes hierarchy, i.e. the 'Collection' class and the example class along the inheritance path. An example should make this clearer. If the example class presented to a user is the class 'OrderedCollection' and the user has selected the method category 'adding' to examine the methods, then in the corresponding Method Examination window, the methods shown include the following parts:

i) The 'adding' methods explicitly defined in the 'OrderedCollection'.

ii) The 'adding' methods inherited from its superclasses which are between the 'Collection' class and the example class, i.e. the 'OrderedCollection'. The superclasses of 'OrderedCollection' which are between 'Collection' and 'OrderedCollection' along the inheritance path are classes: 'Collection' and 'SequenceableCollection', thus the 'adding' methods of 'Collection' as well as the 'adding' methods of 'SequenceableCollection' are also shown in the Method Examination window (see figure 3.9 in section 3.4.2 for a part of the hierarchy of the Collection classes).

The reason for showing users the inherited methods of a class is that those methods are part of the interface of the instances of the class. An instance of a class can not only perform computations with the methods defined explicitly in the class, but also it uses the methods inherited from its superclasses. The function of an object therefore is determined not only by the methods defined in its class, but also by the methods it inherited from its superclasses. The methods inherited from its superclasses thus form an integral part of an object's interface. Therefore, for users who are examining the function of a class, it is necessary to show them all the methods which the class has to give the users an entire image of the class. The methods inherited from its superclasses can however be numerous. In particular, the superclass of the class 'Collection' is the class 'Object' which holds methods common to all classes in Smalltalk. If all methods inherited from all superclasses are presented to users, they might be overwhelmed by the amount of information. It is therefore necessary to show only those methods which are most relevant. A design decision we made was to include the methods inherited from such classes that are in the class hierarchy of the Collection classes, but all methods in the class 'Object' are excluded. This is because the class 'Collection' is an abstract class, which is at the top of the inheritance hierarchy of all Collection classes. It holds the message protocols which are specific to all Collection classes. Therefore, methods from the 'Collection' class and the classes below it are more relevant to a collection class and should be presented to users. The protocols in the class 'Object' are for all classes in Smalltalk, it is not particular for Collection classes, therefore methods inherited from 'Object' are not shown to users.

b) Starting a method query in this window

It is likely that the methods satisfying users' requirements cannot be found in the Method Examination window. This is because the class the user is examining may not contain the required methods. After several cycles of class retrieval, there are usually a group of classes matching the class query. The methods required may be in other classes which match the class query. More particularly, they may be in the classes where the method category containing them has the same name as the one the user selected in the example class. In this situation, the user need not, as they have to in the original Smalltalk system, go to each class to search the methods in that class. In BRRR1, they can construct a method query to retrieve methods across all matched classes. A method query allows them to retrieve the methods which are in those matched classes, and which are represented there by the method category whose name is the same as that which the user is examining. In section 3.4.2, we showed that in BRRR1, methods from different classes are classified into BRRR1 method categories. Each BRRR1 method category represents a group of similar methods which are from different classes. This provides the base for querying methods across different classes. The users may initially be interested in the methods represented by a method category (this is reflected by the fact that they are inspecting the methods in this category), and which are only in one class. They may need however, to find the methods which are of the same kind, but are in other classes so they can query all methods of this kind to get the necessary ones. As the methods in the Method Examination window are part of all methods of this kind, therefore, the properties of these methods can be selected and used to construct an initial method query from this window. Thus, in effect, the method shown in the bottom pane of the Method Examination window serves as an example method.

A method query is created in the similar way as that in which a class query is created. The users select the method descriptors (the attribute-value pairs) and use the menu commands in this pane to send them or the negated form of them to the Method Query pane in the Main window. The menu commands for this pane are:

Require: sends the selected method descriptor to the Method Query pane in the Main window. It specifies that the methods sought must have the attribute and the value to the attribute must be the same as the selected one.

Prohibit: sends the negated form of the selected descriptor (i.e. prefix it with 'not--') to the Method Query pane in the Main window. It specifies that the methods sought must not have the same value as the selected one.

Alternative: shows in a pop up menu all values BRRR1 knows for the selected attribute. During a method retrieval, the value of an attribute given in the description of the example method may not satisfy the user's requirement. The user may want to find a more appropriate value to specify the property of the required methods and this is where the 'alternative' command is useful. BRRR1 will search to find other values for the selected attribute among methods which satisfying the following conditions:

— they are in the method category to which the example method belongs;

— the classes of the methods match the current class query constructed in the Class Query pane.

When the alternative values are shown in the menu, users can select any one and 'require' or 'prohibit' it. This value or its negated form would be sent to the Method Query pane in the Main window.

After users have constructed an initial method query from the Method Examination window, they can go back to the Method Query pane in the Main window and start a retrieval for methods from there. Next, we go back to the Main window and introduce the Method Query pane.

The Method Query pane holds the users' method query. Similar to the class query, all method descriptors in this pane are implicitly connected by the operator 'and'. The only menu command for this pane is '**Retrieve methods**'. This would retrieve all methods which satisfy the query. In order to match a query, a method must satisfy the following conditions:

i) The method must belong to the selected method category.

ii) For all 'required' attribute-value pairs in the query, the method must have those attributes and corresponding values must be the same as those in the query.

iii) For all 'prohibited' attribute-value pairs in the query, the values of the attributes must not be the same as those in the query.

In addition, the methods to be retrieved are constrained by the class query the users created before the method query is composed. A method query should be regarded as an extension of the current class query. In BRRR1, users start a query from the class level. They first create a class query by using the class descriptors. After several cycles of class retrieval, they examine the methods in matched class and then, if necessary, create a method query to find needed methods. Therefore, the methods to be retrieved should also satisfy the conditions the users have specified in the class query so far, i.e. the classes with which the methods associated must match the current class query. For this reason, in response to a method query, only those methods whose classes satisfy the class query shown in the Class Query pane, and the methods themselves satisfy the method query are returned as the result.

When a method retrieval is carried out by BRRR1, the Matched Items pane is updated and methods matching the method query are displayed there. Meanwhile, the Example pane changes and a description of the method highlighted in the Matched Items pane is shown there. This method is used as an example instance which matches the method query. The description of the example method is similar to that in the Method Examination window we mentioned earlier. It comprises a comment explaining the function of the method and a set of method descriptors. If a user is not satisfied with the retrieval result, he/she can manipulate the method descriptors with menu commands in this pane to reformulate the partial method query and perform further retrievals. The menu commands of this pane are:

Require;

Prohibit;

Alternative.

These are the same as those in the Method Examination window. This ensures that users can continue method queries from this pane.

Another command associated with this pane is: '**View classes**'. This replaces the matched methods shown in the Matched Items pane with the original classes which matched the class query. When users are querying the methods, they sometimes want to see what the classes they retrieved originally are. With this command, they can see the matched class list. If they want to come back to view the method, they can use the 'retrieve methods' command in the Method Query pane.

So far, we have described the design of BRRR1, in the next section, its implementation is introduced briefly.

3.6 The implementation of BRRR1 — an overview

BRRR1 is implemented in Smalltalk-80 on a Macintosh IIcx, with about 80k source code. The system has two types of components: a database component and an interface component corresponding to BRRR1's software component library and interface respectively. These components are described in turn below.

3.6.1 The database component

The database component contains information about classes and methods, it also contains a number of search methods used to access the information. This component is implemented as an instance of a class: Brrr1Organization which is a subclass of the existing class: Model. The reason for choosing it as a subclass of Model is that this class is also used as a 'model' for several interface classes which need to access it to get the necessary information and display it in the corresponding panes. The interface classes and their interactions with the database will be described later in this section.

There are a number of tables in Brrr1Organization, the two most important of which are : ClassInformationTable and MethodInformationTable.

The ClassInformationTable contains the information about each class, the main items of which are:

class names;

class descriptors which index each class;

class comments which describes the function of each class;

method categories contained by each class;

method names associated with each class.

The MethodInformationTable contains the information about the methods, the main items are as follows:

method names;

the method category to which each method belongs;

the class with which each method is associated;

method comments which describe the function of each method;

the method descriptors with which each method is indexed.

These tables are implemented as instances of the class Dictionary to facilitate the search process. There is also a group of search methods for accessing the information stored in those tables.

3.6.2 The interface component

The interface of BRRR1 is implemented based on the 'Model-View-Controller' (MVC) paradigm of Smalltalk. Before we introduce the components themselves, we briefly introduce the paradigm first. The 'Model-View-Controller' paradigm is used in Smalltalk to implement user interfaces. Any window on the Smalltalk screen has three essential components associated with it, the *model, view,* and *controller* which are defined as follows:

A *model* is an object which represents the data to be displayed.

A *view* is an object which displays aspects of the model; it deals with everything graphical; it requests data from its model and displays the data.

A *controller* is an object which is used to send messages to the model, and provide the interface between the model with its associated views and the interactive user interface devices (e.g. keyboard, mouse).

Each view may be thought of as being closely associated with a controller, each having exactly one model, but a model may have many view/controller pairs. In the Smalltalk system, numerous classes exist which can be used to build a graphical interface. BRRR1's interface is implemented based on those classes.

As mentioned earlier in this chapter, BRRR1's interface has two main windows: the Main window and the Method Examination window. We describe the implementation of the Main window first. This window has five panes, and each of them is implemented with a 'model', a 'view' and a 'controller' which are instances of appropriate model, view and controller classes. We summarize the components used to implement each pane below.

a) Class Descriptor pane:

The 'model' of this pane is an instance of the class: ClassDescriptors, which is designed as a subclass of an existing class: TextHolder. This new class provides a method to get the class descriptors from the database component. The 'view' object of this pane then gets the descriptors from the model and displays them on the screen. The 'view' object is an instance of an existing class: TextCollectorView which provides a method to display text. The 'controller' of this pane is an instance of a newly added class: ClassDescriptorsController, which is a subclass of an existing class: CodeController. In ClassDescriptorsController, a new set of menu messages is defined to generate the menu for this pane.

b) Class Query pane:

A new class: Query is designed to create the 'model' of this pane. It is a subclass of an existing class: TextCollector. Query provides functions to store and for 'view' object to display the class descriptors selected by users in either the Class Descriptors pane or the Example pane. The 'view' of this pane is an instance of an existing class: TextCollectorView. The controller is an instance of a newly defined class: QueryController which is a subclass of an existing class: TextCollectorController. QueryController provides a new set of menu messages to generate the menu of this pane. It also has methods to parse and validate the descriptors selected by users.

c) Method Query pane:

This pane is similar to the Class Query pane, thus its 'model' and 'view' are instances of the classes: Query and TextCollectorView. However, a new class MethodQueryController is defined to implement the 'controller' of this pane. MethodQueryController is a subclass of the class: QueryController mentioned above and in this class a set of menu messages is defined to generate the menu for this pane.

d) Matched Items pane:

The 'model' of this pane is the database component of BRRR1. Its 'view' is an instance of the existing class: SelectionInListView and its 'controller' an instance of an existing class: SelectionInListViewController. The methods necessary for generating the list for displaying and the methods for sending changes are defined in the class of the 'model' object, i.e. the class Brrr1Organization.

e) Example pane:

The 'model' of this pane is the database component of BRRR1. Its 'view' is an instance of the existing class: CodeView. A special controller class: ExampleWindowController is implemented for creating a 'controller' of this pane. This new class is a subclass of an existing class: CodeController and defined in it is a set of menu messages. Again, the methods necessary for getting the appropriate text information to display are implemented in its 'model' class — Brrr1Organization.

The other window of BRRR1's interface: the Method Examination window, has two panes: a list pane which shows a number of methods and an example pane which shows text. It's implementation is similar to the Main window, thus we will not describe it further here.

3.7 Summary

In this chapter, we described BRRR1— the first version of a prototype system designed to help non-expert users find reusable components in Smalltalk. The design principle of this tool is based on the 'retrieval by reformulation' paradigm. BRRR1 consists of two parts: the component library and the interface. In BRRR1, classes are indexed by a group of class descriptors, and methods are classified into BRRR1 method categories. In addition, methods in each BRRR1 method category are characterized by a set of attribute-value pairs. BRRR1's interface consists of two windows: a Main window and a Method Examination window. Users use menu commands in panes of the windows to create a query. A search of the required component in BRRR1 is completed by reformulating a query. Users first create an initial class query by using the class descriptors provided by the system. They may then use the information presented by the system in responding to the initial query to reformulate the initial class query. After certain cycles of class retrieval, the users may examine the methods in matched classes to choose the required classes. If necessary, the users can further extend the search process by creating a method query. In a method query process, the users again use the information provided by BRRR1 to reformulate an initial method query and retrieve wanted methods. The whole retrieval process may be repeated until the users get satisfactory results. BRRR1 is implemented in Smalltalk-80 and the MVC paradigm is used in the implementation of its interface. In the next chapter, we report an empirical study conducted on BRRR1 to test its effectiveness in helping users find reusable components.

Chapter 4 An empirical evaluation of BRRR1

In this chapter, we describe an empirical evaluation of the implementation of our system BRRR1. The purpose of this evaluation was to test its effectiveness and more importantly to find out the possible problems users might have in using it. The chapter is structured as follows: We first outline the organisation of the study, then present and discuss its findings.

4.1 The organisation of the study

The intention of this study was to use the findings of the evaluation as a basis for another iteration in the development of the system. In this formative evaluation, we used four subjects with varying levels of Smalltalk experience.

4.1.1 The subjects

Of the four subjects, one was an expert Smalltalk programmer; one was an experienced Lisp programmer and an intermediate Smalltalk user; the remaining two were not very familiar with Smalltalk though they had a considerable amount of experience in other programming languages and understood the basic concepts of object-oriented programming. None of the subjects had any experience of using BRRR1. The purpose of this study was to increase the level of feedback which would serve to improve the design of the system. It was with this in mind that one expert user was used as a subject of this study although BRRR1 is intended to help non-expert Smalltalk users. As this subject was experienced in designing systems in Smalltalk, it was felt that his opinion and suggestions about the design of BRRR1 would be helpful for future improvement.

4.1.2 The tasks

There were a total of six tasks which the users were requested to complete. The tasks were designed to represent the typical situations where a user is looking for some collection classes (or methods in these collections) to complete some programming tasks. The tasks are of the pattern: 'find a class with certain properties and which has a particular method that can perform certain functions'. The tasks were as follows:

1) A collection has 6 numbers (3 7 5 8 9 100) as elements. One of the collection's methods will enable you to replace the 2nd, 4th and 6th element with the number 20. Find this method.

2) A collection has 20 numbers as its elements. One of the collection's methods can be used to delete all numbers whose values are less than 10 in the collection. Find this method.

3) There is a group of ten numbers stored in a collection. A method of this collection, when used, will give you the index of the first element which is greater than 5. Find this method.

4) A method of a collection can be used to append a list to another list.

For example, with this method, list1=(1 10 'john') can be appended to list2=(2 'you' 'parents' 5 'children') and the result is a new list:

(2 'you' 'parents' 5 'children' 1 10 'john'). Find this method.

5) A collection — collection1 is as follows: (1 3 5 'john' 'simon' $p 10), one of its methods can be used to produce another new collection — collection2 which is similar to the collection1, but its elements are:

(5 'john' 'simon' $p). Find this method.

6) Find a method such that it can be used to put numbers into the collection to which this method belongs and all these numbers put in will be ordered automatically according to their values.

4.1.3 The procedure

The study consisted of two sessions: a training session and a test session. The training session started with a demonstration provided by the experimenter. In the demonstration, the experimenter briefly introduced the user the purpose and design principles of BRRR1 and showed the user how the system works, i.e. how components (classes and methods) are retrieved and how a query should be constructed. The demonstration lasted about fifteen minutes. After the demonstration, the user was presented a set of six exercise tasks to familiarise themselves with the system. The exercise tasks can be found in Appendix A of this thesis. These preparatory exercise tasks were designed to have a similar format to the tasks to be completed by the user in the test session. During the exercise session, the user could ask the experimenter questions about the system and request help from him, if necessary, to complete the exercises. This session took about twenty five minutes to complete. The actual test session started after the user had finished the training session. At this point the user was presented with the tasks to complete using BRRR1. After each task was finished, the user was asked to write his result on an answer sheet provided. While each user was completing tasks, his interaction with BRRR1 was videotaped. Meanwhile, the user was asked to talk aloud about his actions and his verbal protocol was recorded. After a user had completed all tasks, the experimenter asked him for comments on the system design and for suggestions for improvements. The training and test sessions together took about one and an half hours per user.

4.2 Results and discussion

The data collected from users was analysed in the following way: based on the video tape, the steps a user carried out in completing each task were identified. A step was defined as each instance when a user used a menu command of BRRR1 or selected an item (i.e. class or method) to examine. Additionally, the hypothesis behind the adoption of each step were approximately identified based on the user's verbal protocol collected while he was completing that task. An example of the analysis of data collected from user B during his completion of task three is listed below. The protocol shown here is a typical one, and the user is unfamiliar with Smalltalk but is experienced in other programming languages. In this analysis, the plain style text after the word '**Trans:**' (abbreviation for 'Transcript') is the user's transcript. The boldfaced text after '**Hyp:**' (abbreviation for 'Hypothesis') shows the hypothesis behind the user's step. The italic text in square brackets preceded by the word: 'note:' is the analyser's note about the user's activity between the steps. Two asterisks mark places where errors occurred, i.e the user hypothesizes wrongly. The underlined words are the BRRR1 menu commands which the user selected in his interaction with the system.

<User B's protocol for completing task three>

Steps	**Transcript and hypothesis**
	Trans: (What in the task is) a group of numbers, well, I suppose (I can choose) 'elements-ordered'. **Hyp: As elements are numbers, so they are ordered.** ***[note: wrong, He thinks the order is according to the values of the elements, while the real meaning is that the order is on the positions].*
1. Require 'elements-ordered'.	

Trans: I wonder if I ... En, yes, I don't know anything else to start (a retrieval), I am not really sure I can start from these class descriptors or I can start in some other way. ... So I am going to retrieve a class.

Hyp: Retrieves classes.

2. <u>Retrieve classes.</u>

[note: got the class: 'OrderedCollection' presented by the system as the example class.]

Trans: What's the question again? I need to find first element satisfying some predicates. Let's have a look at the OrderedCollection.

[note: he looks at the OrderedCollection].

Hyp: Wants to find the first element satisfying some predicates. First looks at the class 'OrderedCollection'.

3. <u>View method categories</u>.

[note: looks at all method categories].

4. <u>Specialize</u> the method category: 'accessing'.

*******[note: he selected the wrong method category, but I am not sure why.]*

[note: browses all methods]

5. Selects the method: 'indexOfSubcollection: startingAt:ifAbsent:'.

Trans: Well, I saw there is an 'indexOfSubcollection:startingAt:'. Is that the subcollection I want or is that a particular one...? *[note: reads the task again]*. The first that greater than 5. Right, not an index of a subcollection I wanted. Oh, it could be .... if I scroll here (i.e. scroll the screen).

[note: browses methods]

It will be more helpful to me if it tells me something about what the types of the parameters (of the methods) were.

[note: browses methods].

Well, 'indexOf:ifAbsent'. That is a possibility too.

Hyp: Checks the method 'indexOf:ifAbsent:'.

6. Selects the method: 'indexOf:ifAbsent:'.

[note: reads the method comment]

Trans: It is not really a helpful description. I don't understand that. *[note: keeps reading]* ... Oh, I see (the meaning of it). No, that (method) is looking (at) a particular element.

[note: keeps browsing methods].

None of them (i.e. the methods browsed) seems the correct answer. ...

Oh, 'indexOf:', no, that will be given an element. I want something (that) takes predicates. ... Let's try that (i.e. the method: 'indexOfSubcollection: startingAt:ifAbsent:').

Hyp: Wants something which takes a predicate.

7. Selects the method: 'indexOfSubcollection: startingAt:ifAbsent:'.

[note: reads the method comment]

Trans: 'Answer the index of the receiver's first element such that that element equals the first element of subcollection'. En, I can do it like that way. I could have ... . No, I could not, because they need (the element) greater than 5. If it is integers, I suppose you can, see .... No, I got to find something that takes boolean predicates.

Hyp: Needs to find something which takes boolean predicates.

8. Selects the method: 'atAll:put:'.

[note: browses methods]

Trans: Let's go back here.

Oh, (the method) 'first', that might be.

9. Selects the method 'first'.

[note: reads the method comment]

Trans: No, I want 'first such that'.

Hyp: Wants a method like: 'firstSuchThat:'.

[note: goes back to the Main window and browses classes].

10. Selects the class: SortedCollection.

11. View method categories.

[note: looks at the method category 'accessing']

Trans: Again, we are talking about the 'accessing'. Is there anything interesting down there? (i.e. the class descriptors of SortedCollection).

[note: looks at 'accessing']

I want to access elements ... Do I want to access? Is 'accessing' the same as finding? ... Suppose that is 'accessing'.

Hyp: Want to access elements to find the required one.

***[note: here, he wanted to access elements and find the index. However, no methods in this category can find the index satisfying a boolean predicate.]*

12 Specialize the method category: 'accessing'.

[note: browses methods]

Trans: (I am looking at the) 'accessing' methods in SortedCollection. ...

'indexOfSubcollection...' (That would) turn back before. I couldn't be convinced. The index could not guarantee equals the first element. No. ... En, ... Interval.

[note: goes back to the Main window to look at classes.]

13. Selects the class: 'Interval'.

Trans: 'Interval' are numbers, (it is) no good.

14. Select the class: 'OrderedCollection'.

[note: goes back to the main window's class descriptors pane and read class descriptors].

Trans: I've got 'elements-ordered'. (Should I choose the class descriptor) 'accessible-by-a-key'? No.

[note: looks at the descriptors: 'order-determined-internally' and 'order-determined-externally'].

I don't know what 'externally', 'internally' mean.

***[note: difficulties in understanding these two class descriptors].*

15. Retrieve classes.

[note: browses classes]

Trans: It must be one of these (classes), because these are only ones with ordered elements... I am thinking of any sensible class which could have something which says 'finds something that satisfies some boolean predicates'.

Hyp: Tries to find a class which could have something which says 'finds something that satisfies some boolean predicates'.

[note: as he had spent a long time on the wrong category, the experimenter had to prompt him to look at other method categories].

16. Selects the class: 'OrderedCollection'

17. Specialize the method category 'accessing'.

[note: then closes the Method Examination window for the 'accessing' methods].

Trans: I am muddled about whether (I should choose the) 'accessing' things. Things about 'accessing' (is), whether it always gives you the elements or just gives you the positions of them.

[note: checks other method categories]

... Enumerating,.. it might be the case, enumerate a subcollection. Oh.

Hyp: Not sure if the 'accessing' methods would give elements or just the index of the elements. It might be in 'enumerating' category, since one can enumerate a subcollection.

18. Specialize the method category 'enumerating'.

[note: browses methods]

Trans: (The method) 'detect:', that looks plausible.

Hyp: It seems the method 'detect:' plausible.

19. Selects the method: 'detect:'.

[note: reads the comment]

Trans: Ah, got it. Yes, so, that's the answer. So I suppose I can just choose that and do a proper query.

Hyp: Should be the method 'detect:', so query similar methods.

20. Require the method descriptor 'Operation: select'.

21. Retrieve methods.

[note: browses matched methods]

22. Selects the method: 'detect:ifAbsent:'.

Trans: So, basically, (in the matched method list) you got 'detect:'. Detects a block and exception block. And they all detect. (There are 44) matched methods.

[note: browses methods again and finds the method 'findFirst:' to inspect].

En, 'findFirst:', rather then 'detect:'.

Hyp: Should be the method 'findFirst:' rather than the method 'detect:'.

23. Selects the method: 'findFirst:'.

Trans: It's 'findFirst:' we want. En, so, I ought to be able to type up the query. Right.

24. Require descriptors and retrieve methods.

Finished

[note: He made a comment on his performance on the task: 'Why did I do it so slowly? I think it's the notion of 'enumerating'. The difference between the 'enumerating' and 'accessing' I don't think has been clear to me']

From this user's protocol, the following problems could be identified:

First of all, he misunderstood the meaning of the class descriptor 'elements-ordered' as: elements are ordered according to their values rather than their positions. This can be found from the protocol before step 1. In effect, this descriptor is intended to represent: 'elements in a collection should be ordered as a sequence'.

Secondly, he selected the method category 'accessing' (in step 12) to look for methods which can find an index satisfying a boolean predicate. He thought that he needed to access the elements of a collection to find the index. Actually, the required method 'findFirst:' is classified into the category: 'enumerating', since this method uses the algorithm: enumerate all elements of a collection to select a specific index. This problem seems to represent a misunderstanding of the scope of the method category.

Finally, he was not sure the meaning of the class descriptors: 'order-determined-internally' and 'order-determined-externally'. This can be seen from the protocol before step 15.

Problems similar to those we have just presented were also encountered by other users in completing the tasks. These problems could be approximately classified into the following categories, which will be discussed in the following paragraphs:

a) Problems in understanding some class descriptors.

Three users either misunderstood the meaning of several class descriptors and created wrong queries by using them, or they explicitly stated that they were not sure about the meaning of those descriptors so that they did not use the descriptors in a query. For example, the class descriptor 'elements-ordered' was intended to mean that all elements in a collection are arranged into a sequence so that it is possible to refer to the ith or nth element of the collection. From the protocol we just showed above, it can be seen that User B misunderstood this as 'elements in a collection are ordered according to their values'.

The class descriptors were intended to be used to specify the functional features of the Collection classes and all of them are displayed in the Class

Descriptors pane. Additionally, one part of the description of each class is the descriptors which index the class. The other part of the class description: the comment which explains the function of the class, should provide a context about the usage of the descriptors. The design intention here was that if users understand them, then they select them directly from the Class Descriptors pane. When they have difficulty in understanding a descriptor, they can refer to an example class which has the descriptor embedded in the description to see how it is used. This did not work well in practice. The Class Descriptors pane provides no context or explanatory information about the exact meaning of the descriptors and additionally, the users normally start to create a class query by selecting items from this pane. Therefore, this might have caused some difficulties in comprehending the exact meaning of some descriptors. On the other hand, the context provided by class comments about how a class descriptor is used is not clear for several descriptors. Take as an example the descriptor 'elements-ordered', the comments of the classes which are indexed by this descriptor do not explain it very clearly. For example, the class OrderedCollection has the descriptor as an index. The comment of this class is:

> 'Class OrderedCollection represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. OrderedCollection can act as stacks or queues.'

Though this comment mentions the sequence, it does not state explicitly if the order of the collection is on the values of the elements or just on the positions of the elements. Therefore it may be hard for the users to infer the precise meaning of the descriptor from this comment.

In summary, some class descriptors were ambiguous and the comments of the classes did not provide sufficient information to serve as a satisfactory context to facilitate the users' comprehension to the descriptors. The users thus had difficulties in understanding correctly the meaning of the class descriptors. It therefore seems necessary to have an explanation mechanism, which on the request of the users, explicitly tells them about the meaning of the descriptors.

b) Misunderstanding of method categories.

This type of error happened when users started creating method queries. To create a method query, it was necessary for a user to first select a method category. Then from the example method provided by the system, the user may select attribute-value pairs to create a complete method query. However, every user made some mistakes in identifying the correct method category for creating the method query. They sometimes selected a wrong method category and then tried to create a method query without success. A typical example happened in task three. This task needs the users to find a method which would return the index of the first element in the collection whose value is greater than 5. The answer to this task is the method: 'findFirst:' which is in the method category: 'enumerating'. This is because, in Smalltalk, this kind of selection is completed by evaluating each element in the collection and examining it to see if it satisfies the condition set up by users, and then returning the result which fulfils the condition. However, all users initially selected the method category 'accessing' in the belief that they needed to access elements to find the index of the element satisfying the condition (i.e. value is greater than 5). They didn't realize that though the method: 'findFirst:' in essence has to 'access' the elements of the collection to select the index of a suitable element, it uses the algorithm: 'enumerate each element of the collection and examine if it satisfies the user-defined condition'. Therefore, this method is classified into the category 'enumerating' rather than the category 'accessing'. As BRRR1 was designed in such a way that if a method category was not selected correctly, it is impossible to get the right methods, the users could not find the required methods. Sometimes, the users kept examining a wrong method category several times and could not think of any other category to look at, until eventually, the experimenter had to provide hints to help them choose other categories.

This problem reflects a mismatch between the designer's intention and the users' understanding. In this respect, the classification of the system design was not well matched to the users' expectations. Thus it appears necessary to let users understand the scope of the method categories, and additionally to adapt some of the users' expectations about the method classification into the classification scheme so that they can find the required information more easily.

Here, it is necessary to add to our explanation of the way in which the experiment was conducted. We mentioned earlier that if users were really stuck on a method category, they received hints from the experimenter. In certain contexts intervention such as this from the experimenter might seem to compromise the precision of the data and should be avoided as much as possible. In our case, it seemed to be necessary to keep the test session going, since some users were 'hung up' as it were on one point for too long and could not make any progress. On the other hand, the difficulties of the users did show that there were some problems in certain parts of the system design and from this perspective, the result is informative and still valid for the purpose of this study, i.e. to use the findings for future improvements.

c) Problems arising from the lack of a 'history' mechanism.

While users were searching for a class or method, they sometimes were reminded of a class or a method they had met earlier and wanted to examine it again. However, they often couldn't remember exactly which one it was and did not know how to get it again. Another situation was that users often could not remember that they had examined a particular item (a class or method or method category). Consequently, sometimes they repeatedly inspected a method or method category and only after several repetitions they realized that they had visited it before. This suggests that it would be helpful if a 'history' mechanism were provided by the system which records the items at which a user has looked before. With this mechanism, whenever the user feels it necessary, he/she can see an item which has been examined previously. In addition, the user can see the history of his/her interaction with the system and thus is able to reason more effectively about his/her actions. This 'history' mechanism should reduce users' overload in using the system.

d) The tendency to browse rather than use a query (re)formulation method

From the users' interactions with BRRR1, it could be seen that on several occasions, when users were searching for a method, they deployed the browsing strategy instead of forming or reformulating a method query. They browsed the methods and then examined them to select the required ones. Although in BRRR1, methods from different classes are classified into BRRR1 method categories, users need only to browse methods in each method category and did not need to select each class and then browse methods there. Therefore, even just by browsing, it is easier for a user to find a method since he/she does not have to first select a class then explore the methods. However, BRRR1 was intended to provide the users with a query mechanism so that they could retrieve required components from large numbers of components by querying. The data collected shows that the users did not exploit the advantages of the system. This seems to indicate that the users were not very familiar with how to use the system. It might be due to that the training the users received in the training session was insufficient and so they could not make the best use of the facilities of the system. In the discussions with the users after the test, when users were asked why they did not reformulate a query, the three non-expert users admitted that they were not very familiar with the system, and sometimes forgot to do a query reformulation. In addition, as no written manual was provided, the users had nothing to which they could refer during the test, and so they used the browsing strategy to try and find the methods directly. Moreover, two of those users said that it was after they had finished all the tasks, that they really understood how to use the system. In the training session, each user was requested to complete a group of exercises whose format is similar to the test tasks. However, the difficulty level of the exercises is not as high as the test tasks, therefore, it did not achieve the expected effect.

In short, an important reason why the subjects did not use BRRR1 effectively seems to have been that the initial training provided was not adequate. In addition, it might have been more helpful to supply users with a written manual about how to use the system so that they could refer to it when they have questions.

4.3 Summary

In this chapter, we have reported on an empirical evaluation of BRRR1. This evaluation was formative, the intention being that the findings would provide the basis for further improvement. There were four types of problems identified in evaluating BRRR1:

Users had difficulties in understanding some class descriptors;

Users had problems in identifying an appropriate method category to create a method query;

It was found necessary to provide users some 'history' mechanism;

Finally, in terms of the study itself, sufficient training should be provided to users.

In the next chapter, we describe the design and implementation of BRRR2, an improved version of BRRR1 which was developed based on the findings in this evaluation.

Chapter 5 BRRR2 — an improved version of BRRR1

In this chapter, we describe the design and implementation of BRRR2. The changes and revisions in this version of the system are mainly based on the results of the empirical evaluation of BRRR1 discussed in chapter four. We first outline the changes suggested by the results of our formative evaluation as necessary to BRRR1 and summarise how these were addressed. We then take the example task which was chosen in chapter three to illustrate the use of BRRR1 and use it again to illustrate the use of BRRR2 in finding required components. Following that, we describe the design changes of BRRR2 more fully, emphasising the differences between it and the BRRR1 prototype. Finally, we present an overview of the implementation.

5.1 Revising BRRR1

In this section, we overview the problems of BRRR1 identified from the empirical study described in chapter four and outline the main design changes which this entailed and which are incorporated in this second version of the system, BRRR2.

5.1.1 Design changes suggested by the formative evaluation

In the empirical study of BRRR1 described in the previous chapter, the following problems with the design of BRRR1 were identified:

a) Users misunderstood some class descriptors, this suggested that we need to change the way in which the class descriptors are presented to make their meaning easier to understand. Additionally, we need to provide users with a help facility to explain the meaning of the class descriptors.

b) Users misunderstood some method categories, this suggested that we need to provide a help facility to explain the meaning of the method categories. It also suggested that we need to adjust the method classification approach based on users' understanding of the method categories.

c) Users showed that they needed a 'history' mechanism to help them remember the interaction history with the system as well as to examine the items they have seen before which might remind them about the necessary information.

To overcome these problems in the revised system, BRRR2, the following basic design changes were made.

5.1.2 Design changes incorporated in BRRR2

a) In order to address the first problem, as well as from the consideration that it will be necessary to incorporate all Smalltalk classes into the tool in the future, the indexing scheme to the classes has been changed. Instead of using class descriptors to index each class, all classes are now classified into class categories and the class categories are organized into a hierarchical structure. Corresponding to this change, a new pane — the Class Categories Hierarchy pane has been added in BRRR2's interface to illustrate graphically the structure of class categories. In addition, to further help users understand the meaning of a class category (i.e. what kind of classes the class category represents), an explanation mechanism has also been added.

b) To tackle the second problem, a cross-reference technique has been used in method classification. Instead of classifying a method into only one method category as in BRRR1, a method now may belong to more than one method category. In addition, some method categories are refined to contain several sub method categories. Moreover, an explanation facility is added to explain the meaning of method categories. As part of the revised interface, a special pane is now used to illustrate the structure of the method categories.

c) In response to the third type of problem, a 'Trace' window has been designed which records the most recent items a user has examined for later use.

In the following section, we show how to use the revised system, BRRR2, to find required components. This is done using the example used in chapter three as an illustration. We hope this will demonstrate in a practical way, the design differences between BRRR2 and BRRR1, before we describe these in more detail in section 5.3.

5.2 Finding components in BRRR2 — an example query

To facilitate the step by step description of an example query process, we first give a brief overview of the interface of BRRR2.

5.2.1 The interface of BRRR2 outlined

The interface of BRRR2 consists of three types of windows:

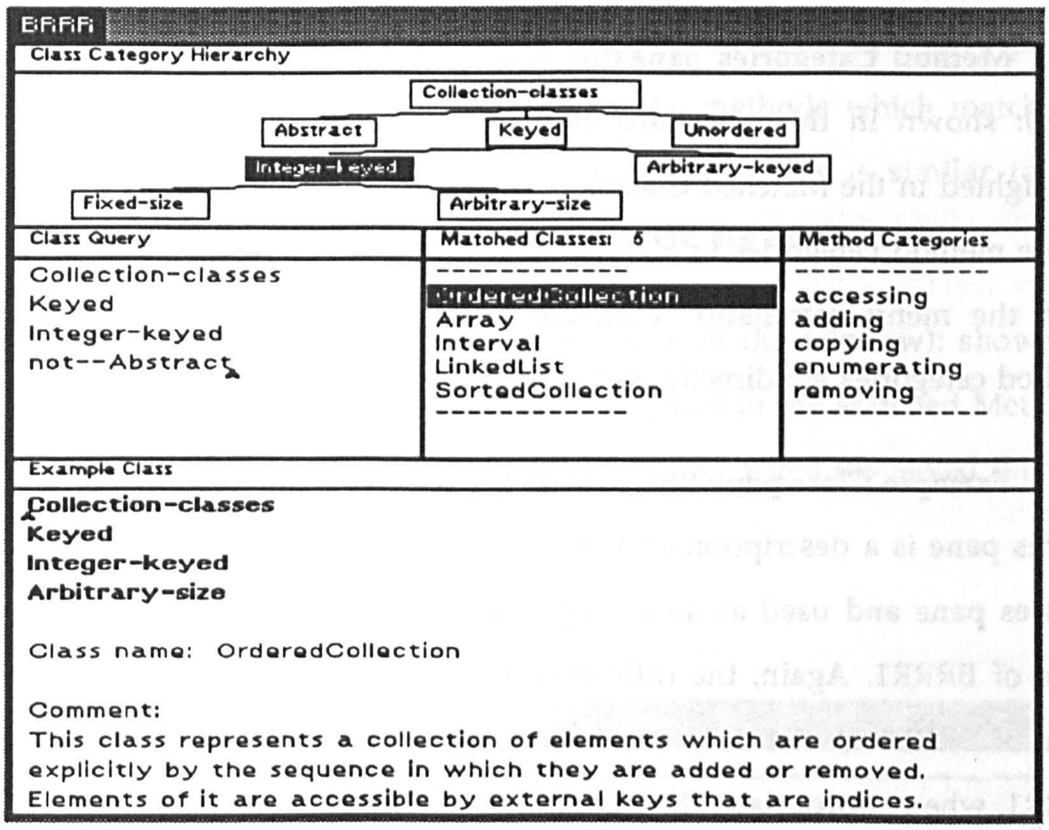
— the Class Level Query window

— the Method Level Query window

— the Trace window

The Class Level Query window

It is mainly in this window that a query for retrieving classes is created. The window consists of five panes (see figure 5.1). This is similar to the Main window of BRRR1 except that unlike BRRR1, it does not incorporate the Method Query pane.



Class Category Hierarchy

Class Query

```
Collection-classes
Keyed
Integer-keyed
not--Abstract
```

Matched Classes: 6

```
-------------
OrderedCollection
Array
Interval
LinkedList
SortedCollection
-------------
```

Method Categories

```
-------------
accessing
adding
copying
enumerating
removing
-------------
```

Example Class

Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name: OrderedCollection

Comment:
This class represents a collection of elements which are ordered explicitly by the sequence in which they are added or removed. Elements of it are accessible by external keys that are indices.

Figure 5.1. The five panes of a Class Level Query window.

The function of each of these panes can be outlined as follows:

i) **Class Category Hierarchy pane** (the pane at the top): illustrated in this pane is the hierarchical structure of the class categories. This pane takes over the duty of the Class Descriptor pane of BRRR1.

ii) **Class Query pane** (the leftmost pane below the Class Category Hierarchy pane): contained in this pane is a query constructed by a user for retrieving classes. This pane has a similar function to the Class Query pane in BRRR1.

iii) **Matched Classes pane** (the pane at the right of the Class Query pane): displayed in this pane are names of the classes which match a class query. It is similar to the Matched Items pane in BRRR1, but in this version, only classes are shown (note that in BRRR1, the Matched Items pane is used to show matched classes as well as methods).

iv) **Method Categories pane** (the pane at the right of the Matched classes pane): shown in this pane are method categories of the class which is highlighted in the Matched Classes pane. There is no such a pane in BRRR1, where method categories in a class could only be seen in the Example pane after the menu command 'View method categories' is used. Now, the method categories are directly visible to users.

v) **Example Class pane** (the pane at the bottom of the window): presented in this pane is a description of a class which is highlighted in the Matched Classes pane and used as an example. This pane is similar to the Example pane of BRRR1. Again, the difference between this pane and the Example pane of BRRR1 is that only the description of a class was shown, while in BRRR1, when appropriate, the description of a method can also be displayed in the Example pane.

The Method Level Query window

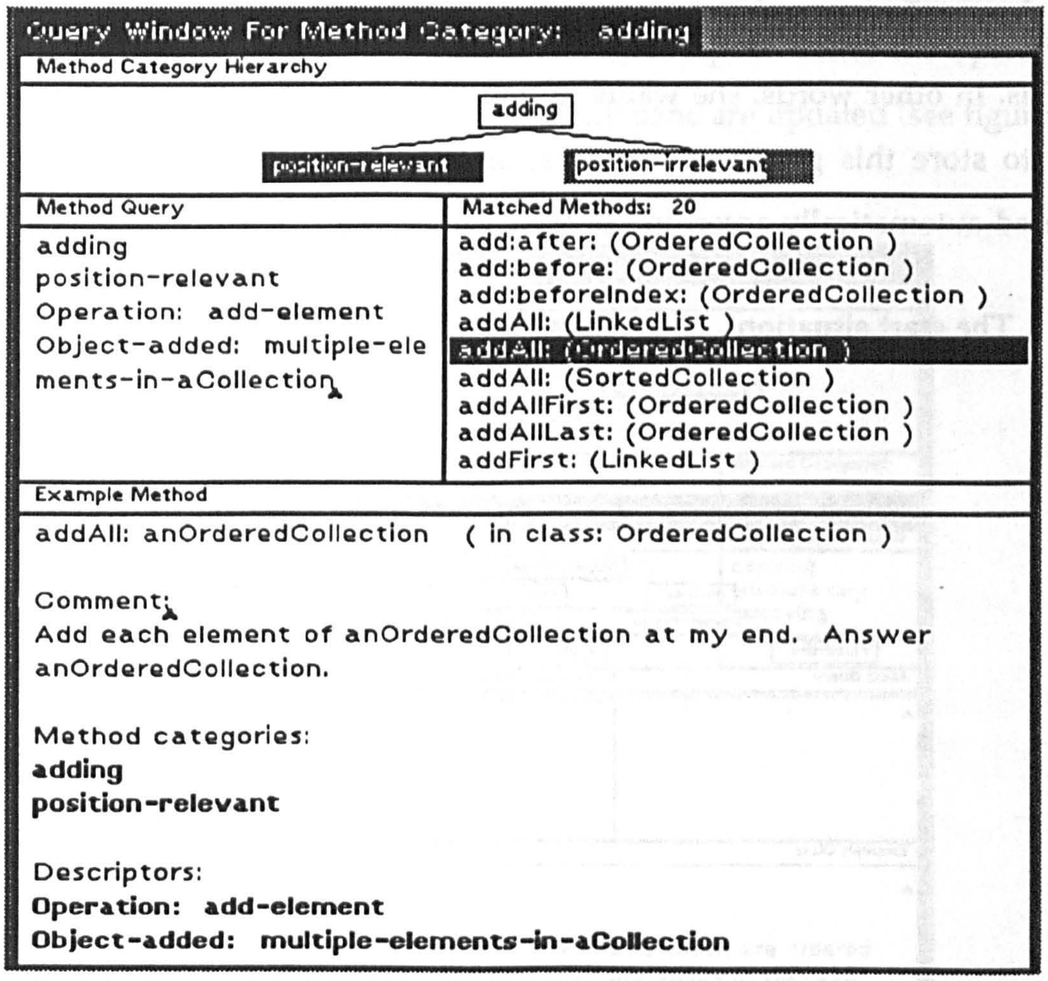
A Method Level Query window is opened only when a user starts to create a method query to retrieve methods. It is in this window that a method query is created. In essence, this window takes the whole duty of the Method Query pane of the Main window and the main duty of the Method Examination window of BRRR1. The window comprises four panes (see figure 5.2) whose functions are summarised below.

i) **Method Category Hierarchy pane** (at the top): displayed in this pane is the hierarchical structure of method categories.

ii) **Method Query pane** (leftmost pane below the Method Category Hierarchy pane): contained in this pane is the method query created by a user to retrieve methods. This is similar to the Method Query pane in BRRR1's Main window.

iii) **Matched method pane** (to the right of the Method Query pane): displayed in this pane is a list of names of the methods which match the current method query in the Method Query pane. This is similar to the Matched Items pane of BRRR1 when shown there are methods.

iv) **Example Method pane** (the bottom pane of the window): shown in this pane is the description of a method highlighted in the Matched Methods pane. This is similar to the Example pane of BRRR1 when displayed there is the description of an example method.



adding
position-relevant
Operation: add-element
Object-added: multiple-elements-in-aCollection

```
add:after: (OrderedCollection )
add:before: (OrderedCollection )
add:beforeIndex: (OrderedCollection )
addAll: (LinkedList )
addAll: (OrderedCollection )
addAll: (SortedCollection )
addAllFirst: (OrderedCollection )
addAllLast: (OrderedCollection )
addFirst: (LinkedList )
```

addAll: anOrderedCollection (in class: OrderedCollection)

Comment:
Add each element of anOrderedCollection at my end. Answer anOrderedCollection.

Method categories:
adding
position-relevant

Descriptors:
Operation: add-element
Object-added: multiple-elements-in-aCollection

Figure 5.2. The four panes of a Method Level Query window.

The Trace window

The function of this window is not directly relevant to the illustration of the example query which follows, so details of its design will be postponed and it will be described more fully in section 5.4.4.

5.2.2 The example

The task is the same as that in chapter three which for convenience we re-present here:

'Suppose a user has a group of numbers and needs to sort them in either ascending or descending order and she wants to find a class in Smalltalk to do this. In other words, she wants to find a class whose instances should be able to store this group of numbers, and the numbers put in should be ordered automatically according to their values.'

a) The start situation

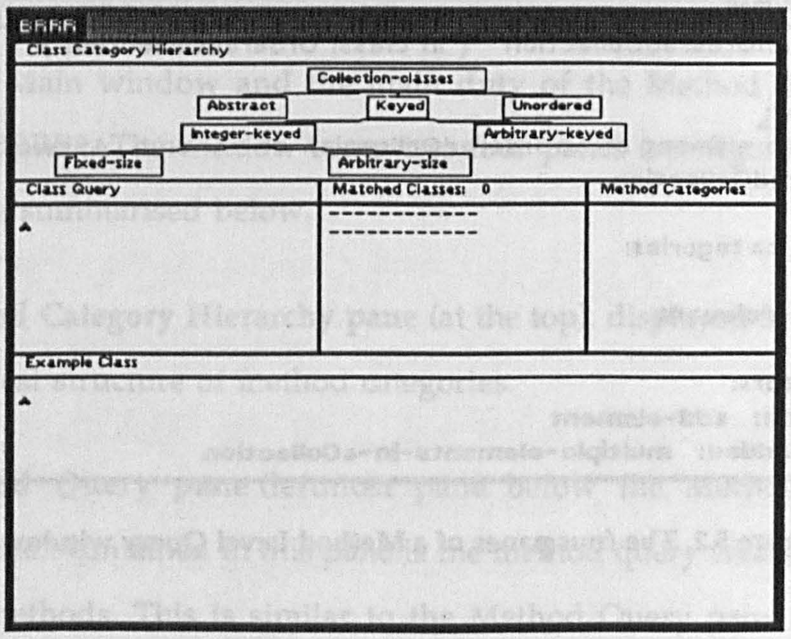


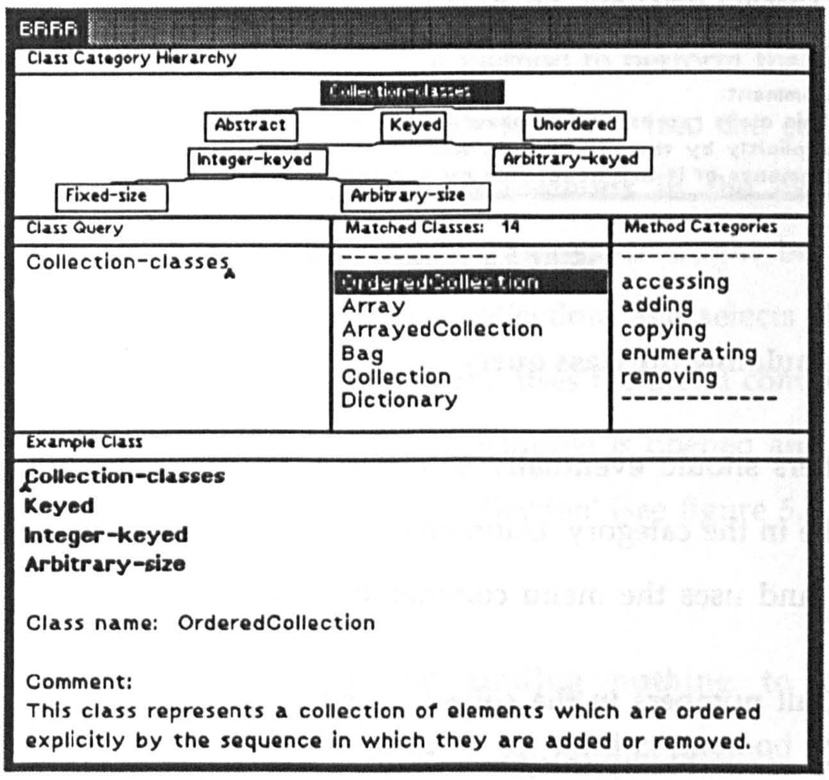
Figure 5.3. The start situation.

At this stage the BRRR2 interface is as shown in figure 5.3 above. The user sees the Class Level Query window described in section 5.2.1 above.

b) Beginning the querying process

The user needs a component to store numbers, so she thinks that she requires a collection class. She selects the class category: 'Collection-classes' from the Class Category Hierarchy pane and then chooses the option 'require' from a pop up menu in this pane. After the operation, the category name appears in the Class Query pane.

The user now asks the system to do a retrieval by selecting the option 'retrieve class' from the menu in the Class Query pane. After the operation, all panes except the Class Category Hierarchy pane are updated (see figure 5.4 below).



Collection-classes

OrderedCollection
Array
ArrayedCollection
Bag
Collection
Dictionary

----
accessing
adding
copying
enumerating
removing

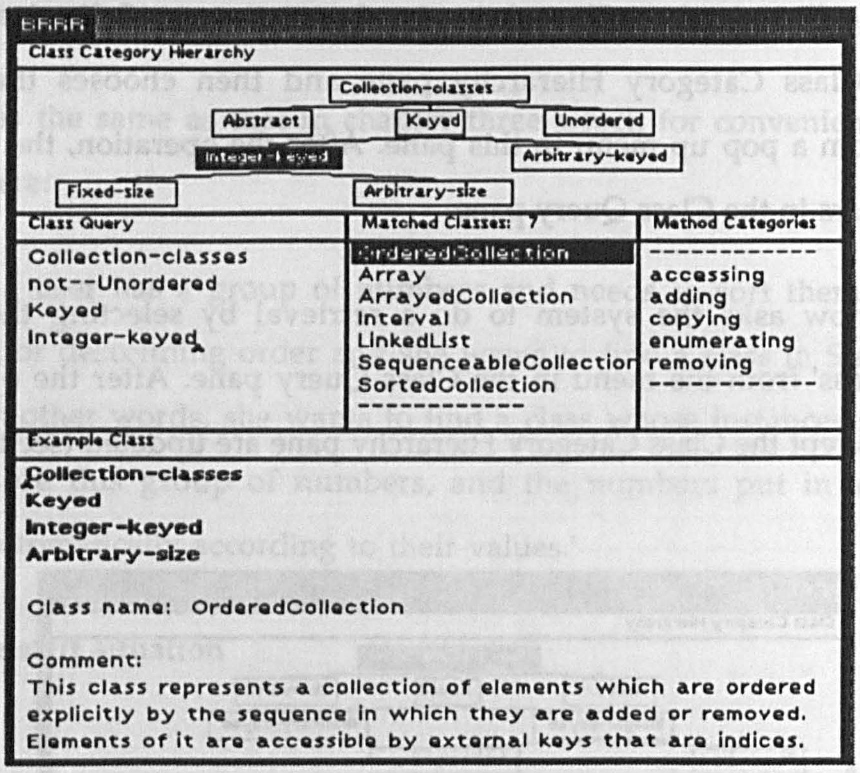
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name: OrderedCollection

Comment:
This class represents a collection of elements which are ordered explicitly by the sequence in which they are added or removed.

Figure 5.4. The Collection classes.

Note that all the collection classes found by BRRR2 are shown in the Matched Class pane, and the first item in the list — the class 'OrderedCollection' is highlighted. Its method categories appear in the Method Categories pane. The description of the class 'OrderedCollection' is presented in the Example Class pane.



```
Collection-classes          OrderedCollection            ------------
not--Unordered              Array                        accessing
Keyed                       ArrayedCollection            adding
Integer-keyed               Interval                     copying
                            LinkedList                   enumerating
                            SequenceableCollection       removing
                            SortedCollection             ------------
                            ------------
```

```
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name: OrderedCollection

Comment:
This class represents a collection of elements which are ordered
explicitly by the sequence in which they are added or removed.
Elements of it are accessible by external keys that are indices.
```

Figure 5.5. Matched classes.

c) Reformulating the class query

As all numbers should eventually be ordered, the collection the user needs should not be in the category 'Unordered'. Therefore, she selects the category 'Unordered' and uses the menu command: 'prohibit'.

In addition, all numbers in the collection are ordered, therefore, she thinks that they are accessible by integer indexes. Thus she 'requires' the category: 'Integer-keyed', this category together with its parent category 'Keyed' appears in the Class Query pane.

Now, she wants to retrieve all classes satisfying the conditions she has specified so far, so she requests a retrieval again (see figure 5.5 in the last page). This time, BRRR2 shows all classes in the Matched Class pane which are :

Collection classes [and]

not Unordered [and]

Keyed [and]

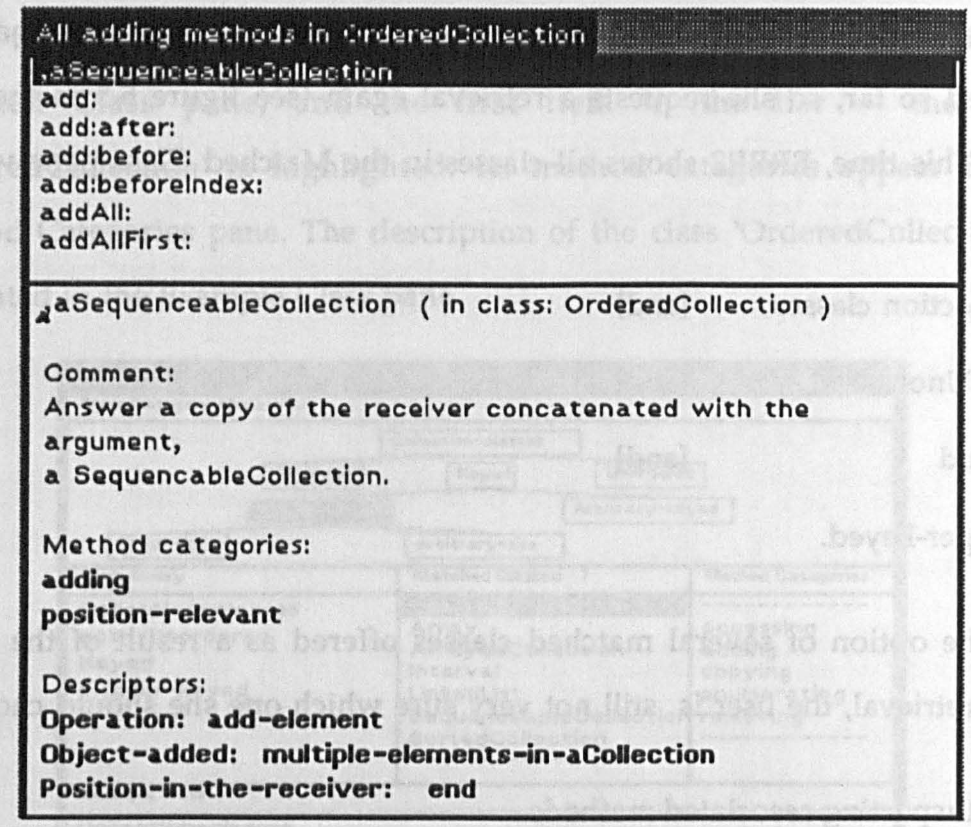
Integer-keyed.

With the option of several matched classes offered as a result of the most recent retrieval, the user is still not very sure which one she should choose.

d) Inspecting associated methods

She thus decides to examine the methods of the matched classes in order to see which class has, associated with it, a method to perform the function she needs, i.e. the method which can put the numbers into the collection and order them. She first wants to see the methods in the example class 'OrderedCollection'. It seems the 'adding' methods might be interesting since she needs to put numbers into a collection. She selects the 'adding' category in the Method Categories pane and uses the menu command: 'show methods in this category'. As a result, a window is opened and listed in it are all the 'adding' methods in 'OrderedCollection' (see figure 5.6 in the next page).

After browsing through them and finding nothing to match her requirements, she decides to see if there is an 'adding' method in any other matched classes.


All adding methods in OrderedCollection

,aSequenceableCollection
add:
add:after:
add:before:
add:beforeIndex:
addAll:
addAllFirst:

,aSequenceableCollection (in class: OrderedCollection)

Comment:
Answer a copy of the receiver concatenated with the argument,
a SequencableCollection.

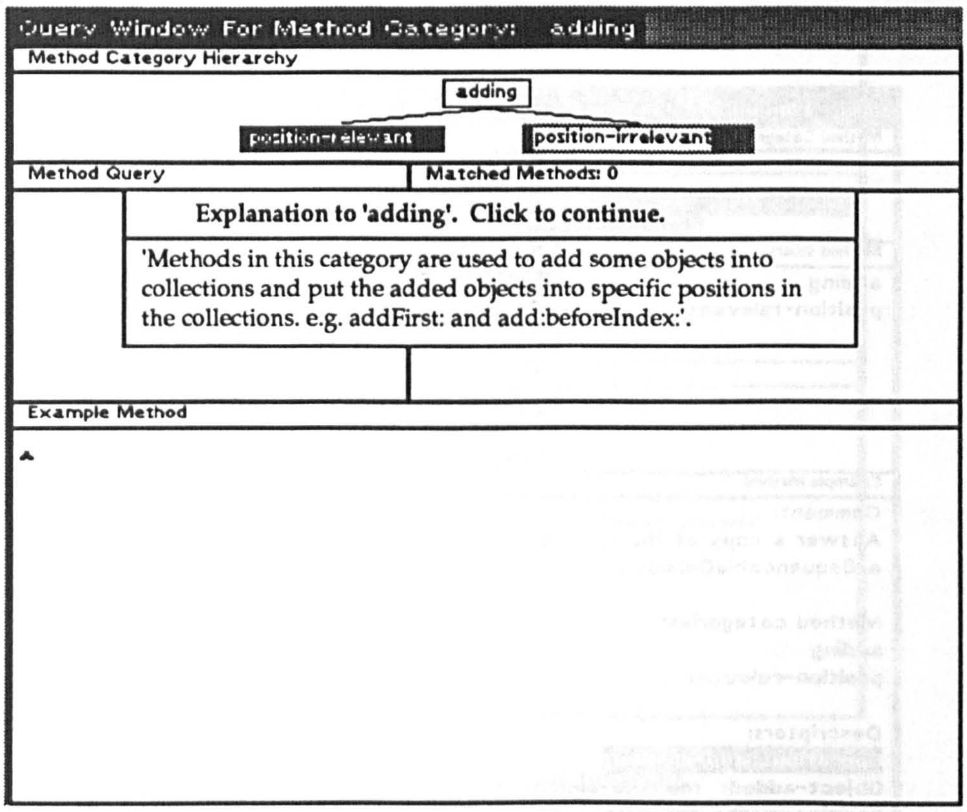
Method categories:
adding
position-relevant

Descriptors:
Operation: add-element
Object-added: multiple-elements-in-aCollection
Position-in-the-receiver: end

Figure 5.6. 'adding' methods in OrderedCollection are shown in a window.

e) Using the Method Level Query window

To do this she selects the menu option 'construct method query' in the Method Categories pane. After this option is executed, a Method Level Query window is opened (see figure 5.7 in the next page). In its Method Category Hierarchy pane, it can be seen that the method category 'adding' is further divided into two sub-categories: 'position-relevant' and 'position-irrelevant'. The user, though, is not sure about what kind of methods the 'position-relevant' represents, so she clicks on the category 'adding' and chooses the menu option: 'explain'. This causes a pop-up menu to appear which shows a text message (see the box "Explanation to 'adding'..." in figure 5.7).



Explanation to 'adding'. Click to continue.

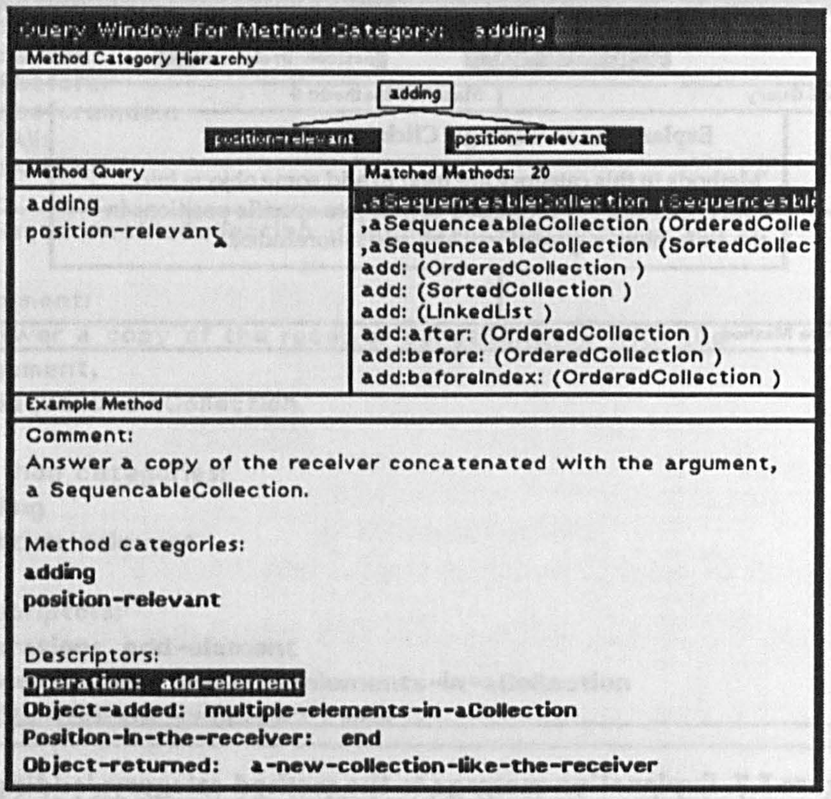
'Methods in this category are used to add some objects into collections and put the added objects into specific positions in the collections. e.g. addFirst: and add:beforeIndex:'.

Figure 5.7. Explanation message to the method category: 'adding'.

f) Query reformulation in the Method Level Query window

The user needs to sort the numbers, so she needs to put numbers into different positions in the collection. She clicks on 'position-relevant' and uses the menu option: 'require'. She then asks for a method retrieval with the menu option: 'retrieve methods' in the Method Query pane. After the retrieval, BRRR2 presents her with all methods which can add elements into a collection and put them into specific positions. The first one in the list — ',aSequenceableCollection' is highlighted and the description of this method is displayed in the Example Method pane (see figure 5.8).

Examining the method descriptors in the Example Method pane, she decides that she needs both the descriptors: 'Operation: add-element' and 'Objects-added: multiple-elements-in-aCollection', so she selects them and uses the menu option: 'require-this-value'. She is not satisfied with the descriptor 'Position-in-the-receiver: end', so she searches for alternatives with the



Query Window For Method Category: adding

Method Category Hierarchy

adding

position-relevant position-irrelevant

Method Query	Matched Methods: 20
adding	,aSequenceableCollection (Sequenceabl
position-relevant	,aSequenceableCollection (OrderedColle
	,aSequenceableCollection (SortedCollec
	add: (OrderedCollection)
	add: (SortedCollection)
	add: (LinkedList)
	add:after: (OrderedCollection)
	add:before: (OrderedCollection)
	add:beforeIndex: (OrderedCollection)

Example Method

Comment:
Answer a copy of the receiver concatenated with the argument, a SequencableCollection.

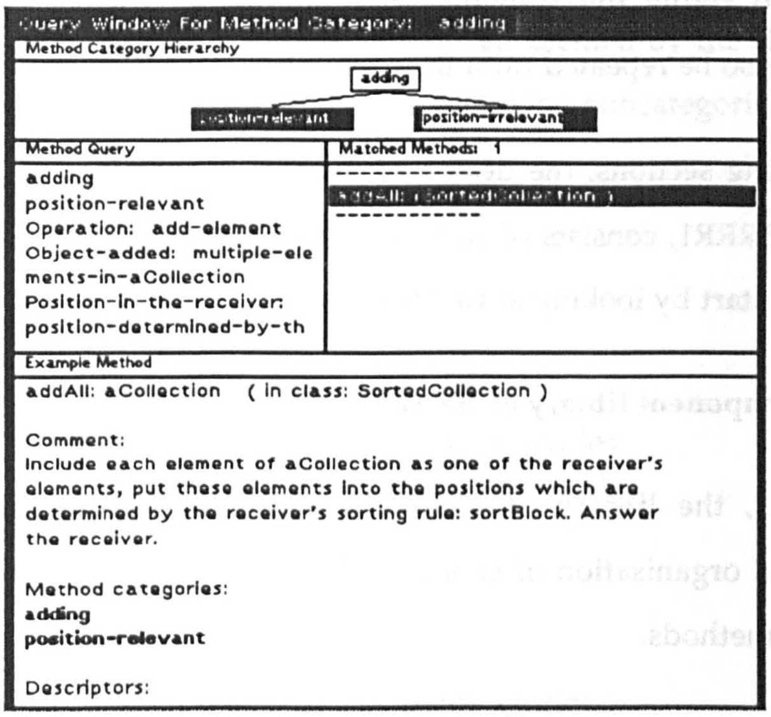
Method categories:
adding
position-relevant

Descriptors:
Operation: add-element
Object-added: multiple-elements-in-aCollection
Position-in-the-receiver: end
Object-returned: a-new-collection-like-the-receiver

Figure 5.8. 'position-relevant' methods.

menu option: 'alternative-values'. She examines the alternative values presented by the system in a pop up menu and then from the menu 'requires' the alternative descriptor 'position-determined-by-the-receiver's-sorting-rule'. Now, she requests a method retrieval again and gets one matched method: 'addAll: (SortedCollection)' (see figure 5.9).

She therefore decides it would be most appropriate to use the class 'SortedCollection' in her programming, thus bringing to an end her query session.



```
Query Window For Method Category:   adding
Method Category Hierarchy
                    adding
   position-relevant        position-irrelevant
Method Query                       | Matched Methods: 1
adding                             | -------------
position-relevant                  | addAll: (SortedCollection )
Operation:  add-element            | -------------
Object-added:  multiple-ele        |
ments-in-aCollection               |
Position-in-the-receiver:          |
position-determined-by-th          |
Example Method
addAll: aCollection   ( in class: SortedCollection )

Comment:
Include each element of aCollection as one of the receiver's
elements, put these elements into the positions which are
determined by the receiver's sorting rule: sortBlock. Answer
the receiver.

Method categories:
adding
position-relevant

Descriptors:
```

Figure 5.9. The result method.

5.2.3 Summary of the example query

From this example, it can be seen that the general principle and the operational procedure of BRRR2 is similar to that of BRRR1. Users start querying by creating a class query. This can be done initially by using the menu command: 'require' ('prohibit') to class categories which are illustrated in the Class Category Hierarchy pane in the Class Level Query window. If they want to refine the class query further, they can 'require' ('prohibit') more class categories from either the Class Category Hierarchy pane or the Example Class pane. After some cycles of classes retrieval are done, the users can examine the methods in matched classes and, if necessary, start a method query to find the necessary methods by opening a Method Level Query window. A method query is created by initially using 'require' ('prohibit') to select a method category which is displayed in the Method Category Hierarchy pane. The users may then use the method descriptors incorporated in the description of an example method provided by BRRR2 to refine the previous method query. The method retrieval process may also be repeated until the users get a satisfactory result.

In the following sections, the design of the tool is examined in more detail. BRRR2, like BRRR1, consists of two main parts: a component library and an interface. We start by looking at the design of the component library.

5.3 The component library of BRRR2

As in BRRR1, the library of BRRR2 contains the Collection classes of Smalltalk. The organisation of classes in BRRR2, are, as in BRRR1, different from that of methods.

5.3.1 Class classification

While the organization of methods is similar in BRRR1 and BRRR2, the organization of classes, however, differs in the two systems. This is more fully explained below.

a) Organization of classes in BRRR2

In BRRR2, all classes are classified into *class categories* and each class category represents a group of functionally similar classes. The classes represented by a category are said to *belong* to the category. In addition, all class categories are organized into a hierarchy called *class category hierarchy.*

A class category in the hierarchy may have a super category and several subcategories. A subcategory represents a special case of its super category, therefore any class which belongs to a subcategory also belongs to its super category. In other words, if a class *a* belongs to a class category B and B is a subcategory of the category C, then the class *a* also belongs to the category C. In addition, a subcategory may in turn have its own subcategories. The classification of the Collection classes in BRRR2 is illustrated in figure 5.10.

It can be seen that all Collection classes are represented by the class category: 'Collection-classes'. This category has the following subcategories:

Abstract

Keyed

Unordered

The subcategory 'Keyed' in turn has two subcategories:

Integer-keyed

Arbitrary-keyed

Again, the category 'Integer-keyed' has subcategories:

Fixed-size

Arbitrary-size

As each subcategory represents a specialization of its super category, we can see, for example that the classes belonging to 'Keyed' are special kind of 'Collection-classes'.

Each class in BRRR2 belongs to at least one class category, some classes, however, may belong to more than one class category. This is to reflect the fact that some classes have the properties represented by more than one category. For example (see figure 5.10), an instance of the class 'Dictionary' can be accessed by arbitrary type of keys, meanwhile, it can also be regarded as an unordered collection because the elements in a Dictionary are not maintained in any order. Therefore, the class 'Dictionary' belongs to both the category 'Arbitrary-keyed' and the category 'Unordered'. Having a class in more than one category increases the opportunity for users to locate it during a retrieval since they can find it in more than one place.

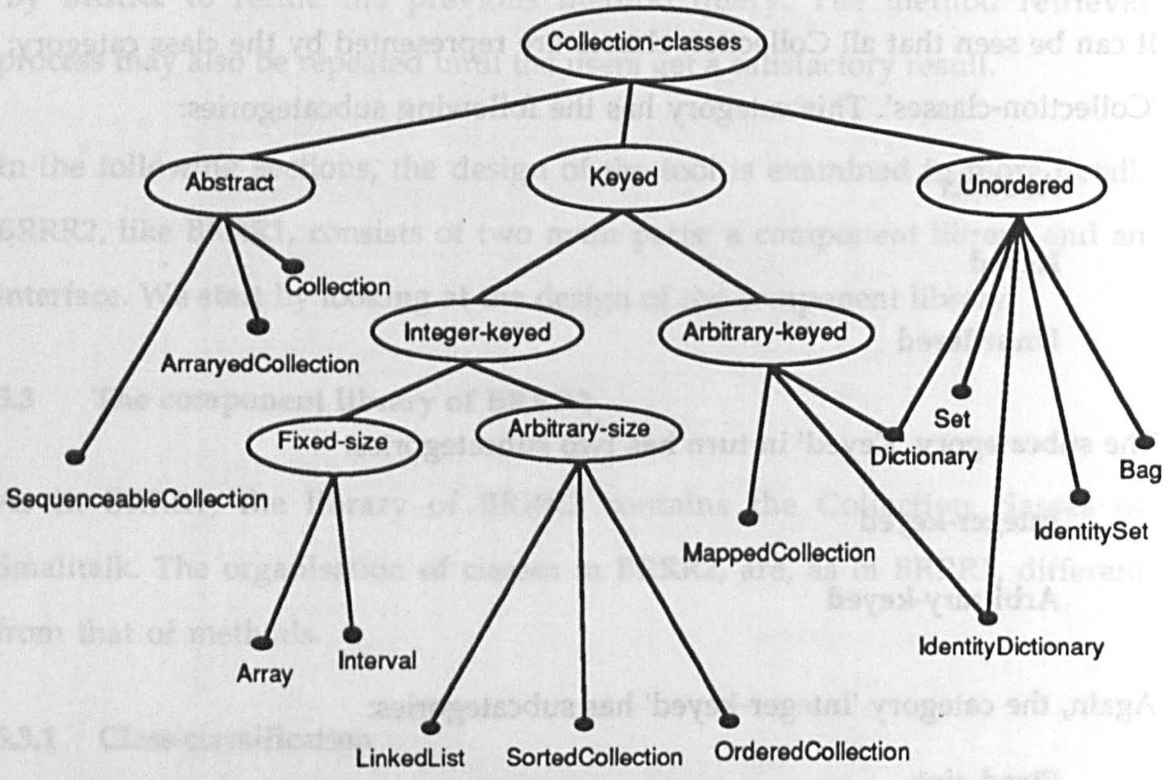


Figure 5.10. Class organization in BRRR2.

b) Design consideration underlying class classification

In BRRR1, all classes were indexed by a set of class descriptors. Each class descriptor indexes one or several classes which have the property specified by the descriptor. The descriptors are shown to users and are used to create class queries. One problem with this classification approach arises from the concern over the ability to scale up the system to include all classes of Smalltalk. BRRR1 only contains the Collection classes, if all Smalltalk classes are put in and indexed in this way, then a vast amount of class descriptors would be needed. Obviously, it is unnecessary and impossible to display at the same time all descriptors in the Class Descriptors pane. The descriptors therefore would need to be further organized. One possible way is to organize them into groups: The descriptors applicable to a particular set of classes (for example, Collection classes) are put into a group, while other descriptors suitable for other sets of classes also put into corresponding groups. In this way, we would have categories of class descriptor: descriptors for the Collection classes; descriptors for the graphics classes; for the interface classes; etc. In the query stage, it would be necessary to provide users a mechanism to ask them first specify which kind of classes they are interested and then present them in the Class Descriptors pane the descriptors corresponding to that group of classes. The users then can create a class query in the usual way as we introduced before. However, this kind of classification is virtually equivalent to classifying all classes into class categories — a scheme which is almost the one we have used in BRRR2.

The second problem with BRRR1's classification approach is that all class descriptors for a group of classes (in BRRR1's case, the Collection classes) are at the same level, the relationships existed between some class descriptors are difficult to be explicitly expressed and shown to users. For example, the descriptor 'elements-ordered' has relationships with both the descriptors: 'order-determined-externally' and 'order-determined-internally'. The descriptor 'elements-ordered' is a pre-condition for the latter two descriptors since only under the condition: 'elements-ordered', does it make sense to specify how the order is determined. We felt that it is important to express and show users this kind of relationship since it would to a certain extent help them understand the meaning of the descriptors. It seems necessary therefore to have a group of descriptors which can be used to both index the classes and to show users the relationships between the descriptors.

It was the two problems above which led us to adopt BRRR2's classification scheme. This scheme seems to fulfil the requirements we just mentioned. It is possible to classify all classes in Smalltalk into class categories. The class categories index the classes. The hierarchical structure of the class categories reveals their contextual relationship, i.e. a class category represents a group of classes which are special cases of its supercategory. In addition, in BRRR2's interface, we explicitly show the structure of the class organization to users (in the Class Categories Hierarchy pane). This should be helpful for them to understand the organization of the components and to locate the required information.

Once all classes in Smalltalk are put into BRRR2, the 'fisheye view' (Furnas, 1986) technique could be used in the interface. At the beginning of a query, only those class categories representing large group of classes are displayed in the system. Users can click on a category and all subcategories can then be displayed while irrelevant ones are hidden. This mechanism however has not yet been implemented in BRRR2.

To help users who still have difficulties in understanding some class categories, in BRRR2, an explanation facility is supplied which presents users with a text message about the type of classes a category represents. This can be seen later in section 5.4 where BRRR2's interface will be presented.

In summary, BRRR2's classification scheme of classes differs from that of BRRR1 mainly because of considerations of the ability to adapt the system to include all classes of Smalltalk and of helping users understand the class descriptors. In the next section, the method organization of BRRR2 will be discussed.

5.3.2 The method organization

a) Methods are classified into method categories

The method organization in BRRR2 is similar to that in BRRR1, i.e. methods from different classes are classified into method categories named *BRRR2 method categories.* BRRR2 method categories are virtually the same as BRRR1 method categories and the methods in each category are the same as its counterpart in BRRR1. Unlike BRRR1, however, some BRRR2 method categories which contain a large number of methods are further divided to contain several sub-method categories.

For example, the category 'adding' represents all methods in all Collection classes which are used to put new objects into a collection. This category itself contains two subcategories: 'position-relevant' and 'position-irrelevant'.

The category 'position-relevant' represents all methods which add new objects into a collection and put them into some specific positions in the collection e.g. 'addFirst: (OrderedCollection)'; 'addAll: (SortedCollection)'.

Another feature of BRRR2 which is different from BRRR1 is that a method in BRRR2 may belong to more than one method category. For example, the method 'findFirst:' belongs to the category 'accessing' since it accesses the elements to find the index; it also belongs to the category 'enumerating' since elements of a collection are enumerated to find the index. This cross-reference mechanism increases the users' opportunities to identify correctly a required method category. To a certain extent this overcomes the problems users had with BRRR1 when they were selecting method categories, because unlike in BRRR2, in that system, where cross-referencing was not available, their selection options at any one point were more limited.

b) Methods in a category are described by method attributes

This is the same as in BRRR1. However, if a method belongs to more than one method category, the method has the sum of all the attributes that it possesses in each category.

Let us look for example at the method:

', aSequenceableCollection'

The function of this method is to make a copy of the collection receiving the message and to append another collection to the copy of the collection. The concatenated collection is returned as the result.

This method belongs to the method category 'position-relevant' (a subcategory of 'adding') since it adds a collection of objects to the end of another collection. Its attribute-value pairs for being in 'position-relevant' are:

Operation:	add
Objects-added:	multiple-elements-in-aCollection
Position-in-the-receiver:	end
Object-returned:	a-new-collection-like-the-receiver.

It also belongs to the method category 'copy-with-changes' (a subcategory of the method category 'copying') because it copies the original collection and changes the contents of the copy collection. Its attribute-value pairs in this category are:

Operation:	copy-and-append-another-collection
Object-returned:	a-new-collection-like-the-receiver

Thus this method has two sets of attributes. During a method retrieval, which set should be presented to a user as the description of the method is determined by the context in which the user is. If a user is looking at the 'adding' methods, then the first set of attributes is shown, if he/she is looking at the 'copying' methods, then the second set is shown.

5.3.3 Summary of component organization

So far, we have described the component organization in BRRR2's component library. The classes are organized into class categories and the categories are organized into a hierarchy. A class may belong to more than one class category. The methods are also arranged into method categories, and the categories are organized into a method category hierarchy, and likewise a method may belong to more than one method category. In the following section, we describe the user interface of BRRR2.

5.4 The interface of BRRR2

We have mentioned in section 5.3.1 that BRRR2's interface comprises three types of windows: the Class Level Query window, the Method Level Query window and the Trace window. We describe them in more detail below. First we look at the functions of the Class Level Query window, then those of the Method Level Query window. Following this we explain certain types of queries which need combining queries from both the Class Level Query window and the Method Level Query window and conclude this section by describing the Trace window.

5.4.1 The Class Level Query window

The Class Level Query Window (previously shown in figure 5.1) is used to retrieve classes. It comprises the following five panes, each of which will be subsequently described in more detail:

a) Class Category Hierarchy pane;

b) Class Query pane;.

c) Matched Classes pane;

d) Method Categories pane;

e) Example Class pane.

a) Class Category Hierarchy pane

Displayed in this pane is the class category hierarchy. Currently only the Collection classes are in BRRR2's library, thus presented in this pane is just the hierarchy for the Collection classes. The root node of the hierarchy is the category: 'Collection-classes'.

The purpose of this pane is twofold. Firstly, it explicitly shows users how the components are organized. This should help users understand the overall structure of the component organization. It also helps the users understand the meaning of individual categories and thus facilitates the construction of a query. Secondly, users can directly use the categories displayed in the pane to construct class queries. They can click on a category which interests them, then use the menu options to add it (or the negation of it) into the query (in the Class Query pane).

The menu options in this pane are:

require; prohibit; explain and show classes.

The first two are used to construct queries and the last two are used to help users understand the meaning of individual categories. The options are explained in more detail below:

require: adds the selected category into the query. It specifies that the classes sought must be in the category.

Each class category in the system except the one at the root of the hierarchy has a super category and a category represents a special case of its super category. If a non-root category is required, it implies that all its super categories (i.e. all categories along the path from the category to the root node of the hierarchy — 'Collection-classes') should also be required. Therefore, once 'require' is used, together with the category itself, all its super categories are automatically added into the query. For example, if a user 'requires' the category 'Integer-keyed', its super categories (see figure 5.10): 'Keyed' and 'Collection-classes' are also automatically put into the query.

prohibit: adds the negated form of the selected category into the query (i.e. prefix the category with the symbol: 'not--'). It specifies that the classes selected must not be in the category.

explain: presents in a menu a text message to explain the properties that the classes in the chosen category have.

show classes: lists in a menu the names of the classes contained in the category. Showing users the class names in a category should enhance their comprehension of the meaning of the category since mention of class names may remind them what kind of classes they are.

b) Class Query pane

This pane contains a query constructed by users for retrieving classes. As in BRRR1, descriptors in this pane are implicitly connected by the logical operator 'and'. BRRR2 still cannot process queries connected by the operator: 'or'.

The menu options for this pane are:

remove descriptors: deletes the selected descriptors from this pane. If a class category is removed, all its subcategories (if it has subcategories and those subcategories are part of the query in this pane) would also be automatically removed.

reset: resets the whole system so that users can start a new query. It clears away all class descriptors in this pane and if there is any method level query window left open, users would be prompted to close it. To avoid the situation where users select this option by mistake and cause the loss of their queries, the system asks users to confirm this selection.

retrieve classes: retrieves all classes which match the query in this pane and displays the results in the Matched Classes pane (see below).

For a class to match a query, the class must be in all 'required' class categories, meanwhile, it must not be in any of the 'prohibited' class categories.

Besides various forms of class categories, the class query usually also includes method attributes sent from a Method Level Query window. This type of query specifies that the classes sought should have a set of methods which satisfy the given method attributes. This will be discussed in more detail in section 5.4.2 where the Method Level Query window is described.

c) Matched Classes pane

This pane displays a list of names of the classes which are found as a result of processing a query in the Class Query pane.

d) Example Class pane

As in BRRR1, the description of the selected class is shown here as an example (again, see figure 5.1). The description comprises two parts: the categories to which the class belongs (the boldface word) and the comment (the plain text below the 'Comment:').

Showing the class categories to which the class belongs helps users understand what properties the class has. It can be seen, for example, from figure 5.1 that the selected class 'OrderedCollection' belongs to all the following categories:

'Collection-classes'

'Keyed'

'Integer-keyed'

'Arbitrary-size'.

e) Method Categories pane

Listed in this pane are the method categories of the class highlighted in the Matched Classes pane. It shows users what kinds of methods the selected class has. Users can use the following menu options to search for the needed information:

show methods in this category: opens an extra window, which shows the methods belonging to the selected method category of the currently highlighted class in the Matched Class pane. With this extra window, users can examine the methods in the selected method category (see figure 5.6). This option replaces the 'Specialize' option in BRRR1 and consequently the window opened is similar to the Method Examination window in BRRR1 except that there is no menu options in this window for creating a method query. In BRRR2, that function is shifted into the Method Level Query window.

explain: presents users in a pop up menu a text message to explain the properties of the methods in a method category. Its purpose is to help users understand what kind of methods the category represents.

construct method query: opens a Method Level Query window to allow users to query all methods which are in a selected method category and which are contained in matched classes. It is from here that a method query is started. Users can select from this pane a method category which they believe represents the methods they need and use this option.

other method categories: shows in a pop up menu method categories which are not in the currently selected class but are in other classes which also match the current query. Sometimes the class currently selected by users has only a subset of the method categories possessed by all matched classes. Therefore the appropriate method category may not be available in the class which users are examining but in other matched classes. It is in this situation that this option is of use in finding the required category.

An example should make this clearer. Suppose a class query created by a user is:

'Collection-classes'

'Keyed'

'Integer-keyed'.

There are several classes matching the query and among them is the class 'Array'. Suppose the user selects the class 'Array' and is checking its method categories. The method categories in 'Array' appears in the 'Matched method categories' pane. They are as follows:

accessing;

enumerating;

copying.

Let us assume that the user now wants to find some methods which would add elements into a collection and put the added elements into a given position. Hence, she needs to find the method category 'adding' to construct a method query to find the required methods. More specifically, she needs to find a class which has the method category: 'adding' to query methods.

As she is now examining the class 'Array' and it does not have any 'adding' methods, therefore, the category 'adding' is not available to her. However, she guesses that the category 'adding' might be in other matched classes which she isn't examining so she selects the option 'other method categories in matched classes'. BRRR2 will present her all method categories which are not in the class she is inspecting, i.e. 'Array', but are in other classes which also match the current class query.

In this case, the method categories: 'adding' and 'removing' will be presented to her in a pop up menu. These two categories are not contained by 'Array' but by for example, both the classes 'OrderedCollection' and 'SortedCollection'. Now she can click on 'adding' and a Method Level Query window will open. She can then start to query all 'adding' methods according to the method query procedure to be described in the following section.

5.4.2 The Method Level Query Window

The Method Level Query window is used to query methods and is similar in structure to the Class Level Query Window (see figure 5.2). It has the following four panes, which will be described in detail below:

a) Method Category Hierarchy.

b) Method Query.

c) Matched Method.

d) Example Method.

a) Method Category Hierarchy pane

Shown in this pane is a method category hierarchy. The root node represents the method category selected by the user from the Method Category pane in the Class Level Query Window. The non-root nodes are sub-categories of the root category. For example, if a user selects the category 'adding' from the Method Category pane, then this category and its two subcategories: 'position-relevant' and 'position-irrelevant' would be presented in this pane.

The function of this pane is similar to the Class Category Hierarchy pane, i.e. to show users about how the components are organized and to let users manipulate those categories with a set of menu options to construct method queries.

Sometimes, constraints specified by a class query would lead to the situation where no methods in a particular method category are accessible, because the classes with which the methods in the method category are associated do not match the class query. In such a case, the colour of the node in the hierarchy representing that method category would be grey. This indicates that no methods in that category can be queried in subsequent method queries. To make this point clearer, let us look at an example.

Suppose a user in her initial class query has specified that the collection she needs should be 'integer-keyed', i.e. elements should be ordered as a sequence.

Next, she wants to query the 'adding' methods, so she opens a Method Level Query window. The method category 'adding' has two subcategories: 'position-relevant' and 'position-irrelevant'. Only the 'position-relevant' methods satisfy the user's requirements and only these can be further queried by the user.

The 'position-irrelevant' methods are not available for the subsequent querying. This is because the user has required that all collections should be 'integer-keyed'. Only 'position-relevant' methods satisfy this constraint. It therefore makes little sense to let user query methods in 'position-irrelevant' since they do not meet the requirement. Consequently, in the opened Method Level Query window, the node representing 'position-irrelevant' in the Method Category Hierarchy is greyed to notify the user of this.

There are four menu options for this pane:

require; prohibit; explain; show methods.

Their functions are similar to those of the Class Category Hierarchy pane. As the 'grey' node represents the non-accessible methods, if those options are used on a 'grey' node, an error message will be shown.

b) Method Query pane

As in the Class Query pane, this contains the method query constructed by users for retrieving methods. The descriptors in the Method Query pane are also implicitly connected with the logical operator: 'and'. Its menu options are:

retrieve methods: retrieves all methods satisfying the current method query and displays the results in the Matched Method pane (see below). To satisfy a method query, a method must satisfy the following constraints:

i) The method must belong to all of the 'required' method categories.

ii) The method must not belong to any of the 'prohibited' method categories.

iii) For all 'required' attribute-value pairs in the query, the attribute-values must be the same as those in the query.

iv) For all 'prohibited' attribute-value pairs in the query, the values must not be the same as those in the query.

remove descriptors: deletes the selected method descriptors (categories or attribute-value pairs).

clear: resets the Method Level Query Window. All method descriptors in this pane would be removed and the contents of the Matched methods pane and the Example method pane would be cleared.

get the current method: this is used in the following situation: a user may browse the methods retrieved after he/she has done some initial retrievals to examine their functions. During the browsing, he/she may find a method which satisfies his/her requirements. In such a situation, if he/she only wants that specific method, then he/she can directly use it. However, sometimes, the user may want to get <u>all</u> methods which have the same properties (i.e. method descriptors) as the one he/she is examining. Without this option, the user would have to construct a query to get all such methods (remember that one can only get one method each time by browsing). With this option in use, BRRR2 will automatically construct a default query (shown in this pane) and then retrieve all methods (including the one selected by the user in the Matched Method pane) which match the default query in the system.

merge into class query: sends the method query, prefixed with the phrase '**With method attributes:' into the Class Query pane of the Class Level Query Window. The purpose of this operation is discussed below in section 5.4.3.

c) Matched Method pane

As in the Matched Class pane, it contains the methods which match the method query.

d) Example Method pane

As in the Example Class pane, displayed in this pane is a description of the method highlighted in the Matched Method pane. The description of a method consists of the following parts:

the method categories to which the method belongs (the boldface text under the 'Method categories:' in figure 5.2).

the comment to the method which describes the function of the method (the text under the 'Comment:' in figure 5.2).

the attribute-value pairs which characterize the function of the method (the boldface text under the 'Descriptors:' in figure 5.2).

There are six menu options for this pane. The first three of these:

require-this-category

prohibit-this-category

explain-this-category

have the same meaning as **'require'**; **'prohibit'** and **'explain'** in the Method Category Hierarchy pane respectively. With the provision of this set of options, users can manipulate the method categories from either the Method Category Hierarchy pane or this pane.

The other three menu options are used to manipulate the attribute-value pairs to construct a method query and are:

require-this-value: adds the selected attribute-value pair into the Method Query pane. It is equivalent to the command 'require' in BRRR1's Example pane.

prohibit-this-value: adds the negated form of the selected attribute-value pair (i.e. prefix the value with: 'not--') into the Method Query pane. It is equivalent to the command 'prohibit' in BRRR1's Example pane.

alternative-values: shows in a pop up menu all values the system knows of the selected attribute. Users can select a value from the menu and then 'require' or 'prohibit' it.

5.4.3 The merge of the Class and Method queries

Now that we have outlined the functions of the Class Level Query window and the Method Level Query window, we shall explain the operation of the menu option: 'merge into class query' in the Method Query pane. As we said in chapter three, the method query should be regarded as an extension of the class query. After several cycles of class retrievals, users usually get a number of matched classes. They then need to examine the methods in these classes to find the ones which have the methods satisfying their requirements. However, the conditions the methods should satisfy usually cannot be adequately specified by class level queries (comprising the class categories) alone. Therefore, they need to open a Method Level Query window to start a method retrieval. A method query is equivalent to requesting the system to find out all methods in the matched classes which satisfy certain criteria.

Given that a method query is an extension of a class query, it makes sense to integrate them together and treat them as a whole new class query. This new class query then finds classes which satisfy the following conditions:

they satisfy the requirements represented by the original class query;

the methods they have satisfy the requirements represented by the original method query.

This is to increase the system's query capacities, as we show in the example below:

If, for example, a class query is:

'Collection-classes'

'Keyed'

'Integer-keyed'.

Suppose that after the class retrieval, the user has already created a method query and retrieved several methods. The current method query (in a Method Level Query window) is:

'adding'

'position-relevant'

'Operation: add'

'Objects-added: one'

'Position-in-the-receiver: first'

'Object-returned: add-element'.

The user can merge this method query from the Method Level Query window to the Class Query pane of the Class Level Query window with the menu option 'merge into class query'. Now, the contents of the Class Query pane becomes:

'Collection-classes'

'Keyed'

'Integer-keyed'

**With method attributes:

'adding'

'position-relevant'

'Operation: add'

'Objects-added: one'

'Position-in-the-receiver: first'

'Object-returned: add-element'

This new query asks the system to retrieve all classes which are collection classes; which are 'keyed' and have integer keys, in addition, the classes to be retrieved should have such methods that they are able to add one object into the first place of the collection and return the element just added as the result.

After such a retrieval, the system will show all classes matching the conditions specified in the query in the Matched Class pane. From now on, users may start another method retrieval (for example, to retrieve methods in another method category: 'accessing') in another Method Level Query window and then merge the new method query back to the class query again.

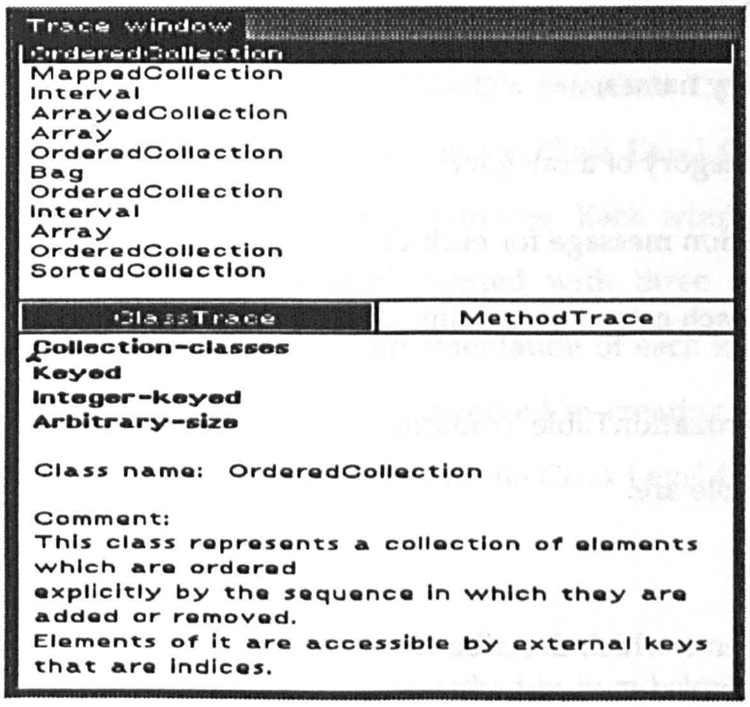
In this way, BRRR2 can now retrieve classes which, in addition to matching the previous merged query, contain the methods which match the newly created method query about 'accessing' methods.

It is therefore possible in BRRR2 to retrieve classes which contain methods from more than one method category, while in BRRR1 it was only possible to retrieve classes which contained the methods from one method category.

5.4.4 The 'Trace' window

The Trace window (see figure 5.11) serves as a 'history' facility. It records the last 15 classes and methods users examined during the retrieval. It is to meet the users' need for a facility to help them remember the components they have investigated before and to allow them to refer to these components at a later stage. During a retrieval, users might examine many classes (or methods). At a later stage, users might want to refer to the earlier ones to remind themselves of something (e.g. a method they looked before may have similar features to a method they are looking for; they may want to see if they have examined a method before to make sure they don't always go back to a same position; etc.). The 'Trace' window also reminds users about how they reached a class (or method) they are examining during the search process.

The 'Trace' window is automatically updated whenever users select a class (or a method) from the Class Level Query (or Method Level Query) window. By selecting the 'ClassTrace' or 'MethodTrace' option, the class or method history is displayed respectively. Users can select any item in the top pane of the window, the description of the selected item is shown in the bottom pane. The description of an item is the same as that in the Example Class (method) pane of the query windows. Therefore, users are reminded about the characteristics of the classes or methods they saw before.



```
OrderedCollection
MappedCollection
Interval
ArrayedCollection
Array
OrderedCollection
Bag
OrderedCollection
Interval
Array
OrderedCollection
SortedCollection
```

```
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name:  OrderedCollection

Comment:
This class represents a collection of elements
which are ordered
explicitly by the sequence in which they are
added or removed.
Elements of it are accessible by external keys
that are indices.
```

Figure 5.11. The Trace window.

5.5 The implementation of BRRR2 — an overview

BRRR2 is implemented with 18 new classes and the size of the program's source code is around 180k. It has two main components: a database which stores the information about individual classes and methods; and an interface component which implements the system's interface. We describe them in turn below.

5.5.1 The database

The class for implementing BRRR2's database is a newly designed class Brrr2Organization which is a subclass of the existing class Model. Defined in this class are a number of tables which store information about classes and methods. A group of methods are also provided in this class to access information in those tables. The most important tables are: classCategoryTable; classOrganizationTable; methodCategoryTable and methodOrganizationTable.

The classCategoryTable stores information about class categories, it has the following items:

class category names;

the super category of a category;

the explanation message for each category;

the classes each category contains.

The classOrganizationTable contains descriptions of each class, the main items in this table are:

class names;

class comments which describe function of each class;

class categories to which each class belongs;

method categories belonging to each class.

superclass of each class.

The methodCategoryTable stores information about method categories; it holds items similar to those of the classCategoryTable.

The methodOrganization contains information about the methods in the system, it has the following items:

method names;

comment of each method;

method categories to which each method belongs;

class with which a method is associated;

method descriptors (attribute-value pairs).

All tables are implemented as instances of the class Dictionary and those instances are represented by instance variables in Brrr2Organization.

5.5.2 The interface component

The interface component implements BRRR2's interface, i.e. the windows with panes. BRRR2 has three types of windows: Class Level Query window; Method Level Query window and Trace window. Each window consists of several panes and each pane is implemented with three components: a model; a view and a controller. The implementation of each kind of window is introduced below. We list the classes involved in creating the panes and briefly describe their functions. We start with the Class Level Query window.

a. Class Level Query window

This window has five panes which are described in turn below.

i) **Class Category Hierarchy pane**

The class for the 'model' of this pane is Brrr2Browser, which is a subclass of an existing class Model. It includes methods to access the database to obtain the information about class categories. The 'view' object of this pane then displays the categories graphically based on the information from the 'model'. The 'view' class is CategoryView which is a subclass of the existing class View. Defined in CategoryView are methods to display each category. Each node representing a class category is implemented as an instance of the class Form on which a string is displayed to show the category name it represents. The 'controller' class is CategoryViewController which is a subclass of the existing class ScrollController. A main function of the 'controller' is to track the cursor's position to see if a node is selected by users. Additionally, in this class, a set of menu messages is also defined to produce the menu of this pane.

ii) **Class Query pane**

The 'model' class of this pane is Query which is designed as a subclass of the existing class TextCollector. The 'view' class is QueryView, a subclass of the existing class TextCollector. The new class provides a new 'update:' method to display the query descriptors in a clear format. The 'controller' class is QueryController which is a subclass of the existing TextCollectorController. Provided in this class is a new set of menu messages for the menu of this pane.

iii) **Matched Class pane**

The 'model' class is Brrr2Browser, which supplies methods to the 'view' object of this pane in order to display a list of classes which matched the query. The 'view' class is the existing class SelectionInListView and the 'controller' class is the existing class SelectionInListViewController.

iv) Method Categories pane

This pane is very similar to the Matched Class pane in that both of them display a list of items. Therefore, the classes of 'model', 'view' and 'controller' of this pane are the same as those of Matched Class pane. However, in the 'model' class, Brrr2Browser, several methods are defined to provide the appropriate method categories for the 'view' object of this pane to display.

v) Example Class pane

The 'model' class is still Brrr2Browser, in which several methods are defined to obtain the appropriate text message from the database for the 'view' object to display. The 'view' class is the existing class CodeView. The 'controller' class is a new class ExampleClassController which is a subclass of the existing class CodeViewController. In the new class, a set of new menu messages is defined for the menu of this pane.

b. Method Level Query window

This window's implementation is very similar to the Class Level Query window, each pane is implemented in the same way as that in the Class Level Query window. Therefore, it is not described here further.

c. Trace window

The Trace window has four panes: the top pane shows a list of classes or methods, the two panes marked: 'classTrace' and 'methodTrace' serve as switches for users to examine classes or methods. The bottom pane shows a description of the selected class or method. All these panes share a 'model' class TraceModel which is a subclass of Model. TraceModel provides methods to record users' selections of classes or methods. The 'view' class of the top pane is SelectionInListView, and of the bottom pane is CodeView. The top and bottom pane use an instance of the existing class Controller respectively as their 'controller'. In the 'controller' objects of these two panes, no menu messages are defined. The 'view' class for both the switches is the existing class SwitchView.

5.6 Summary

In this chapter we have described BRRR2 — a revised version of BRRR1. BRRR2 incorporates several changes to deal with the problems identified through the formative evaluation of BRRR1.

The class classification method is different from that used in BRRR1. Currently, classes are organized into class categories with these categories further arranged into a hierarchy. A class now may be in more than one class category. In BRRR2's interface, a new pane is added to illustrate explicitly the structure of the class organization. This change is in consideration of the prospect of scaling the system up to incorporate all classes in Smalltalk in the future. It also aims to help users understand the class descriptors and facilitate creation of class queries.

In BRRR2, the way in which methods are organized is similar to that of BRRR1 with the exception that some method categories are divided into smaller categories and the categories are also organized into hierarchical structures. In addition, a method may belong to several method categories so that users' opportunities of identifying a correct method category are increased. An explanation facility for method categories is also provided in BRRR2.

Besides these two main changes, another new facility in BRRR2 is the trace window. This window records the most recent items examined by users so that they can inspect them at any stage. This also serves to remind users of their interacting histories with the system. Finally, in BRRR2, query capacity has been enhanced over that of BRRR1.

In the next chapter we report an empirical evaluation of BRRR2, which was designed to study the effectiveness of these changes in helping users retrieve required components.

Chapter 6 An empirical evaluation of BRRR2

In this chapter, we describe an empirical evaluation of BRRR2, the revised version discussed in the previous chapter, of the query tool BRRR1.

The purpose of the study, as with the evaluation of BRRR1 was to assess its strengths and weaknesses in practical use. In the following sections, we outline the organization of the experiment, present the results obtained and discuss their relevance for the design of BRRR2.

6.1 The organization of the study

As with the evaluation of BRRR1, a group of Smalltalk users, each of whom volunteered to take part, was used for this empirical study. Each subject spent in total approximately two to two and half hours completing the study.

6.1.1 Subjects taking part

A total of 10 subjects participated in the study. Nine of these were M.Sc. postgraduate students in computer science at Queen Mary and Westfield College in London. These subjects all had similar backgrounds. They all had a first degree in computer science and had taken a Smalltalk course which is a part of the M.Sc. syllabus. This course, spread over 20 weeks, comprises two hours of lectures and two hours of laboratory work a week. However, in this course, the Collection classes, (the components contained in BRRR2's library and the target search items of the study) were not studied in depth. Although these subjects had taken a Smalltalk course, considered in the larger context of the Smalltalk system, they were still regarded as non-expert users. The tenth subject was a member of staff in the Institute of Educational Technology at the Open University and who has a reasonably good knowledge of Smalltalk. Again it was hoped that this would provide an opportunity for eliciting useful feedback on the overall design of the system. None of the subjects had prior experience of using BRRR2.

6.1.2 The tasks used in the study

In the study, each user was requested to complete a set of five tasks. Each task required the user to find a component in the system to complete a programming task. The tasks were considered to represent typical situations where Smalltalk users would be looking for components while programming. The tasks were similar to those used in the evaluation study of BRRR1 and had the following pattern: find collection classes that have certain properties and additionally have a method which can perform a specific function. The tasks given are listed below:

1) You are looking for a collection to store ten numbers. The collection should allow its elements to be accessible through external indices. In addition, the collection should have a method which would give you the index of the last element in the collection whose value is greater than five.

Find all candidate classes and methods.

2) You are looking for a collection to store a sequence of objects:

(1 3 5 'john' 'simon' $p rectangle1 10).

The collection should have a method which would return you a *new collection* with the following properties:

- The elements of the new collection are the same as those in the original collection except that the 4th and 5th elements of the original collection are replaced by 7 and 9 respectively.

i.e. the new collection is: *(1 3 5 7 9 $p rectangle1 10).*

- After the method is executed, the original collection:

(1 3 5 'john' 'simon' $p rectangle1 10) remains unchanged.

Find all candidate classes and methods.

3) You are looking for a collection to store a group of 20 numbers. The collection should have a method which can be used to delete at once all numbers in the collection whose values are less than 10.

Find all candidate classes and methods.

4) You are looking for a collection to store a group of objects. You are not concerned about the order in which the objects are arranged in the collection. This collection should have a method which allows you to put an object into the collection each time it is used. However, if the object to be put in is already in the collection, this method will do nothing.

Find all candidate classes and methods.

5) Suppose you have a sequence of objects:

s1=(1 10 'john')

and you are looking for a collection to store them in the given order. At a later stage, you will want to use a method of this collection to append to it another sequence of objects:

s2=(2 'you' 'parent' 5 'children'),

The resulting collection would be:

(1 10 'john' 2 'you' 'parent' 5 'children').

However, you want the original collection *s1* to remain unchanged.

Find all candidate classes and methods.

6.1.3 The procedure

As in the previous evaluation described in chapter three, the study here consisted of two sessions: a training session at the beginning followed by an actual test session. A major difference, however, between these two evaluations was that in the training session for BRRR2, instead of being given a demonstration about how to use the system, each user was given a manual to read which gives instructions on how to use BRRR2. At the beginning of the manual, the 'retrieval by reformulation' principle is briefly introduced. The operation processes of the system are then described in detail coupled with examples and graphical illustrations. The manual is enclosed as Appendix B of this thesis.

After reading the manual, each user was asked to complete two exercises. The exercises have similar forms and similar difficulty levels to the tasks which the users would complete after the training session. The purpose of these exercises was to let users acquaint themselves with the system operation through practice. In the manual, some hints about how to find the answer to the first exercise were given but none were provided for the second. While users were completing the exercises, they were free to ask the experimenter questions about how to use the system. The whole training session, including both reading the manual and completing the exercises, took about one hour.

The test session followed directly on from the training session. In the test session, the user was presented with the set of five tasks to complete. Once a user had completed a task, he/she was prompted by the experimenter to check his/her result against the requirements stated in the task to make sure that, based on his/her understanding, the result satisfies the requirements. The user was then asked to write the result on an answer sheet provided.

The users' interactions with the system in the test session were videotaped. While they were completing the tasks, they were asked to 'think aloud' and their verbal protocols were also recorded.

6.2 The analysis of data

As in the evaluation of BRRR1, the user's performance on each task was analysed. The steps a user took to complete a task were listed, based on the videotapes. The user's hypothesis for taking those actions were identified based as closely as possible on his/her verbal protocols. Figure 6.1 shows such an analysis on a subject's (S4) performance on task 1. In this analysis, the plain style text after the word '**Trans:**' (abbreviation for 'Transcript') is the user's transcript. The boldfaced text after '**Hyp:**' (abbreviation for 'Hypothesis') shows the hypothesis behind the user's step which was inferred by the analyser from his protocol. The italic text in square brackets preceded by the word 'note:' is the analyser's note about the user's activity between the steps. Two asterisks mark places where errors occurred, i.e the user hypothesizes wrongly. The underlined words are BRRR2's menu commands which the user selected in his interaction with the system.

The users' errors identified from the analysis were then classified into error categories which will be discussed in next section.

Task 1

Steps taken by the user

The user's protocol and hypothesis

Trans: I'm looking for collection classes.

Hyp: Look for collection classes.

1. Require the class category: 'Collection-classes'.

Trans: We need fixed size collections, ok, but for the moment, we deal with the indices. They are keyed.

Hyp: They should be keyed.

2. Require 'Keyed'

Trans: But (they are) not integer keyed, en..., [*note: reads the task again*] 'external indices', I suppose 'external indices' is non-integer, right?

Hyp: Suppose 'external indices' is non-integer indices.

***[note: here, he misunderstood the requirements of the task, the task doesn't indicate the type of 'external indices', so he should not have assumed it as 'non-integer'.]*

3. Require the class category: 'Arbitrary-keyed'.

4. Retrieve classes.

[note: got the class 'Dictionary' as the example class].

Trans: ...I'm going to examine 'accessing' method category to find out more information.

Hyp: Examine the method category 'accessing' to find more information.

5. Selects the method category: 'accessing' and uses the command 'construct method query'.

6. Require the method category: 'accessing'.

7. Require the method category: 'finding-index'.

8. Retrieve methods.

[note: gets the method: 'keyAtValue:' as the example method]

[note: browses matched methods]

Trans: Ok, ... no, we need an index, the index of the last element in the collection whose value is greater than 5. Let's go down here and have a look.

[note: reads the method descriptors.]

'find-index'; 'exception-action', ... let's look at this.

Hyp: Need to find an index, so check the attribute 'exception action'.

9. Selects the descriptor: 'Exception-action: none' and uses the command '<u>alternative-values</u>'.

Trans: No, not interesting.

[note: read the descriptor 'Operation: find-index'].

Operation: find-index, en ...

I seem to have come to a dead end. I can't find any methods which suggest we have some condition as well, where we can find some index, under the condition that the value is greater than 5. So I'll backtrack a bit. I'll go back to the class (level query window) and browse.

Hyp: Want to find a method which has some condition and can give the index which satisfies the condition that the value is greater than 5. Cannot find it here. So backtrack to the class level.

[note: goes back to the Class Level Query window. Reads the task again about the requirement to the methods].

Trans: ... so the 'accessing' seems to be the right way, it wouldn't be 'enumerating'. ... Ok, I'm going to look around.... It seems to me that 'arbitrary-keyed' classes don't support the feature we need which is to find a key subject to a condition. These (i.e. the methods retrieved last time) just tell you keys at certain places. So I'll go back, close this window and go along this (i.e. the 'integer-keyed' branch of the class category hierarchy). I'll remove this 'arbitrary-keyed' and go to 'integer-keyed' to see what is there.

Hyp: Look for a method which should give a key subject to a condition. It seems that 'arbitrary-keyed' classes do not support this property, so look at the 'integer-keyed' classes.

10. Remove the class category: 'Arbitrary-keyed' and require the class category: 'Integer-keyed'.

Trans: I've got ten numbers (in the collection), so I require (the class category) 'Fixed-size'.

Hyp: They should be fixed size, as there are 10 numbers in the collection.

11. Require the class category: 'Fixed-size'.

12. Retrieve classes.

Trans: (The method category) 'accessing' is what we require, so construct a method query.

Hyp: Need 'accessing' methods, so construct a method query.

13. Selects the method category: 'accessing' and uses the command 'construct method query'.

14. Require the method category: 'accessing'.

15. Require the method category: 'finding-index'.

16. Retrieve methods.

[note: gets the method: 'findFirst:' as the example method].

[note: browses methods]

Trans: 'findFirst:', yeah, that seems more like it. *[note: reads one of the descriptors]* 'the-first-element-matching-a-block', all right, let's change that to ... the 'last element'.

Hyp: The method 'findFirst:' seems more like it. Change the value: 'the-first-element-matching-aBlock' of the attribute: 'Object-indexed'. to the value 'last-element-...'.

17. Selects the descriptor: 'Object-indexed: the-first-element-matching-aBlock' and uses the command: 'alternative-values'.

18. From the alternative value list require the value: 'the-last-element-matching-aBlock'.

Trans: En, I want the integer index, ..., that's ok.

19. Retrieve methods.

[note: gets methods: 'findLast:'].

20. Merges the method query into the class query pane and <u>retrieve classes.</u>

Finished.

Figure 6.1. A sample analysis of the subject S4's performance on task 1.

6.3 Results and discussion

Apart from one user (S2) who failed to complete task 4, all other users completed their tasks. Overall, eighty-six percent of the total number of tasks were completed successfully. Forty percent of the total number of tasks were completed without errors. On average, users made 0.98 errors per task and the number of errors made on each task ranges from 0 to 4. Users mastered the principles of BRRR2 and quickly learned how to use the query tool. As we shall discuss later, the results of the study were encouraging in showing that the tool is helpful in assisting users to find required components for their programming. The performance of the users in terms of completing the tasks correctly is illustrated in table 6-1 below.

C= task completed correctly; F= task incorrectly completed.					
Subjects	Task 1	Task 2	Task 3	Task 4	Task 5
S1	C	C	C	C	C
S2	C	C	C	F	C
S3	C	C	F	C	F
S4	C	C	C	C	C
S5	C	C	C	C	C
S6	C	C	F	C	C
S7	F	F	C	F	C
S8	C	C	C	C	C
S9	C	C	C	C	C
S10	C	C	C	C	C

Table 6-1. Users' performance on the task set.

From observation of the subjects, we have seen that the design revisions of BRRR2 have successfully overcome the following problems we found in BRRR1. Firstly, because of the cross-reference technique we adopted in classifying components, the users had much less difficulty in identifying an appropriate method category to start a method query, a problem experienced by every user in using BRRR1. Secondly, every user used a method of query (re)formulation to find components rather than solely using the browsing approach.

Though the users still encountered some difficulties in using BRRR2, as we discuss below, the problems do not indicate the need for fundamental changes to the system. In the following sections, we look a little more closely at the errors generated by the users. While the object of the evaluation was primarily to test the effectiveness of BRRR2 as a query tool, the results brought to light other interesting findings. The study highlighted some of the problems relatively inexperienced users encounter in the task of learning Smalltalk. The overall distribution of errors which the users made in completing the tasks is shown in table 6-2 below.

Number of errors made in each task						
Subjects	Task 1	Task 2	Task 3	Task 4	Task 5	**Sub-total (per user)**
S1	1	1	0	1	0	**3**
S2	1	1	2	2	4	**10**
S3	1	0	3	2	1	**7**
S4	1	1	2	2	0	**6**
S5	0	2	1	0	0	**3**
S6	0	3	2	0	2	**7**
S7	3	2	1	1	0	**7**
S8	2	0	1	0	1	**4**
S9	0	0	0	0	0	**0**
S10	1	0	1	0	0	**2**
Sub-total (per task)	**10**	**10**	**13**	**8**	**8**	

Table 6-2. The distribution of errors the users made in completing the tasks.

While this table shows how many errors users made, the types of errors are discussed below. We have classified users' errors into the following three categories:

— Misunderstanding the operations of the system;

— Misunderstanding the contents of the system;

— Misconceptions of the programming tasks.

Table 6-3 below shows the percentage accounted for by each of these categories.

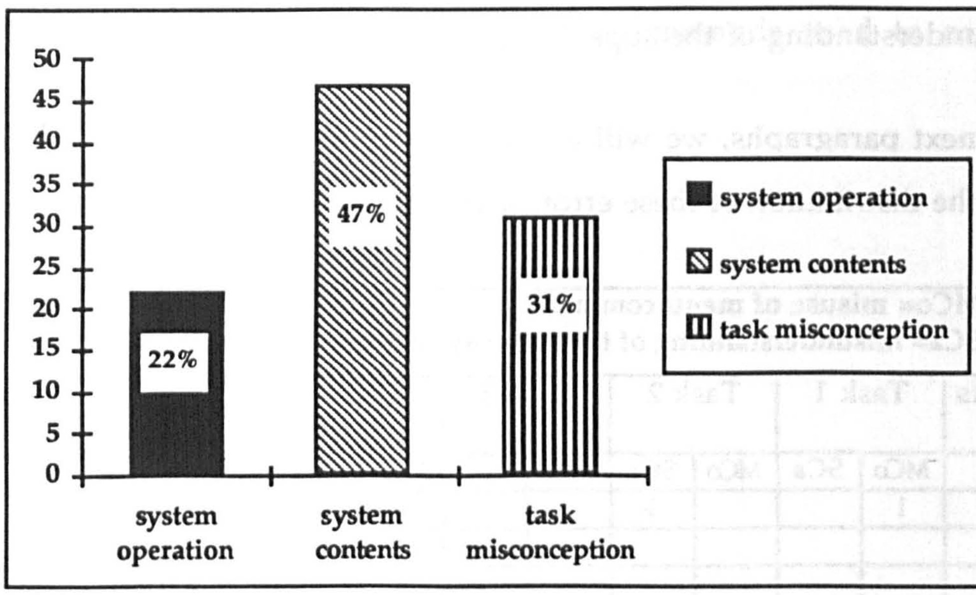


Table 6-3. Percentage that each error type accounts for.

While this classification is necessarily to a certain extent an arbitrary one, it satisfactorily indicates the differences, which we wish to discuss, in the types of errors which we encountered in the subjects' responses. A number of these errors are of most direct relevance to our evaluation study, since they are of interest in relation to the design and use of the query tool. Others, while meriting some attention, are more indicative of the difficulties non-expert users have in learning to program.

In the following sections, we describe each type in more detail, first presenting examples of errors, then putting forward likely causes for their occurrence. We also, where appropriate, suggest ways of reducing the likelihood of users making these errors.

6.3.1 Misunderstanding the operations of the system

This type of error accounted for twenty-two percent of all errors. Those errors can be further divided into the following two subcategories:

— misuse of menu commands

— misunderstanding of the super-subcategory relationships.

In the next paragraphs, we will analyse this type of error. Table 6-4 below shows the distribution of these errors across the tasks given.

MCo= misuse of menu commands; **SCa= misunderstanding of the super-subcategory relationships.**											
Subjects	Task 1		Task 2		Task 3		Task 4		Task 5		Sub-total (per user)
	MCo	SCa	MCo	SCa	MCo	SCa	MCo	SCa	MCo	SCa	
S1	1			1							**2**
S2											**0**
S3						1	1				**2**
S4											**0**
S5											**0**
S6			3		1				1		**5**
S7	1										**1**
S8		1									**1**
S9											**0**
S10											**0**
Sub-total (per task)	**2**	**1**	**3**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	

Table 6-4. Distribution of errors.

(The figure in each cell indicates the number of errors a user made during completing the corresponding task. Blank cells show that users did not make any mistake of that type.)

a) Misuse of menu commands

Errors of this kind were mainly made while users were constructing method queries in a Method Level Query window. Some users manipulated the attribute-value pairs with the menu commands designed for manipulating method categories, as we explain below. In BRRR2, the contents in the 'Example method' pane of a Method Level Query window is a description of one of the retrieved methods. It consists of three parts:

the *comment* which is a text description of the method's function;

the *method categories* to which the method belongs;

the *attribute-value pairs* which characterize the function of the method.

In the 'Example method' pane, the menu commands used to construct method queries are:

require-this-category;

prohibit-this-category;

explain-this-category;

require-this-value;

prohibit-this-value;

alternative-values.

Among them, the first three commands are designed to manipulate the method categories. The remaining three are used to manipulate the attribute-value pairs. Several users were not aware of the difference between these two groups of commands, and so during the creation of a method query, used the commands for category to manipulate attribute-value pairs. The users in many cases knew that they needed to 'require' an attribute-value pair, but mistakenly selected the 'require-this-category' command.

After the system gave an error message, they realized that they had made a mistake and then chose the correct command.

This type of error may well occur because the names of those two groups of commands are very similar and the difference between them are not indicated explicitly in the manual.

One possible solution to this problem is to re-assign names to the menu commands, i.e., do not differentiate the commands for categories from those for the attribute-value pairs, instead, assign the same name to both the command for category and its counterpart for attribute-value pairs and allow the system to deal with the different cases. If this were done, the command set in this pane would become:

require; prohibit; explain-this-category.

The 'require-this-category' and 'require-this-value' both mean that the needed method must have the property represented by the selected descriptor (a method category or an attribute-value pair). Therefore, it seems that the 'require' would be sufficient for both cases, the system, instead of the users, would take the load. This method can also be applied to the commands: 'prohibit-this-category' and 'prohibit-this-value', they would be replaced by the command 'prohibit'. In this way, the users' confusions might be reduced.

b) The super-subcategory relationships

In BRRR2, classes and methods are organized into class categories and method categories respectively. Categories are further arranged into hierarchical structures (see figure 6.2). Each category except the one at the root of the hierarchy has a parent category called its super category. A non-root category represents a special case of its super-categories. If a non-root category is required (with the command 'require-this-category') by a user, it implies that all its super categories are also needed by the user. Therefore BRRR2 would automatically put all super categories of the category into the query. Similarly, if a category is removed (with the command 'remove-this-category'), all its sub-categories would be automatically removed as well. However, a number of users failed to realize the super-subcategory relationships in the system. Consequently, they didn't fully understand the way in which the commands: 'require-this-category' and 'remove-this-category' work, therefore they constructed contradictory queries. For example, whilst completing task 1, a subject (S8) first required the category 'Arbitrary-keyed', then he thought that the collections he was looking for should have fixed size, so he also required 'Fixed-size' without realizing that this category is only applicable to 'Integer-keyed' classes. It can be seen from figure 6.2 that classes with fixed size in BRRR2 are also: 'Collection-classes' and 'Keyed' and 'Integer-keyed', so after he required 'Fixed-size', the super categories of the 'Fixed-size': 'Collection-classes', 'Keyed' and 'Integer-keyed' were also added into the query. The query therefore became as follows:

'Collection-classes' {and}

'Keyed' {and}

'Arbitrary-keyed' {and}

'Integer-keyed' {and}

'Fixed-size'.

As no collection classes in BRRR2 are both 'Arbitrary-keyed' and 'Integer-keyed', nothing was retrieved.

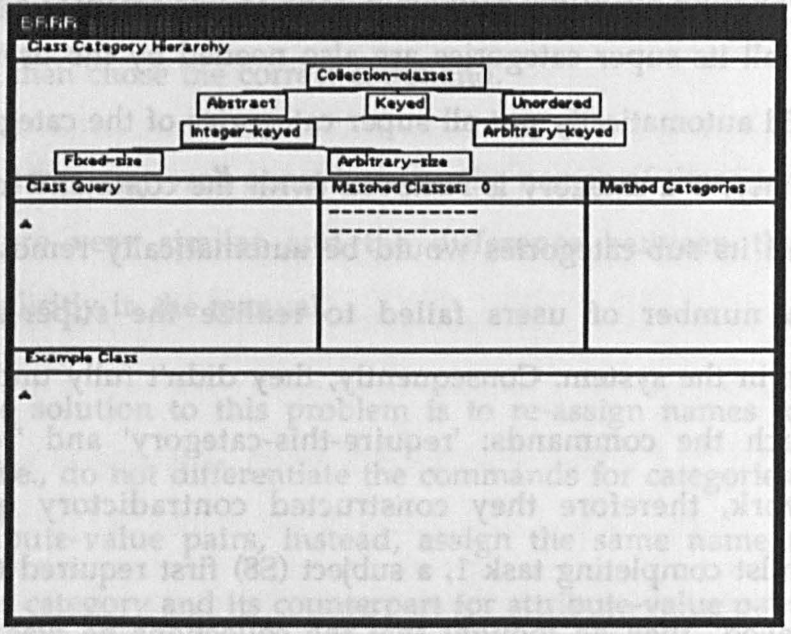


Figure 6.2. The class level query window.

Again, such a kind of error may well have its origin in the lack of training. In the system manual, the relationship between super and sub-categories is not particularly emphasized. In addition, nothing is said about the fact that the command 'require-this-category' and the command 'remove-this-category' would add (or remove) all super (or sub) categories of a category as well. In the examples given in the manual, users were shown that categories are 'required' (or 'prohibited') step by step, i.e. each supercategory of a category is required before the category itself is required. That particular way of showing users how to use the commands was chosen to ensure that users would first learn the basic way of using the commands, leaving more skilful ways till a later stage. Therefore we didn't present the full information to users in the manual. Viewed in this light, it is not hard to see that users made such mistakes. This result indicates the need for some revision of the manual given to users. We may need to explain to users explicitly the relationships between super-subcategories, state the full functions of these menu commands and show the complete effect of the commands in the examples used in the manual.

6.3.2 Misunderstanding the contents of the system

This type of error accounts for 47% of the whole error set. The majority of these errors occurred while the users were doing method retrievals (as each test task requires the users to find a method, the users spent considerable amount of time in method retrievals, thus there were more errors generated by the users during that part of the process). These errors can be more easily described in three sub-categories:

— misunderstanding values of attributes;

— misunderstanding functions of the retrieved items;

— misunderstanding method categories.

The distribution of these errors across the tasks given can be found in table 6-5. This is followed by a discussion in which each category of problems will be analysed in turn.

VA= misunderstanding values of attributes;
MF= misunderstanding functions of methods;
MC= misunderstanding method categories.

Subjects	Task 1			Task 2			Task 3			Task 4			Task 5			Sub-total (per user)
	VA	MF	MC	VA	MF	MC	VA	MF	MC	VA	MF	MC	VA	MF	MC	
S1										1						**1**
S2	1			1			1			1		1	1		2	**8**
S3			1						1							**2**
S4					1							2				**3**
S5				1	1		1									**3**
S6													1			**1**
S7			1	2												**3**
S8													1			**1**
S9																**0**
S10			1													**1**
Sub-total (per task)	**1**	**0**	**3**	**4**	**2**	**0**	**2**	**0**	**1**	**2**	**0**	**3**	**3**	**0**	**2**	

Table 6-5. The distribution of errors.

(Figures in cells indicate the number of errors users made while completing the corresponding tasks. Blank cells show that users did not make any mistakes of that kind.)

a) Misunderstanding values of attributes

Several users misunderstood the values of certain attributes used in describing individual methods and had difficulties in using them to construct method queries. In BRRR2, apart from the method categories, each method is characterized by a group of attributes and their corresponding values, the attribute-value pairs.

For example, the attribute-value pairs for the method 'copyReplaceFrom:to:with:', in the OrderedCollection class, are:

Operation: copy-and-replace-a-subCollection;

Object-returned: a-collection-like-the-receiver

In the above, 'Operation' is an attribute and the part after the ':' is the value of the attribute, similarly with 'Object-returned'. Users may use attributes-values to construct method queries. However, users were sometimes not sure about, or misunderstood the meanings, of certain values, and so they selected a wrong value or did not select a value appropriate to their queries. The problem appears to be mainly caused by the somewhat inappropriate selection of descriptors in BRRR2. For example, methods in the method category: 'removing' are characterized by the following attributes:

Operation;

Exception-action (i.e. the action to take if the object to be removed cannot be found in the collection);

Number-of-objects-removed;

Constraint-to-removed-objects;

Object-returned.

For all 'removing' methods, the value for the attribute: 'Operation' is: 'remove-element'.

Take as an example the method 'removeAll:' in class 'OrderedCollection', whose function is to remove a group of given elements from the collection. Its descriptors are:

Operation:	remove-element;
Exception-action:	none;
Number-of-objects-removed:	multiple;
Constraint-to-removed-objects:	match-elements-in-a-given- -collection;
Object-returned:	a-collection-of-removed-elements.

For another method 'remove:' in the same class, which removes only one element, the descriptors are:

Operation:	remove-element;
Exception-action:	none;
Number-of-objects-removed:	one;
Constraint-to-removed-objects:	match-the-given-element;
Object-returned:	removed-element.

Note that for both methods, the value for the attribute 'Operation' is 'remove-element'. In other words, this value applies to both the situation where one element is removed and the situation where multiple elements are removed.

The attribute used to specify the amount of the removed elements is 'Number-of-objects-removed' rather than the 'Operation'. However, a few users misunderstood the value 'remove-element' to mean 'remove only one element'. Consequently, when asked to find a method which should remove several elements, they thought the value 'remove-element' was not appropriate, hence looked for alternative values before they checked the attribute: 'Number-of-objects-removed'. The values of this attribute are identical for all 'removing' methods in the system, thus they couldn't find any alternative values.

This kind of problem could be overcome by adjusting the values which cause confusions. For example, if we change the value of the attribute 'Operation' from 'remove-element' to 'remove-element(s)' or even to 'remove', this kind of error might be avoided.

b) The function of retrieved methods

This type of error manifested itself in method retrievals, where the users might simply ignore the required methods provided by the system. After several method retrievals, the users were usually presented with a method by the system. Although the method shown by the system was in fact the required method, as the users couldn't fully comprehend its function, they didn't realize that the method was what they needed and continued their search. For example, during the course of completing task 2, a subject (S4) had found the method 'copyReplaceFrom:to:with:', which was in fact the correct method for the task. However, he misunderstood its function, so continued the search process.

The cause for these errors appears to be that the functions of the methods are complex and not straightforward to understand, and the comments for the methods are not clear enough to provide a sufficient explanation about their functions. The comments for all methods in BRRR2 came directly from Smalltalk and unfortunately some of them are not very comprehensible. BRRR2 is only a prototype system and is not yet integrated with the original

Smalltalk, and so users cannot run a component directly and discover its function through such an experiment or find how a component is used in the real programming situation.

One possible solution is to provide users with examples about how a complex component is used in real programming. The examples together with the comment for a component should improve users' understanding of the functions of the components and therefore help them in retrieving the required information. Of course, when BRRR2 is integrated with Smalltalk, users would be able to understand the function of a component better by directly running it or by seeing which other existing programs use this component and how they use it. This may also be helpful in easing this kind of difficulties.

c) Misunderstanding method categories

This type of error refers to those made by users who had difficulties in identifying the correct method categories in a method category hierarchy while they were creating a method query. They sometimes searched a wrong branch of a method hierarchy or used two categories which are mutually exclusive.

One such an example occurred while a subject (S10) was completing task 1. He was trying to find a method which would give the index of the last element (of a collection which contains a group of given numbers) whose value is greater than 5. He searched in the method category hierarchy whose root category is 'accessing'. There are four subcategories of 'accessing':

measurement: represents methods used to query various properties of collections.

inspecting-elements: represents methods used to retrieve values of elements in a collection.

finding-index: represents methods used to return indices of some elements which satisfy certain user-specified criteria.

replacing-elements: represents methods used to replace some existing elements in a collection.

The subject thought that he should first measure the elements in a collection and then find the required index. He therefore required both the categories 'measuring' and 'finding-index'. He didn't realize that methods in 'measuring' category only return values of some parameters about the collection (for example, size, sorting rule) and do not 'measure' the elements at all in the sense of examining elements based on certain criteria as he expected. As no method can both return a value of a parameter and an index at the same time, those two categories are mutually exclusive, therefore nothing was retrieved. This error was caused by the fact that the subject had misunderstood the meaning of the method category 'measurement'.

In BRRR2, methods are classified into method categories and the categories are organized into hierarchical structures. In order to find a method, users need to first identify a root category ('adding'; 'accessing'; 'copying'; etc.) as a start and then select a specific subcategory of the root category ('position-relevant'; 'position-irrelevant' etc.) to which the method belongs. They may then use the attributes in a category to further refine a method query. In BRRR1, users also had problems in identifying a correct method category. To overcome this problem, in BRRR2, a cross-reference technique is employed and the method category hierarchy is displayed visually. In BRRR2, the users did not have much difficulty in selecting the root category of a method category hierarchy. For example, they did not get stuck by selecting 'accessing' rather than 'enumerating' as happened in BRRR1. This suggests that the revision was at least partly successful.

Nonetheless, the users now have difficulties in identifying an appropriate subcategory of a root category. In BRRR2, a help facility is provided which on request presents users with an explanation message about what kind of methods a method category represents. However, we noticed that in many situations, the explanation facility was not exploited to its full by the users. They did not look at the explanation messages first, rather they used the 'trial and error' strategy until they found or did not find, the methods they required. In discussion with users after their test sessions, some users admitted that they had been very used to just browsing everything in Smalltalk with the System Browser. This strategy had been recommended by their instructor in their tutorials and had been practised all the time when they were using Smalltalk. From a pedagogical point of view it would surely be more beneficial to users to find a way of helping them to understand the meaning of a method category more easily.

6.3.3 Users' misconceptions of the tasks

This type of error accounts for thirty-one percent of the total errors. These errors could be classified into the following categories:

— false assumptions;

— misconception of operations;

— 'plausible' solutions.

In the next paragraphs, each category of error will be analysed. Table 6-6 below shows the distribution of these errors across the tasks given.

FA= false assumptions; PA= 'plausible' algorithms; MO= misconceptions of operations.																
Subjects	Task 1			Task 2			Task 3			Task 4			Task 5			**Sub-total (per user)**
	FA	PA	MO	FA	PA	MO	FA	PA	MO	FA	PA	MO	FA	PA	MO	
S1																**0**
S2							1							1		**2**
S3									1	1				1		**3**
S4			1				1		1							**3**
S5																**0**
S6									1							**1**
S7		1						1			1					**3**
S8			1				1									**2**
S9																**0**
S10							1									**1**
Sub-total (per task)	**0**	**1**	**2**	**0**	**0**	**0**	**4**	**1**	**3**	**1**	**1**	**0**	**0**	**2**	**0**	

Table 6-6. The distribution of errors.

a) False assumptions

A number of users misunderstood the nature of the tasks and made assumptions which were not originally stated in the tasks and this led to errors.

An example of such an error was made by a user (S2) in completing task 3. This task requires the user to find all classes which have a method that is able to remove in one operation all elements whose values are less than 10. In the task, nothing is stated about the order of the elements in the target collection. The subject thought that as the index of the elements in the collection was *not specified* in the task, the collections should be unordered and hence selected the class category 'Unordered'. Such errors mainly happened in the tasks that gave few clues about certain properties which the users wanted to know in order to select a class category in the 'Class Category Hierarchy' pane. The intention of such tasks was to let users find required components through a method query. In BRRR2, the class categories are used to create a class query so that the retrieval range of the subsequent method retrieval would be narrowed down quickly. If users are not sure about whether the classes sought should belong to a class category, they can just leave it there (i.e. do not need to either 'require' or 'prohibit' it) and continue with a method retrieval. However, several users tended to make an (unnecessary) definite decision as to whether or not they needed a class category. When they could not get needed information from the tasks, they made their own assumptions and these were sometimes wrong.

This could be remedied by stating explicitly in the manual that if users don't have enough information to decide whether they need a class category, they should simply leave it rather than select one which may be wrong and may seriously affect their subsequent method retrievals.

b) Misconception of operations

Several users had misconceptions of basic concepts of the operations in the collection components of Smalltalk. For example, task 3 requires users to find a method to remove in one operation from the original collection of ten numbers all numbers whose values are less than 10. A number of subjects thought that as the collection only contains 10 elements, it should be fixed size. They didn't realize that as the collection should support the 'remove' operation which would reduce the size of the collection, it should be of an arbitrary size. This mistake seems to be caused by users' misunderstanding of the characteristics of the 'remove' operation in Smalltalk components.

c) 'Plausible' solutions

Several users didn't adhere to the requirements of the tasks given, they picked up some components which were not the required ones but would be plausible in a real programming situation. They thought that they could use these components to complete the task, i.e. they could write a program with additional parts, hence retrieved them as the result. For example, subject 7, whilst doing task 3 (i.e. find a method which can remove all elements which are less than 10 in one operation), found the method 'collect:' as the answer. The function of this method is to create a new collection which has the same class as the original collection and elements of it are selected from the original collection based on some user-specified criteria. This subject thought that he could use this method to 'collect' all elements which are equal or greater than 10 and use the result collection as the answer. He therefore retrieved all 'collect:' methods as his result.

The problem with his solution is that the method he found wouldn't actually remove all elements from the original collection. His algorithm could generate a new collection which would contains the same set of elements as that would result from using a correct method: 'removeAllSuchThat:'. However the original object would not be changed, so his solution would fail to fulfil the requirement of the task.

The problem with this kind of error is that the results found are not the correct or optimal answers, though they may be plausible in real programming situations, if some modifications are made to them. In this study, the answers considered acceptable for the tasks set are regarded optimal and all tasks should be completed by one method. In real programming situations, users can achieve a goal in many ways, and they are not restricted by using only one method. They can use several methods to perform an operation which can be done with one optimal method, if they know that method. Therefore, though the components they found may not be optimal and do not satisfy the requirements we set up here, in a real programming situation, users could have worked out a way to complete the task. This affected the retrieval result since the users believed the components they found were the right ones as they thought they could make them work. Ideally one would welcome a way to inform users that the components they have chosen are not optimal, and that still 'better' components exist in the system and that they should continue their search.

6.4 A comparison with BRRR1

From what was observed in the evaluation study, we have seen that BRRR2 has to a certain extent overcome the following problems we found in BRRR1.

In BRRR1, users had difficulties in identifying an appropriate method category to start a method query. To overcome this difficulty, a cross-reference technique is adopted in BRRR2. Users now have much less difficulty in identifying a correct method category to start a method query.

In the evaluation of BRRR1, users had nothing to refer to during the study, and they could not exploit the advantages of the system. In the evaluation of BRRR2, a manual was provided to users. This was welcomed by users and proved to be very valuable in helping users to accustom themselves to the system.

In BRRR1, users often found components by browsing rather than by doing query reformulations. In BRRR2, better training was provided (e.g. using a manual for users' reference; using exercises which have a similar difficult level to that of the tasks); and difficulties in creating a query were reduced (e.g. using the cross-reference technique to help them start a method query, using graphical illustrations to help them understand class descriptors). As a result, in BRRR2, every user was seen using a form of query (re)formulation to find components rather than by just browsing.

6.5 Conclusion and summary

In this chapter, the evaluation study we conducted on BRRR2 has been presented and the results discussed. In summary, BRRR2 seems to have succeeded in helping users finding required components. This is evident from the facts that most tasks were completed successfully (86%) and the subjects quickly learned to use it. It also shows that BRRR2 has to a certain extent successfully overcome the problems users had with BRRR1. Though the users still experienced some difficulties in using the system, our analysis of those problems in previous sections shows that to a large extent the problems can be solved by providing better training to users, by adjusting some aspects of component classification and by improving some parts of the interface design. The results did not indicate any need to make major changes in the design of the system. In the next chapter, we summarize the research done in this project, suggest relatively minor adjustments which could be made to the present prototype BRRR2 and outline work needed to be done in future research in this direction.

Chapter 7 Conclusions and future work

In this final chapter, we summarize the research work reported in this thesis, outline its contributions to research in this area, discuss the limitations of the work, suggest a number of short term extensions to the system and finally indicate some directions for further longer term research.

7.1 Summary of the research

The overall goal of this research has been to investigate ways of helping users find reusable components in object-oriented programming systems. The Smalltalk system was chosen as the target system to explore our ideas. The 'retrieval by reformulation' approach was used as the basis of the research. Adapting this paradigm to the domain of object-oriented programming, we developed two prototype systems for a subset of Smalltalk. After the first prototype tool — BRRR1 was implemented, an empirical, formative evaluation was conducted to identify problems the users encountered. Based on this evaluation, an improved version of BRRR1, called BRRR2 was developed which aimed to overcome the problems found in BRRR1. A second empirical evaluation was then conducted on BRRR2 to test its effectiveness. The formative evaluation of BRRR2 showed encouraging results. It appears that the tool is helpful for non-expert users finding reusable components in Smalltalk. Our research also demonstrates the feasibility of employing the 'retrieval by reformulation' paradigm in the domain of facilitating non-expert users finding required information in object-oriented programming systems.

7.2 Contributions

The contributions of this research are listed below and each of these will be explained in turn:

— Helping users learn and use the Smalltalk system;

— Facilitating software reuse in object-oriented programming systems;

— Extending the applicability of the 'retrieval by reformulation' paradigm into a new domain;

— Using a method of iterative design combined with formative evaluations to design interface systems.

7.2.1 Helping users learn and use the Smalltalk system

The contribution of this research to the area of helping users learn and use Smalltalk can be seen in the following two aspects:

— It provides a new type of tool to facilitate users finding required components in their programming;

— It provides more data on the learnability of the Smalltalk system.

a) A new type of tool

During this research, a new type of tool, BRRR2, has been developed to help users overcome their difficulties in finding reusable components in Smalltalk. Smalltalk is a large and complex system, and reusing existing components in its component library is a recommended programming style in Smalltalk. However, as we discussed at the beginning of this thesis, users have difficulties in finding what they need. The existing System Browser of this system does not provide users with satisfactory help. It only allows users to access components by names and users often have a vocabulary barrier. This problem makes programming in Smalltalk difficult since users may have to spend large amount of time and effort to look for required components in the large Smalltalk hierarchy. Because of such difficulties in finding necessary components in the system, users sometimes give up the search and write their own code whose function actually can be performed by existing components in Smalltalk, if only they could be found (O'Shea, in press). This problem has seriously affected the learnability and usability of this system. The research reported here illustrates a novel way to tackle this problem: users are allowed to query the system to find what they require. This approach reduces users' search space since they only need to study the relevant components, and thus reduce the difficulties they have. With the query mechanism supplied by BRRR2, users look for necessary components by specifying the properties which the required components have (e.g. What kind of classes should they be: ordered? keyed? etc. What kind of methods should the classes contain? What properties should the methods have?). In this way, users can retrieve the required information based on their properties rather than just based on their names. Additionally, in BRRR2, users create a query by reformulation. They can construct a query iteratively and incrementally, using the information provided by the system. They can first create a class query by reformulation, and then further refine it with a method query. Furthermore, the method query itself can also be created by reformulation. From the results of the empirical evaluation of BRRR2, it can be seen that with the help of BRRR2, the users successfully completed the majority of tasks. This seems to show that BRRR2 is helpful to non-expert users in overcoming their difficulties in finding reusable components in the large component library of Smalltalk.

b) More data on Smalltalk's learnability

The evaluation of BRRR2 revealed some problems which users have in learning Smalltalk components. For example, the functions of certain components are not easy to understand and examples on how they are used in programming may be necessary to help users understand these components; users have misconceptions on certain operations of Smalltalk components. This has provided more information on the learnability of this system.

7.2.2 Facilitating software reuse in object-oriented programming systems

In a wider context, this research facilitates the software reuse approach in object-oriented programming systems. The object-oriented programming paradigm has been said to promote the software reuse approach. One of the reasons is because object-oriented programming systems tend to encourage the use of large reusable component libraries. At a certain stage, therefore, these systems need facilities to help users, particularly non-expert users, find reusable components in these large libraries. As discussed in the first chapter of this thesis, the existing tools, however, do not provide satisfactory solutions. Browsing tools like the Smalltalk Browser have the problem we mentioned in the last section. Query tools based on the keyword matching techniques often retrieve components which users don't need or they fail to find the ones users really require. Tools based on structured database query techniques require users to construct a precise query beforehand, which could be a difficult task for non-expert users.

This research illustrates a novel approach, i.e. developing a query tool based on the 'retrieval by reformulation' paradigm, to help non-expert users find reusable components in object-oriented programming systems. As we have discussed in the last section, BRRR2 appears to a certain extent to have overcome the problem facing the browsing tools, since, by using it, users can search for required components within a smaller range. Furthermore, in BRRR2, a query is formed incrementally with the information provided by the system and users do not have to create a precise query in advance. This should reduce users' difficulties in forming a query. The research result seems to indicate that the approach we used could ease users' difficulties in reusing software components. Moreover, although this approach aims to help non-expert users, it also has the potential of helping more experienced users when they need to explore unfamiliar areas of large object-oriented programming systems.

Although the research reported here used Smalltalk as the target system, the design of BRRR2 utilized only such properties of this system as: class; method and inheritance. This approach thus seems to be generalizable across other object-oriented programming systems which have similar properties, such as: C++; Objective-C; Eiffel and Flavor. To build a tool like BRRR2 for these systems, the classes in those systems should be classified into class categories. Then methods in groups of class could be classified into method categories and methods in a particular method category are characterized by a set of attributes-value pairs. The interface of the tool would take a form similar to that of BRRR2, though some extensions of it would be necessary as we will suggest in section 7.3 of this chapter.

7.2.3 Extending the applicability of the 'retrieval by reformulation' paradigm into a new domain

This research extends applicability of the 'retrieval by reformulation' approach into the domain of finding software components in object-oriented programming systems.

'Retrieval by reformulation' (Williams, 1984) is a paradigm of designing interfaces of large information systems. It suggests that the information in a system should be retrieved by iterative, incremental descriptions of the required items. It also proposes using examples to help users construct a description. It has been used successfully in retrieving information in domains with which users are familiar (e.g. literature and personnel information). In this research, based on this paradigm, two prototype systems, BRRR1 and BRRR2 have been developed for an object-oriented programming system: Smalltalk. The empirical evaluation of BRRR2 showed positive results. This demonstrates the feasibility of using this paradigm in the domain of helping users retrieve software components in object-oriented programming systems. Thus an important contribution of this research is that it extends the applicability of this paradigm into a new domain.

7.2.4 Using the method of iterative design combined with formative evaluations to design interface systems

The development of software in this research used the method which combined iterative design with formative evaluations. First of all, a prototype query tool, BRRR1 was developed based on the 'retrieval by reformulation' paradigm. After that, an empirical, formative evaluation was conducted to test the effectiveness of this first prototype. Problems of the system were identified and this information was then used to the development of the second prototype system: BRRR2. This approach appears to have worked to good effect. It is very difficult to design a user interface right the first time, since it is hard for the designers to foresee all problems users would have with the system (Nielson, 1992). It is thus necessary to design such interfaces iteratively, with the assistance of empirical tests. The formative evaluation method chosen in this research (e.g. video tape, think aloud, test tasks) seems to have achieved satisfactory results. In the formative evaluation of BRRR1, problems which were not realized during its design were found. These ranged from the problems on the design of the system itself to that on the organization of the evaluation study. Our experience showed that this had provided invaluable information to both the design of BRRR2 and the subsequent evaluation of this system. This experience thus provides positive evidence of the advantages of using the approach of iterative design combined with formative evaluations to develop interfaces of BRRR2's type.

7.3 Further work

Though our research has shown some promising results, it has some limitations. In this section, we outline the limitations of this research work and indicate directions for future work. We describe the further work in terms of short term work and long term work and present them in turn below.

7.3.1 Short term extensions

a) The immediate steps which need to be taken are:

i. Several menu commands caused users confusion, they need to be changed as we discussed in section 6.3.1 of chapter 6.

ii. Several method descriptors also need to be adjusted for the same reason.

iii. As we discussed in chapter 6, some components are complex and difficult to understand, we should include examples of how these components are used in real programming situations into the system to facilitate users' understanding of their functions. Before this can be done, it may be necessary to first investigate more fully which components are most likely to cause users difficulties in comprehension.

b) Presently, the system cannot process disjunctive queries, i.e. descriptors connected by the logical operator 'or'. This restricts the expressiveness of the queries which can be processed by the system. In a further implementation, this facility should be incorporated so as to increase the types of query which can be expressed by users.

c) The system currently contains only a subset of the classes in Smalltalk, i.e. the Collection classes, and users can only query the Collection classes. In further implementations, the system should be extended to include all classes in Smalltalk. To deal with that situation, it would be necessary to extend the current interface of the system. This may be done with some form of the 'fisheye' technique as we discussed in section 5.3.1 of chapter 5. So, for example, users can first specify which kind of class (for example, graphical classes; interface classes) they are interested in and consequently want to further investigate their functions by queries. They can then query to find the components they need in a similar way to that currently used in BRRR2. In practical terms, the classification of all Smalltalk components, however, would be a time-consuming task; extending the interface may require a Smalltalk programmer about two to three month to complete.

d) The system as it stands cannot accommodate user-defined components. It can only let users retrieve existing Smalltalk components in its library. To be more helpful for users' programming, the system must be able to deal with user-defined components. To do so, the system should incorporate an 'edit' facility, so that the component can be added in the way of 'editing by reformulation'. To add a new class into the system library, users need to select a class category (or several class categories) to which they believe the class should belong, and then use a menu command to add the class to the class category (or categories). Similarly, to add to the system library a new method in the class which they have just added to the system, users may retrieve a method which has some similarities to the new method they want to put in and use the retrieved method as a template. They can use the method category (or categories) obtained from the description of the retrieved method and add the new method into the category (or categories) with a menu command. They can then fill the method attributes with values. The values may be taken from the retrieved method or from other existing values in the system (by using the alternative value list as they do in a querying process) or may be some values of the users' own, if none of the existing ones in the system are satisfactory. In this way, user-defined components can be put into the system and then be treated in the same way as built-in system components. In practical terms, this could be completed by a Smalltalk programmer in three or four weeks time.

e) The system should be integrated with the original Smalltalk system so that users can program directly with the components they have found. They would be able to take advantage of the original Smalltalk's System Browser (for example, inspect the code of components directly). In addition, they should be able to investigate the functions of some components by directly running or experimenting with them to see the results, as they would do in the original Smalltalk environment.

So far, we have discussed the limitations of the implementation of the system. There are also some limitations of the empirical study we conducted on BRRR2. The study was carried out with relatively few subjects and in the study, users were asked to complete tasks which were designed by the experimenter rather than defined by the users themselves. To evaluate the effectiveness of our approach better, further empirical studies should be done. As a comprehensive empirical study which would evaluate the effectiveness of the next BRRR prototype may itself be a large project, this will be described in the next section.

In this section, we have outlined the limitations of our research and suggested the work needed to be carried out in an immediate future. In the next section, we indicate several possible directions for further research.

7.3.2 Longer term research

In this section, we suggest further research work in the following areas:

approximate matches;

empirical studies;

applying our approach to other object-oriented systems.

a) Approximate matches

An aspect of the system which needs further investigation is that of the degree of exactness necessary in the retrieval process. BRRR2 currently can only retrieve components which match exactly the descriptors specified in users' query. It would be more helpful if the system could show users the components which match the query within approximate bounds and rank the retrieved components according to the similarities between the components and the query. This is because sometimes users may not be quite satisfied with a value of a particular attribute but may want something which is close or similar to that value. Therefore, it would be beneficial to users if the system could deal with approximate queries specified by them and show them all the relevant components. Some work done in the area of approximate matching may be used as a starting point. For example, for the structured database, Motro (1988) has extended relational database systems to include a 'similar-to' comparator so that users can retrieve data which are similar to a specified one. For unstructured databases, Jones (1986) developed a system to retrieve files in a file system which is based on the approach of spreading activation. This approach has also been adopted by Croft et al. (1989), Cohen et al. (1987) and Rau (1987). These may be fruitful paths to explore. In addition, in Smalltalk, there are some classes which are related to each other and they cooperate to complete certain tasks. One such example is the 'model-view-controller' mechanism of Smalltalk in which three types of classes 'Model', 'View' and 'Controller' cooperate to form and manipulate a window. These kinds of relationships should also be expressed in some way in the query and retrieval to facilitate users finding required components. In order to to combine these factors to perform effective retrieval, more fundamental research needs to be carried out.

b) Further empirical studies

We mentioned in the last section that further empirical evaluations need to be done to guide further development of our approach. One study would re-examine the existing data obtained from the evaluation of BRRR2 to examine the process by which users completed the tasks. The study would focus on the possible differences between users who found the answers to the tasks by using the browsing approach and users who found the answers by using the query reformulation approach. This could be done by looking at the paths each user taken to find the answer to each task (i.e. what classes or methods had a user browsed before an answer was found? Did a user find the answer completely by query reformulation?). The results of such a study may provide further information to the development of the next version of BRRR.

One other empirical study may be to test the next implementation of BRRR with a larger number of subjects comparing users' performance with the BRRR system to that of users allowed access only to Smalltalk's System Browser. The users would be divided into two groups: an experimental group and a control group. Each group of users would be asked to complete the same set of tasks. The experimental group would complete the tasks with the help of BRRR and the control group with only the original Smalltalk's System Browser. The performance results can then be used for comparison. However, before these evaluations are carried out, it would be necessary to extend the software itself as we have suggested in section 7.3.1 of this chapter to reflect a more realistic programming situation. This will be more effective when the system contains more classes than at present.

To evaluate the effectiveness of our approach more rigourously, a large scale empirical study may be necessary. This study would examine how expert Smalltalk programmers use a well-developed version of BRRR in a real programming situation to establish how far this tool might promote code reuse.

c) Use our approach in other OOP systems

Finally, this research has been carried out with Smalltalk as a target system. It does not however rely on the special features of Smalltalk other than those shared by other class-based object-oriented programming systems such as: Objective-C, C++, Eiffel, Flavor. It is thus reasonable to believe that the approach which we have used is readily generalizable across these object-oriented programming systems. Further research should investigate this belief and test the applicability of the paradigm to these object-oriented programming systems.

References

(Papers marked with an asterisk (*) have been produced in the course of this research. All others are referenced in the text of this thesis.)

Arnold, S. P. and Stepoway, S. L. (1988). The REUSE system: Cataloging and retrieval of reusable software. *Tutorial: Software reuse: emerging technology*. IEEE Computer Society, EHO278-2, pp. 138-141.

Biggerstaff, T. and Charles, R. (1989). Reusability framework, assessment, and directions. In Biggerstaff, T. and Perlis, A. (eds). *Software Reusability, vol.1: concepts and models*. Addison-Wesley, Reading, MA, pp. 1-17.

Blair, G., Gallagher, J., Hutchison, D. and Shepherd, D. (eds) (1991). *Object-oriented languages, systems and applications*. Pitman Publishing, London.

Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F. (1986). CommonLoops: Merging Lisp and object-oriented programming. *Proceedings of OOPSLA'86*. Portland, Oregon, pp. 17-29.

Böcker, Heinz-Dieter and Herczdg, J. (1990). TRACK — A trace construction kit. *CHI'90 Conference Proceedings*. Seattle, Washington, pp. 415-422.

Boyce, R. F., Chamberlin, D. D., King, W. F. and Hammer, M. M. (1975). Specifying queries as relational expressions: The SQUARE data sublanguage. *Communication of ACM*. Vol. 18, No. 11, pp. 621-628.

Budd, T. (1991). *An introduction to object-oriented programming*. Addison-Wesley, Reading, MA.

Burton, B. A., Aragon, R. W., Bailey, S. A., Koehler, K. D. and Mayes, L. A. (1987). The Reusable Software Library. *IEEE Software*. July, pp. 25-33.

Carroll, J. M., Singer, J. A., Bellamy, R. K. E. and Alpert, S. R. (1990). A View Matcher for learning Smalltalk. *CHI'90 Conference Proceedings*. Seattle, Washington, pp. 431-438.

Cohen, P. R. and Kjeldsen, R. (1987). Information retrieval by constrained spreading activation in semantic networks. *Information processing & Management.* Vol. 23, No. 4, pp. 255-268.

Collins, D. (1990). What is an object-oriented user interface? *Proceedings of the symposium on object-oriented programming, emphasizing practical applications.* Poughkeepsie, NY, September. pp. 269-306.

Conklin, J. (1987). Hypertext: An introduction and survey. *IEEE Computer,* September, pp. 17-41.

Cox, B. J. (1986). *Object-oriented programming: An evaluation approach.* Addison-Wesley, Reading, MA.

Croft, W. B., Lucia, T. J., Cringean, J. and Willett, P. (1989). Retrieving documents by plausible inference: An experimental study. *Information Processing & Management.* Vol. 25, No. 6, pp. 599-614.

Cunningham, W. and Beck, K. (1986). A diagram for object-oriented programs. *Proceedings of OOPSLA'86.* Portland, Oregon, pp. 361-367.

Draper, S. W. (1984). The nature of expertise in UNIX. *Proceedings of INTERACT'84.* Amsterdam, September, pp. 182-186.

Dumais, S. T. (1988). Textual information retrieval. In Helander, M. (ed.) *Handbook of human-computer interaction.* Elsevier Science Publishers B. V. (North-Holland).

Elmasri, R. and Navathe, S. B. (1989). *Fundamentals of database systems.* The Benjamin/Cummings Publishing Company, Inc.

Esp, D. G. (1991). A beginner's experience of Smalltalk-80 for the evolutionary prototyping of an expert system. *Proceedings of the Colloquium on 'Applications and Experience of Object-Oriented Design'.* IEE. Digest No: 1991/018, pp. 2/1-2/6.

Fischer, G. (1987). Cognitive view of reuse and redesign. *IEEE Software,* July, pp. 60-72.

Fischer, G. and Nieper-Lemke, H. (1989). HELGON: Extending the retrieval by reformulation paradigm. *CHI'89 Conference Proceedings*. Austin, TX, pp. 357-362.

Frake, W. B. and Nejmeh, B. A. (1987). Software reuse through information retrieval. *Proceedings of the 21 annual Hawaii international conference on system sciences*. Hawaii, pp. 530-535.

Furnas, G.W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1983). Statistical semantics: Analysis of the potential performance of key-word information systems. *Bell System Technology Journal*. 62, 6, pp. 1753-1806.

Furnas, G. W. (1986). Generalized fisheye views. *CHI'86 Conference Proceedings, human factors in computing systems*. San Francisco, Calif., pp. 16-23.

Furnas, G. W., Landuaer, T. K., Gomez, L. M. and Dumais, S. T. (1987). The vocabulary problem in human system communication. *Communication of ACM*, Vol. 30, No. 11, pp. 964-971.

Gibbs, S., Tsichritzis, D., Casais, E., Nierstrszand, O. and Pintado, X. (1990). Class management for software communities. *Communications of the ACM*, Vol. 33, No. 9, pp. 90-103.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA.

Goldberg, A. (1983). *Smalltalk-80: Interactive programming environment*. Addison-Wesley, Reading, MA.

Gomez, L. M., and Lochbaum, C. C. (1984). People can retrieve more objects with enriched key-word vocabularies. But is there a human performance cost? *Human-computer interaction — Interact'84*. Shackel, B. (Ed.), North-Holland, Amsterdam, pp. 257-261.

Gray, P. D. and Mohamed, R. (1990). *A Practical Introduction to Smalltalk-80*. Pitmans.

Halbert, D. (1986). The learnability of object-oriented programming systems. *Proceedings of OOPSLA'86*, Portland, Oregon, pp. 503-504.

Harrison, W. (1987). RPDE^3: A framework for integrating tool fragments. *IEEE Software,* November, pp. 46-56.

Helm, R. and Maarek, Y. S. (1991). Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. *Proceedings of OOPSLA'91.* Phoenix, Arizona, pp. 47-61.

Jones, G. and Prieto-Díaz, R. (1988). Building and managing software libraries. *Proceedings of the 12th annual international computer software and application conference.* IEEE Computer Society Press, pp. 228-236.

Jones, W. P. (1986). On the applied use of human memory models: the memory extender personal filing system. *International Journal of Man-Machine studies.* 25, pp. 191-228.

Kaehler, T. and Patterson, D. (1986). *A taste of Smalltalk.* Hayden Book Co, U.S.A..

Keene, S. E. (1989). *Object-oriented programming in Common Lisp.* Addison-Wesley, Reading, MA.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of object-oriented programming.* August/September, pp. 26-49.

Lalonde, W. R. and Pugh, J. P. (1990). *Inside Smalltalk,* Vol I. Prentice-Hall. Inc.

*Li, Y. and O'Shea, T. (1990). BRRR: A tool for facilitating users' navigation in Smalltalk-80. *Proceedings of the people and computer interaction systems students' workshop.* Computing Department, The Open University, Milton Keynes, pp. 35-58.

*Li, Y. and O'Shea, T. (1990). BRRR: A tool for facilitating users' navigation in Smalltalk-80. *Proceedings of the symposium on object-oriented programming, emphasizing practical applications.* Poughkeepsie, NY. pp. 175-189.

*Li, Y. (1991). Helping users find reusable components in Smalltalk-80. *Proceedings of PEG 91, Knowledge Based Environments for Teaching and Learning*, Rapallo (Genova), Italy, May 31-June 2.

*Li, Y. (Forthcoming). Finding reusable components in Smalltalk-80. *Computers & Education.*

Maarek, Y. S., Berry, D. M. and Kaiser, G. E. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*. Vol. 17, No. 8, pp. 800-813.

Meyer, B. (1987). Reusability: The case for object-oriented design. *IEEE Software,* March, pp. 50-64.

Meyer, B. (1988). *Object-oriented software construction.* Prentice-Hall Inc.

Moon, D. A. (1986). Object-oriented programming with Flavors. *Proceedings of OOPSLA'86,* Portland, Oregon, pp. 1-8.

Motro, A. (1988). VAGUE: A user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems.* Vol. 6, No. 3, pp. 187-214.

Nielsen, J. and Richards, J.T. (1989). The experience of learning and using Smalltalk. *IEEE Software,* May, pp. 73-77.

Nielson, J. (1992). The usability engineering life cycle. *IEEE Computer,* March, pp. 12-22.

Norman, D. A. and Bobrow, D. G. (1979). Descriptions: An intemediate stage in memory retrieval. *Cognitive Psychology.* 11, pp. 107-123.

O'Shea, T. (in press). Why object-oriented programming is hard to learn. *Human computer interaction.*

Ossher, H. (1990). Multi-dimensional organization and browsing of object-oriented systems. *Proceedings of 1990 International Conference on Computer Languages.* New Orleans, Louisiana, March 12-15, pp. 128-135.

Patel-Schneider, P. F., Brachman, R. J. and Levesque, H. J. (1984). ARGON: knowledge representation meets information retrieval. *Fairchild technical report 654,* Schlumberger Palo Alto Research, September.

Pinson, L. J. and Wiener, R. S. (1988). *An introduction to object-oriented programming and Smalltalk.* Addison-Wesley, Reading, MA.

Pintado, X. (1990). Selection and exploration in an object-oriented environment: the affinity browser. In Tsichritzis, D., (ed.) *Object management.* Centre Universitaire d'Informatique, Universite de Geneve, pp. 79-88.

Prieto-Díaz, R. and Freeman, P. (1987). Classifying software for reusability. *IEEE Software,* January, pp. 6-16.

Prieto-Díaz, R. (1991). Implementing faceted classification for software reuse. *Communication of ACM.* Vol. 34, No. 5, pp. 89-97.

Ramamoorthy, C. V. and Sheu, P. C. (1988). Object-oriented systems. *IEEE Expert,* Fall, pp. 9-15.

Rau, L. F. (1987). Knowledge organization and access in a conceptual information system. *Information Processing & Management.* Vol. 23, No. 4, pp. 269-283.

Remde, J. R., Gomez, L. M. and Landauer, T. K. (1987). SuperBook: An automatic tool for information exploration — hypertext? *Proceedings of Hypertext'87.* University of North Carolina, Chapel Hill, North Carolina. November 13-15, pp. 175-188.

Rosson, M.B. and Carroll, J. M. (1990a). Climbing the Smalltalk mountain. *SIGCHI Bulletin,* Vol. 21, No. 3, pp. 76-79.

Rosson, M.B., Carroll, J. M. and Bellamy, R. K. E. (1990b). Smalltalk scaffolding: A case study of minimalist instruction. *CHI'90 Conference Proceedings.* Seattle, Washington, pp. 423-429.

Rubin, K. S. (1990). Reuse in software engineering: An object-oriented perspective. *Digest of papers. COMPCON90.* 31 IEEE Computer Society International Conference. San Francisco, February 26-March 2.

Salton, G and McGill, M. J. (1983). *Introduction to modern information retrieval.* Computer series, McGraw-Hill, New York.

Saunders, J. (1989). A survey of object-oriented programming languages. *Journal of Object-Oriented Programming.* March/April, pp. 5-11.

Schank, R. C. (1972). Conceptual Dependency: A theory of natural language understanding. *Cognitive Psychology.* 3, pp. 552-631.

Stroustrup, B. (1986). *The C++ programming language.* Addison-Wesley, Reading, MA.

Taenzer, D., Ganti, M. and Podar, S. (1989). Problems in object-oriented software reuse. *Proceedings of ECOOP'89.* Nottingham, U.K. July 10-14, pp. 25-38.

Tesler, L. (1985). *Object Pascal report.* Apple Computer, Santa Clara, CA.

Tou, F. (1982). RABBIT: a novel approach to information retrieval. *M.S. Thesis,* Massachusetts Institute of Technology. MA.

van Rijsbergen, C. J. (1979). *Information retrieval.* 2nd edition, Butterworths.

Walker, J. H. (1987). Document Examiner: Delivery interface for hypertext documents. *Proceedings of Hypertext'87.* University of North Carolina, Chapel Hill, North Carolina. November 13-15, pp. 307-324.

Waltz, D. L. and Goodman, B.A. (1977). Writing a natural language data base system. *Proceedings of 5th International Joint Conference on Artificial Intelligence.* pp. 144-150.

Waltz, D. L. (1978). An English language question answering system for a large relational database. *Communications of ACM.* Vol. 21, No. 7, pp. 526-539.

Williams, M. D. and Hollan, J. D. (1981). The process of retrieval from very long term memory. *Cognitive Science* 5, pp. 87-119.

Williams, M. D. (1984). What makes RABBIT run? *International Journal of Man-Machine Studies*. 21, pp. 333-352.

Wood, M. and Sommerville, I. (1986). A software components catalogue. In Davies, R. (ed.) *Intelligent information systems: Progress and prospects*. Ellis Horwood Limited. pp. 13-32.

Wood, M. and Sommerville, I. (1988). An information retrieval system for software components. *SIGIR FORUM,* Spring/Summer, Vol. 22, Issue 3, 4, pp. 11-25.

Wu, C. T. (1990). A better browser for object-oriented programming. *Journal of object-oriented programming,* November/December, pp. 22-29.

Appendix A Exercises used in the evaluation of BRRR1

1) Find all the collections which can keep its elements in some order.

2) Find all the collections in which any element does not occur more than once.

3) In some collections, elements can not be accessed according to their index. Find all these collections.

4) Suppose there are five objects. Find all candidate collection into which these objects can be stored, and with a method of the collections you can access the third object you stored (You should find these methods first).

5) Find all methods that can be used to insert more than one element into ordered collections.

6) Some collections have methods with which you can access individual elements according to the positions of these elements. Find all these methods.

Appendix B The manual used in the evaluation of BRRR2

BRRR Reference Booklet

Contents

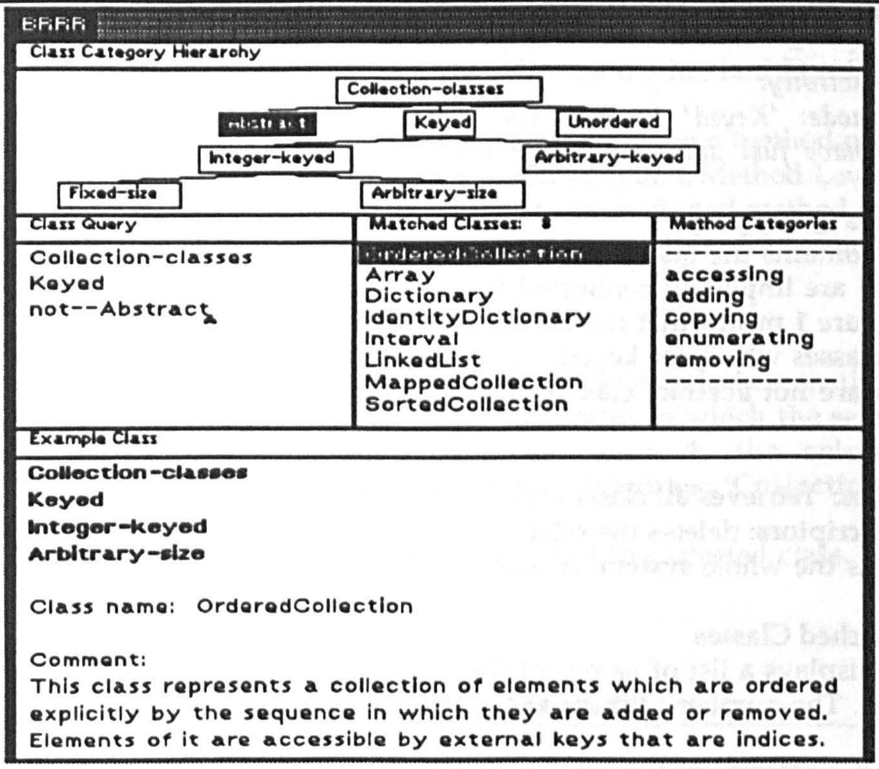
Section 1. Introduction

This system is called BRRR (BrowseR for Retrieval by Reformulation). It is a tool used to help users find classes and methods they require in their programming in Smalltalk-80.

BRRR is a browsing tool with a query capability. When you need to find classes (or methods) in Smalltalk, you can use BRRR to construct a query, BRRR will provide you with a list of candidate components (classes or methods) which match the query, you then examine the list to find the required components. If the number of the components is large, you can use the information provided by BRRR to refine (reformulate) your previous query, then request BRRR to do a further retrieval. This process may be repeated until you find the satisfactory components. In BRRR, the information needed to construct a query is provided by the system, you just manipulate the information with a set of options presented in pop up menus. There is no need for you to type in anything.

BRRR has two types of query windows: the Class Level Query window and the Method Level Query Window. These are used to retrieve classes and methods respectively. In the coming sections we describe each of these two windows in detail and show how they may be used.

Section 2. The Class level Query Window



```
Class Query             Matched Classes: 8       Method Categories
Collection-classes      OrderedCollection        -------------
Keyed                   Array                    accessing
not--Abstract           Dictionary               adding
                        IdentityDictionary       copying
                        Interval                 enumerating
                        LinkedList               removing
                        MappedCollection         -------------
                        SortedCollection
```

```
Example Class
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name:  OrderedCollection

Comment:
This class represents a collection of elements which are ordered
explicitly by the sequence in which they are added or removed.
Elements of it are accessible by external keys that are indices.
```

figure 1 The Class Level Query window

This window consists of the following five panes, each of which we will subsequently describe in more detail

i	**Class Category Hierarchy**	(the pane at the top)
ii	**Class Query**	(the leftmost pane below the Class Category Hierarchy pane)
iii	**Matched Classes**	(the pane at the right of the Class Query pane)
iv	**Method Categories**	(the pane at the right of the Matched Classes pane)
v	**Example class**	(the pane at the bottom of the window).

i **Class Category Hierarchy pane**

Displayed in this pane is a tree, each of its nodes represents a class category. A class category is similar to that in the original Smalltalk. Each contains several classes having some common properties, e.g. the classes in the category 'Collection-classes' can all be used as containers for other objects.

Class categories in BRRR are organized into a hierarchical structure: a category may have sub-categories which contain classes having more specific properties (this structure is somewhat different from that of the original Smalltalk). For example, the category 'Keyed', which is a sub-category of 'Collection-classes', contains classes which are collection classes and 'Keyed' i.e. their elements are accessible by external indices.

Class categories are used to construct queries to retrieve classes. They can be manipulated with options presented in a yellow button menu. You first click the node of the category you wish to highlight, then manipulate it with menu options. The menu options are:

require: sends the name of a selected category to the Class Query pane. You use this option when you think that the classes you need to retrieve should have the properties characterized by the category name.

prohibit: prefixes 'not--' to the name of a selected category and sends it to the Class Query pane. This is the opposite case to the **'require'** option.

explain: explains the properties that the classes in that category have.
show classes: lists, in another menu, the names of the classes in that category.

Suggested Activity:
Select the node: 'Keyed' in the Class Category Hierarchy pane and try the four options we have just discussed to see what happens.

ii Class Query pane

This pane contains the query you constructed for retrieving classes. All descriptors in this pane are implicitly connected by the logical operator 'and'. For example, the query in figure 1 means that the classes which are to be retrieved are:
collection classes which are keyed, i.e. their elements are accessible by external keys, and which are not abstract classes. The yellow button menu options for this pane are:

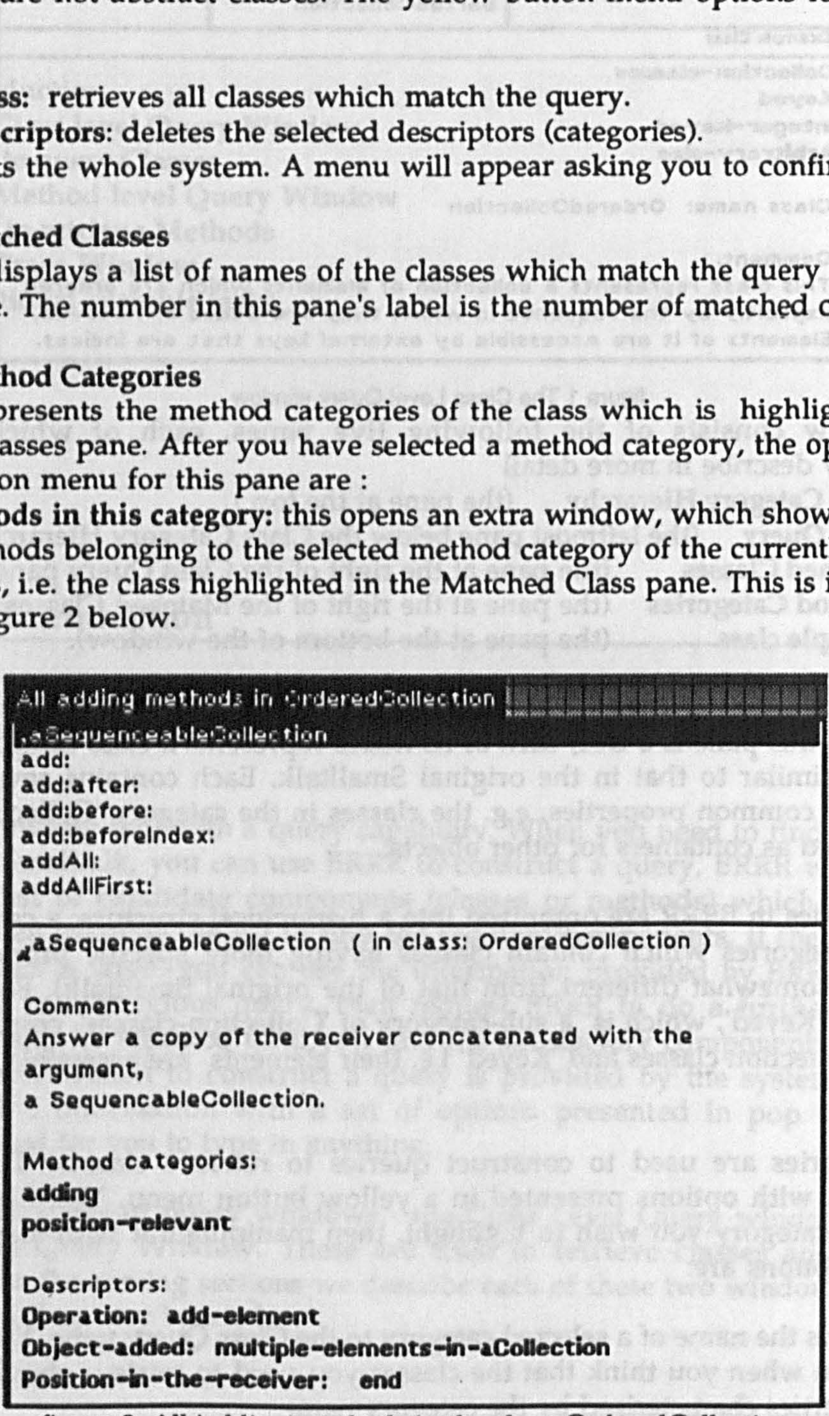
retrieve class: retrieves all classes which match the query.
remove descriptors: deletes the selected descriptors (categories).
reset: resets the whole system. A menu will appear asking you to confirm this.

iii Matched Classes

This pane displays a list of names of the classes which match the query in the Class Query pane. The number in this pane's label is the number of matched classes.

iii Method Categories

This pane presents the method categories of the class which is highlighted in the Matched Classes pane. After you have selected a method category, the options in the yellow button menu for this pane are :
show methods in this category: this opens an extra window, which shows the methods belonging to the selected method category of the currently selected class, i.e. the class highlighted in the Matched Class pane. This is illustrated in figure 2 below.


```
All adding methods in OrderedCollection
,aSequenceableCollection
add:
add:after:
add:before:
add:beforeIndex:
addAll:
addAllFirst:

,aSequenceableCollection  ( in class: OrderedCollection )

Comment:
Answer a copy of the receiver concatenated with the
argument,
a SequencableCollection.

Method categories:
adding
position-relevant

Descriptors:
Operation:  add-element
Object-added:  multiple-elements-in-aCollection
Position-in-the-receiver:   end
```
figure 2. All 'adding' methods in the class: OrderedCollection

other method categories: this option shows method categories which are not in the currently selected class but are in other classes which also match the current query. This option also allows you to open a Method Level Query window to query these other methods (we shall discuss the Method Level Query window in sec. 4).

explain: this option explains the properties of the methods in a method category.

constructing method query: this option allows you to open a Method Level Query window for querying all methods which are in a selected method category and which are contained in matched classes (again, this will be discussed in sec. 4).

v **Example class**

The text in this pane describes the function of the selected class in the Matched Classes pane. The boldface parts are the class categories to which the selected class belongs. For example, if we look back to figure 1, the selected class '**OrderedCollection**' belongs to all the following categories: '**Collection-classes**'; '**Keyed**'; '**Integer-keyed**' and '**Arbitrary-size**'.
The text after the 'Comment:' specifies the function of the selected class.

Section 3. The procedure of querying classes:

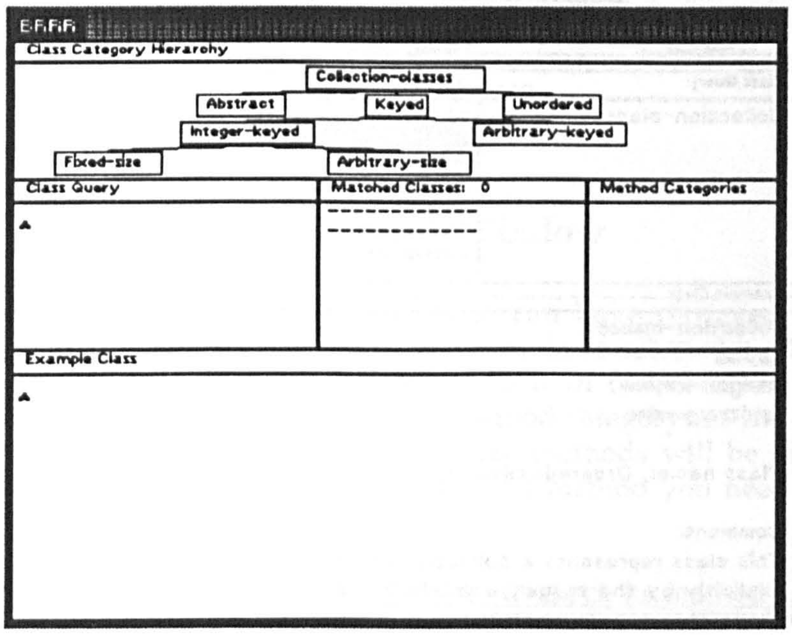


figure 3. The start situation

i Starting with the Class Category Hierarchy pane, '**require**' the categories to which you think the class you need should belong, '**prohibit**' those to which you think it does not. Use the BRRR help facilities '**show classes**' or '**explain**' if you are not certain about what kind of classes a category may contain.

ii After selecting some class categories, retrieve classes (using the '**retrieve class**' in the Class Query pane).

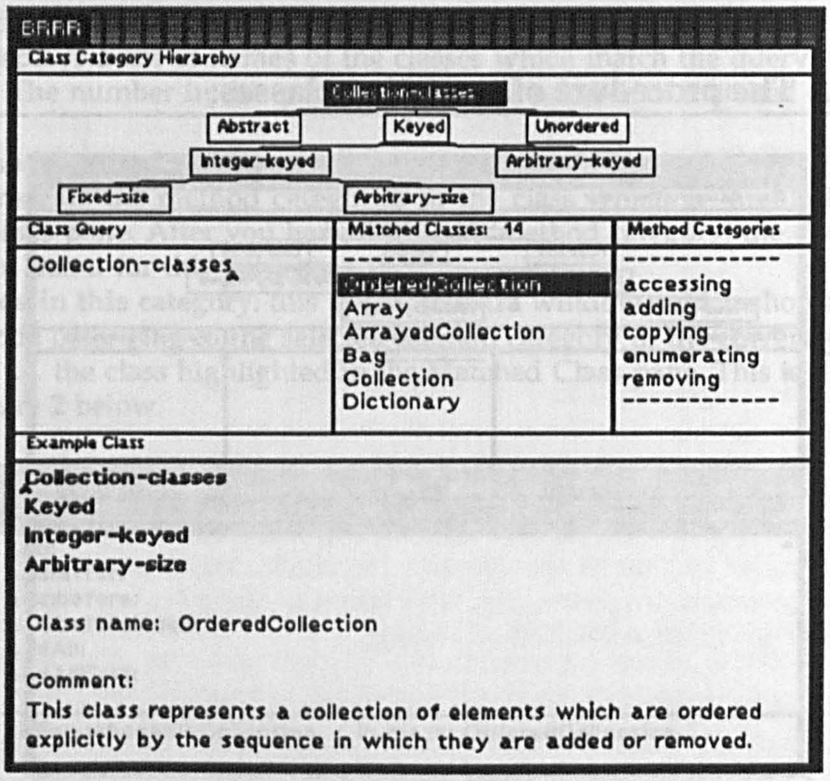
iii Examine the candidate classes to find the one you need. During this process, if you find a new class category which is not in the current query but in which the class should be, always '**require**' it to refine your current query, and then do a new retrieval.

On the following page there is an example task to illustrate using BRRR to find classes.

Example Task 1: Suppose you are looking for a collection to store a group of numbers. You want to arrange the numbers in the collection as a sequence so that you can access them through integer indices e.g. access the 'first' or 'second' or 'nth' element. Find all candidate classes.

Search process:

i At the beginning, the system is as shown in figure 3. You need a collection, so the class you require should be one of the collection classes. You select the category 'Collection-classes', and then select the '**require**' option from its yellow button menu. After the operation, the category name appears in the Class Query pane. You then ask the system to do a retrieval by selecting the option '**retrieve class**' from the menu in the Class Query pane. After the operation, all panes except the Class Category Hierarchy are updated, as shown in figure 4 below.



Class Query

Collection-classes

Matched Classes: 14

OrderedCollection
Array
ArrayedCollection
Bag
Collection
Dictionary

Method Categories

----
accessing
adding
copying
enumerating
removing

Example Class

Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name: OrderedCollection

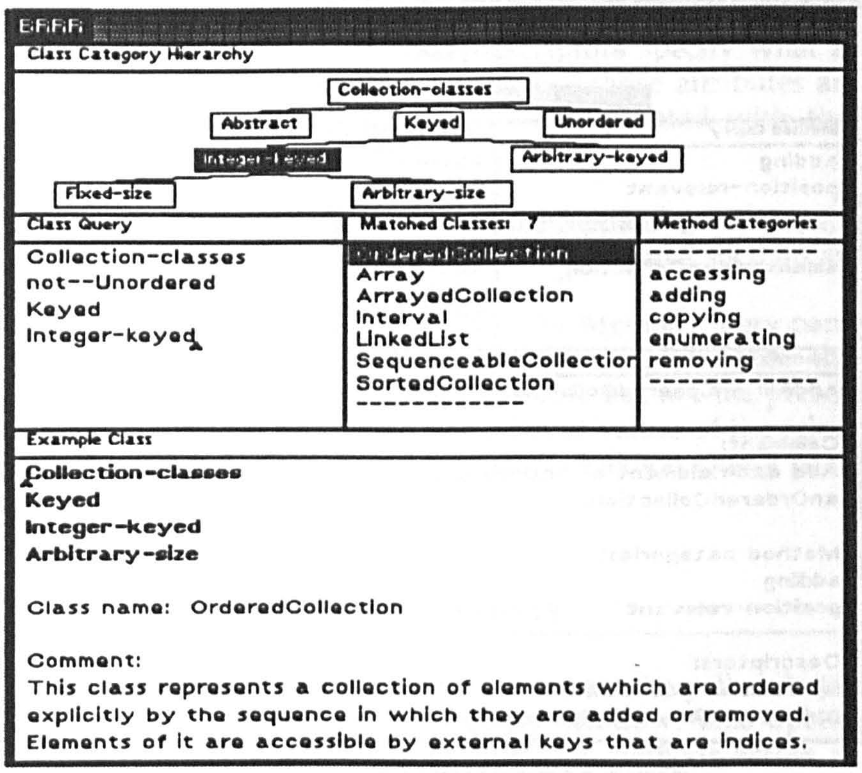
Comment:
This class represents a collection of elements which are ordered explicitly by the sequence in which they are added or removed.

figure 4. All 'collection' classes

All the collection classes BRRR has found are shown in the Matched Class pane, and the first one-'OrderedCollection' is highlighted. Its method categories appear in the Method Categories pane. The text describing the function of this class is shown in the Example Class pane.

ii. The collection you need should maintain its elements as a sequence, therefore it should not be in the category 'Unordered'. So you select the category 'Unordered' and '**prohibit**' it. In addition, the elements in the collection should be accessible by integer indices, so you '**require**' the category 'Keyed' and then '**require**' the category 'Integer-keyed'. These two categories appear in the Class Query pane. You use the '**retrieve class**' again, then BRRR presents you with all classes which are: collection classes [and] not--

Unordered [and] Keyed [and] Integer-keyed (see figure 5). The matched classes shown are the candidates.



```
Collection-classes       OrderedCollection          ------------
not--Unordered           Array                      accessing
Keyed                    ArrayedCollection          adding
Integer-keyed            Interval                   copying
                         LinkedList                 enumerating
                         SequenceableCollection     removing
                         SortedCollection           ------------
                         -------------
```

```
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name:  OrderedCollection

Comment:
This class represents a collection of elements which are ordered
explicitly by the sequence in which they are added or removed.
Elements of it are accessible by external keys that are indices.
```

figure 5. matched classes

[End of Example task 1]

Section 4. The Method Level Query Window

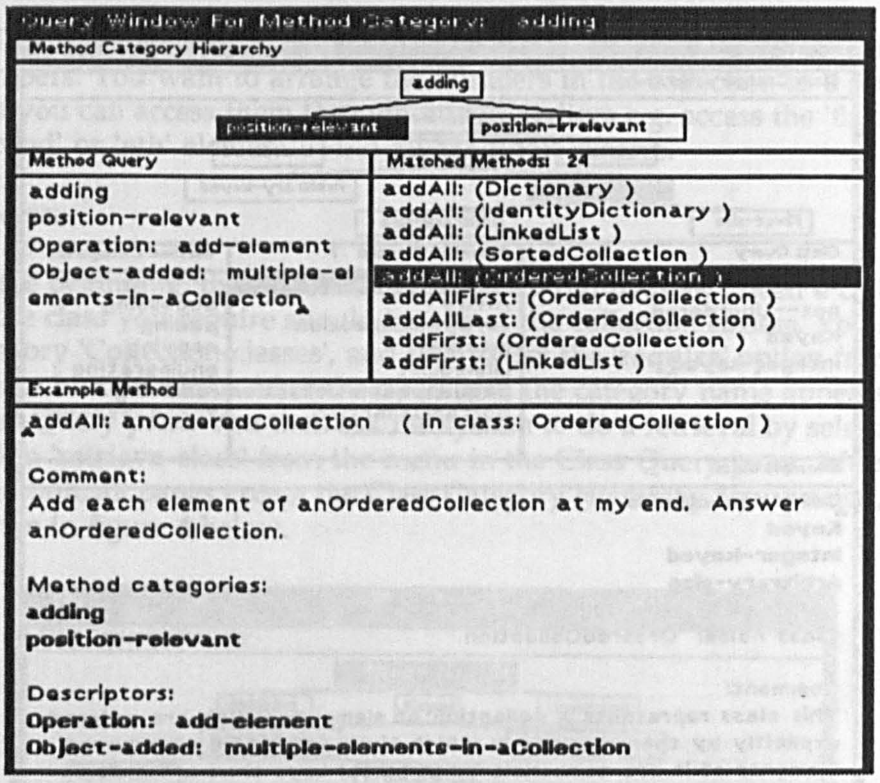
This window is used to query methods. After you have retrieved a number of classes in the Class Level Query window, you may have selected a particular class and wish to examine the methods belonging to it in order to see if it has the function you need. You can get methods in a method category for the selected class by using the menu option '**show method**'. These methods will be displayed in a window as shown in figure 2. However, often the method you need is not in the class you are examining.

<u>In this case you don't have to select another class to check its method as you have to do in the original Smalltalk.</u>

Staying in the *currently selected* class, select a method category in which you think the method you need should be and choose the '**constructing method query**' from the menu to open the **Method Level Query Window.**

The **Method Level Query Window** (see figure 6 below) is similar in structure to the Class Level Query window. It has four panes, which we will subsequently describe in detail.

i **Method Category Hierarchy** (top pane)
ii **Method Query** (below and left of previous pane)
iii **Matched Method** (to the right of 'Method Query' pane)
iv **Example Method** (bottom pane)



```
Query Window For Method Category:   adding
Method Category Hierarchy
                         adding
        position-relevant      position-irrelevant
Method Query                    | Matched Methods: 24
adding                          | addAll: (Dictionary )
position-relevant               | addAll: (IdentityDictionary )
Operation:  add-element         | addAll: (LinkedList )
Object-added:  multiple-el      | addAll: (SortedCollection )
ements-in-aCollection           | addAll: (OrderedCollection )
                                | addAllFirst: (OrderedCollection )
                                | addAllLast: (OrderedCollection )
                                | addFirst: (OrderedCollection )
                                | addFirst: (LinkedList )
Example Method
addAll: anOrderedCollection   ( in class: OrderedCollection )

Comment:
Add each element of anOrderedCollection at my end.  Answer
anOrderedCollection.

Method categories:
adding
position-relevant

Descriptors:
Operation:  add-element
Object-added:  multiple-elements-in-aCollection
```

figure 6. A Method Level Query window

i **Method Category Hierarchy pane:**
Displayed in this pane is the method category tree. Each node of it is a method category. A method category may have sub-categories which represent more specific categories of methods (note that in the original Smalltalk system, a method category has no sub-categories). Its yellow button menu is similar to that of the Class Category Hierarchy pane and similarly its options are:
require; prohibit; explain; show methods.

ii **Method Query pane:**
Similar to the Class Query pane. Its menu options are:

retrieve methods;
remove descriptors;
get the current method: retrieves the currently selected method in the Matched Method pane.
clear: removes *all* descriptors in this pane.
merge into class query: merges the method query you constructed into the Class Query pane so that you can ask the system to retrieve the classes which contain certain kinds of methods.

iii **Matched Method pane:**
Similar to the Matched Class pane, it shows the methods which match the method query.

iv **Example Method pane**
Similar to the Example Class pane. Text in this pane describes the function of the selected method.

The text under the 'Method categories:' shows the method categories to which the method belongs.
The boldface text under the 'Descriptors:' gives the method descriptors. A descriptor has two parts, the part before the colon, ':' is an *attribute* and the part after the colon is the *value* of the attribute. Method descriptors specify what attributes a method has and for that method, what the values of those attributes are. They are used to construct method queries and can be manipulated with the following options in this pane's yellow button menu:

require-this-value sends a selected descriptor to the Method Query pane, this is used when you think that the method required should have such an attribute and corresponding value.
prohibit-this-value sends a selected descriptor to the Method Query pane, however, it prefixes the value with: 'not--'. This is used when you think that the method required should have such an attribute but not the present value.
alternative-values presents for users' selection a list of values of the selected attribute, the selected value is then sent to the Method Query pane in the 'require' or 'prohibit' form.

Section 5. How to retrieve methods

i Use the '**constructing method query**' option from the yellow button menu of the 'Method Categories' pane in the Class Query window. This opens a Method Level Query window which you can use to query all methods across all matched classes in a specific method category.

ii In the Method Level Query window, use a procedure similar to that used in the Class Query pane. However, not only should you manipulate the method categories, but you should also manipulate the *method descriptors* presented in the Example Method pane with the following options described previously in section 4, i.e.
'**require-this-value**' (you need the selected descriptor [attribute and value])
'**prohibit-this-value**' (the value of an attribute is not appropriate).
Especially, do not forget to use:
'alternative-values' (asks the system to provide other values for your selected attribute.)

In summary, make as much use as possible of the information BRRR provides to gradually refine your query until you have the information you require.

Next, we step through an example to show you the method query process.

Example Task 2

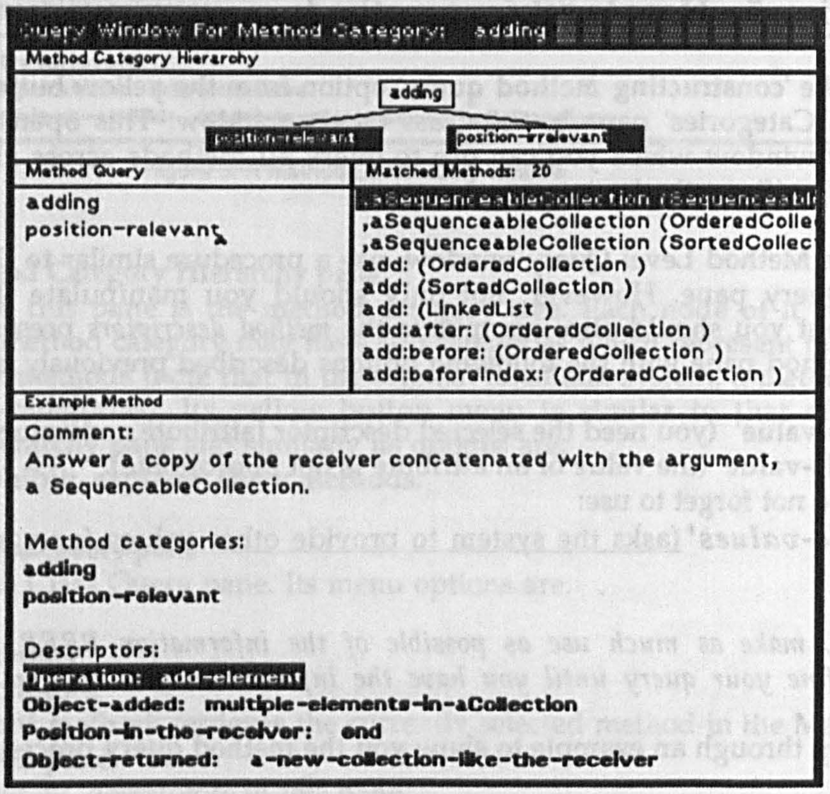
Suppose, extending the Example task 1 given earlier, you wish to find all the candidate classes which have a method that will allow you to put all the numbers you want into the collection at once, ordered automatically according to their values.

Search process:
By using the searching process for task 1, you arrived at 7 candidate classes. (figure 5). Now you want to know which one has the method you required, i.e. the method which can put elements into the collection and order them. You examine the method categories of the 'OrderedCollection' in the

Method Categories pane, and decide that the method should be in the 'adding' category. To see if this class contains such a method, you select the category and from the yellow button menu choose the option **'show methods in this category'**.
A window is opened and all the 'adding' methods in 'OrderedCollection' are listed in it (figure 2). After examining them and finding nothing to match your requirements, you decide to see if there is an 'adding' method in any other matched classes.

To do this, select the **'constructing method query'** option from the menu. A method query window is opened. In its Method Category Hierarchy pane you see that the category 'adding' is further divided into two sub-categories: **'position-relevant'** and **'position-irrelevant'**.

Since you need a method which can sort numbers, you **'require'** the category **'position-relevant'** and then ask for a retrieval. BRRR presents you with all methods which can add elements into a collection and put them into specific positions. The first one in the list- ', aSequenceableCollection' is highlighted (see figure 7).



```
adding
position-relevant
```

```
,aSequenceableCollection (SequenceableC
,aSequenceableCollection (OrderedColle
,aSequenceableCollection (SortedCollec
add: (OrderedCollection )
add: (SortedCollection )
add: (LinkedList )
add:after: (OrderedCollection )
add:before: (OrderedCollection )
add:beforeIndex: (OrderedCollection )
```

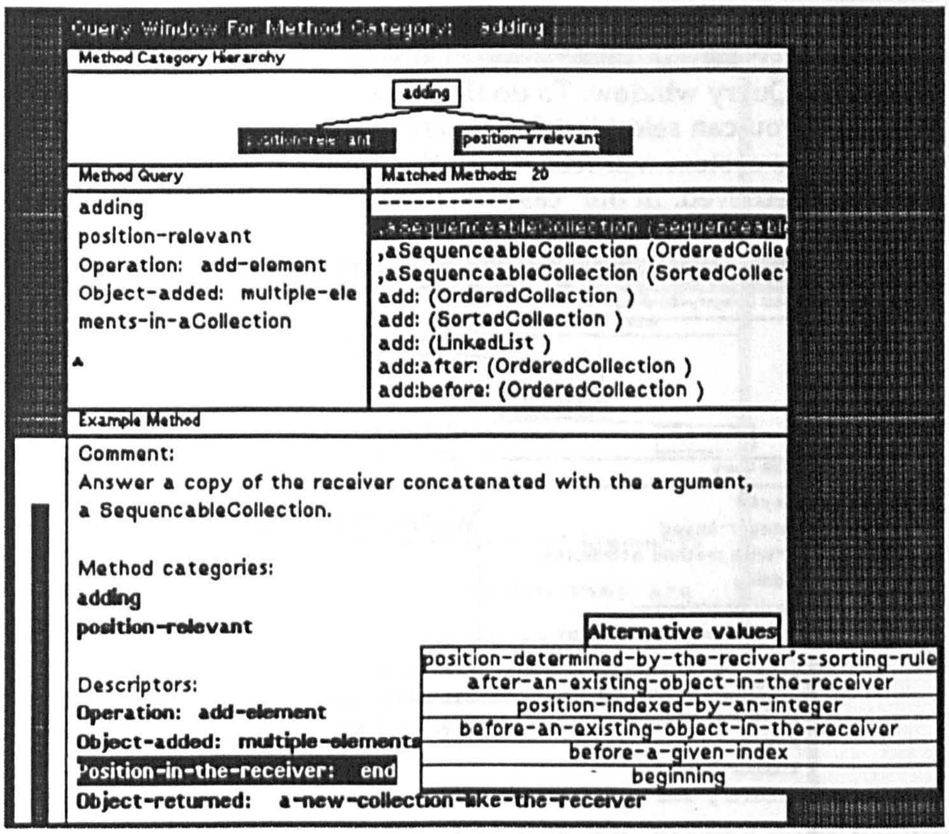
```
Comment:
Answer a copy of the receiver concatenated with the argument,
a SequencableCollection.

Method categories:
adding
position-relevant

Descriptors:
Operation:  add-element
Object-added:  multiple-elements-in-aCollection
Position-in-the-receiver:  end
Object-returned:  a-new-collection-like-the-receiver
```

figure 7. querying methods

Examine the method descriptors in the Example Method pane. You want to put elements into a collection, therefore the operation of the method should be: 'add-element'. To do this you select the descriptor '<u>Operation: add-element</u>' and then use the menu option **'require-this-value'**. You also want to put in several numbers, so you do the same to the descriptor: '<u>Object-added: multiple-elements-in-aCollection</u>'. However, the method you need should not always put an element at the end of the collection, therefore you need to find a more appropriate value for the attribute: 'Position-in-the-receiver:'. To do this, you choose the menu option: **'alternative-values'**, which presents a menu with all alternative values for this attribute (see figure 8).



Query Window For Method Category: adding

Method Category Hierarchy

adding

position-relevant position-irrelevant

Method Query

adding
position-relevant
Operation: add-element
Object-added: multiple-ele
ments-in-aCollection

Matched Methods: 20

```
------------
,aSequenceableCollection (SequenceableC
,aSequenceableCollection (OrderedColle
,aSequenceableCollection (SortedCollec
add: (OrderedCollection )
add: (SortedCollection )
add: (LinkedList )
add:after: (OrderedCollection )
add:before: (OrderedCollection )
```

Example Method

Comment:
Answer a copy of the receiver concatenated with the argument,
a SequencableCollection.

Method categories:
adding
position-relevant

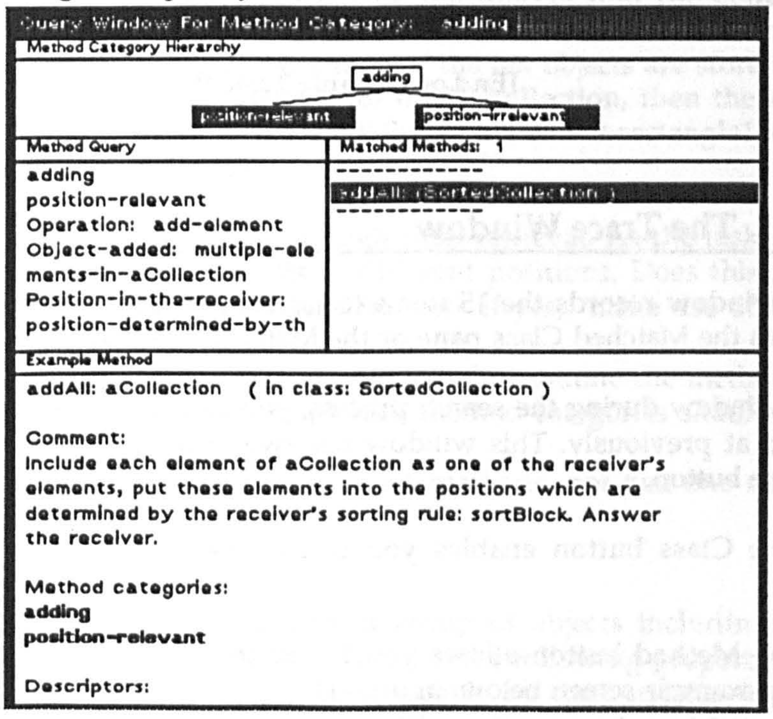
Descriptors:
Operation: add-element
Object-added: multiple-elements
Position-in-the-receiver: end
Object-returned: a-new-collection-like-the-receiver

Alternative values

position-determined-by-the-reciver's-sorting-rule
after-an-existing-object-in-the-receiver
position-indexed-by-an-integer
before-an-existing-object-in-the-receiver
before-a-given-index
beginning

figure 8. alternative values

Among them, you think the 'position-determined-by-the-receiver's-sorting-rule' is the appropriate one, so you select it and **'require'** it and this value along with the attribute is sent to the Method Query pane. As this seems the most promising descriptor, you ask for a retrieval.



Query Window For Method Category: adding

Method Category Hierarchy

adding

position-relevant position-irrelevant

Method Query

adding
position-relevant
Operation: add-element
Object-added: multiple-ele
ments-in-aCollection
Position-in-the-receiver:
position-determined-by-th

Matched Methods: 1

```
------------
addAll: (SortedCollection )
------------
```

Example Method

addAll: aCollection (in class: SortedCollection)

Comment:
Include each element of aCollection as one of the receiver's
elements, put these elements into the positions which are
determined by the receiver's sorting rule: sortBlock. Answer
the receiver.

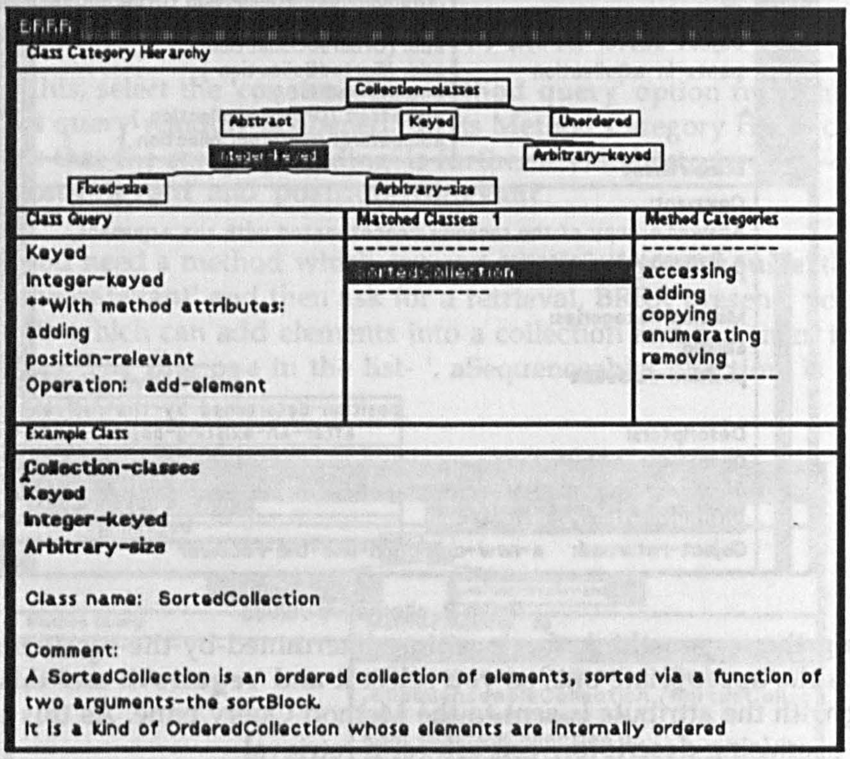
Method categories:
adding
position-relevant

Descriptors:

figure 9. matched method

After this, the matched method *addAll:* in class SortedCollection is shown in the Matched Method pane. The comment indicates that this is what you want (see figure 9 above).

Now, you have found the method and its class. At this point you are going to merge the information about methods with the information in the Class Level Query window. To do this, use the option '**merge into class query**'. Now you can select '**retrieve class**' from the menu in the 'Class Query' pane and the system will retrieve all the classes which contain the methods you have retrieved. In our case the final screen is as in figure 10 below.



```
Keyed
Integer-keyed
**With method attributes:
adding
position-relevant
Operation:  add-element
```

```
------------
SortedCollection
------------
```

```
------------
accessing
adding
copying
enumerating
removing
------------
```

```
Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name:  SortedCollection

Comment:
A SortedCollection is an ordered collection of elements, sorted via a function of
two arguments-the sortBlock.
It is a kind of OrderedCollection whose elements are internally ordered
```

figure 10. matched class

[End of Example Task 2]

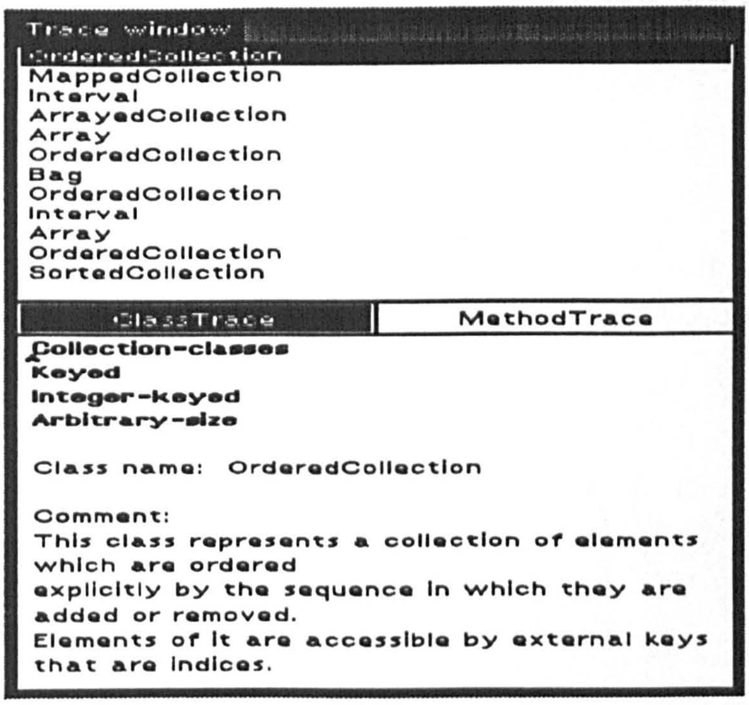
Section 6 The Trace Window

The **Trace** window records the 15 items (class or method) you have most recently selected from the Matched Class pane or the Matched Method pane.

Using this window during the search process, you can check a class or method you have looked at previously. This window has two buttons, **ClassTrace** button and **MethodTrace** button.

Selecting the Class button enables you to see the classes you have previously selected.

Selecting the Method button allows you to see the methods you have previously selected (see example screen below, figure 11).


Trace window

OrderedCollection
MappedCollection
Interval
ArrayedCollection
Array
OrderedCollection
Bag
OrderedCollection
Interval
Array
OrderedCollection
SortedCollection

Collection-classes
Keyed
Integer-keyed
Arbitrary-size

Class name: OrderedCollection

Comment:
This class represents a collection of elements which are ordered
explicitly by the sequence in which they are added or removed.
Elements of it are accessible by external keys that are indices.

figure 11. TheTrace window

Section 7. Additional Example Tasks

The Example Tasks which follow are designed to help you become more familiar with the system. Working through them, the first of which includes some helpful hints, should make you feel at home with BRRR.

Example Task 3.

You are looking for a collection to store ten objects. The collection should have a method such that after you have stored the ten objects into the collection, this method enables you to replace at once the 2nd, 4th and 6th elements of the collection with the string: 'hello'. For example, if the ten objects are stored as: (3 'john' 'fred' 4 6 8 0 rectangle1 string1 10) in the collection, then the result after using this method would be: (3 'hello' 'fred' 'hello' 6 'hello' 0 rectangle1 string1 10).

Find all candidate classes and methods.

(Hints:

1. You should first find some classes through class retrieval. As the task stated, you should be able to replace the elements at different positions. Does this imply that you can access elements through external indices? If yes, make use of this fact to construct your query.
2. After you have retrieved some classes, you should examine the methods in these classes. From what is stated in the task, which method categories should you select to inspect?
3. Remember to find all candidate classes and methods, not just one method in a class.)

Example Task 4

You are looking for a collection to store a group of objects including: numbers, strings, rectangles, etc., the collection should have the following properties:

The elements in the collection should be accessible by their indices (e.g. you can access the first element, the 2nd element,...). The collection should have a method which you can use to delete the element at the end of the collection (note: after you delete the element at the end of the collection, the length of the collection should be 1 less than before).

Find all candidate classes and methods.

[End of the manual]