OCB: An Object/Class Browser for Java

Graham N.C. Kirby and Ron Morrison

School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland
{graham, ron}@dcs.st-and.ac.uk

Abstract

This paper describes an interactive browser used for exploring the structure of Java objects and their classes. It is implemented in Java and uses JDK 1.1 core reflection classes to discover details of the objects passed to it. The initial motivation for development arose from the need to browse persistent Java stores; the browser may also be useful as part of a symbolic debugging or visualisation tool.

1 Introduction

The provision of orthogonal persistence for Java, e.g. [ADJ+96, DHF96, GN96, MCK+96], allows the programmer to create potentially large persistent stores of Java objects. There are a number of ways of discovering the contents of these stores, including interactive browsing, writing programs which navigate inter-object references, and the use of a query language. All of these, and others, will probably be required in practice. This paper describes a visualisation tool called OCB (Object/Class Browser) which addresses the first requirement, supporting the interactive display of Java objects and classes, and the navigation of references linking them.

The main design goals for the initial version of OCB were:

- to provide a simple and clean user interface;
- to produce an implementation quickly; and
- to implement the browser using only standard Java for maximum portability[1].

The OCB browser was designed in response to a need identified by the developers of the PJama persistent Java implementation [ADJ+96]. It was quickly recognised, however, that most of its facilities would also be useful in conventional Java systems and other persistent versions. All OCB facilities other than access to persistent roots, and in some cases method invocation, will work with any Java system. Persistent root access for other persistent versions can be added simply on a per-system basis; the details depend on the model of persistence provided.

The remainder of this paper contains a summary of related work in Section 2, descriptions of the user and program interfaces in Sections 3 and 4, an outline of the implementation in Section 5 and a discussion of avenues for further development in Section 6.

2 Related Work

There are several aspects of a persistent object store which it may be useful to visualise, including:

- the states of the objects in the store and the graph of references linking them;
- the states of the threads running in the store; and
- the class hierarchy or type structure associated with the objects in the store.

Other systems support these requirements for Java and other languages to varying degrees. The OCB browser in its current implementation addresses only the first and third requirements, although it is planned to extend it to include thread states in a future version.

[1] No native methods are used, although OCB is not 100% Pure Java™ for reasons explained in Section 6.

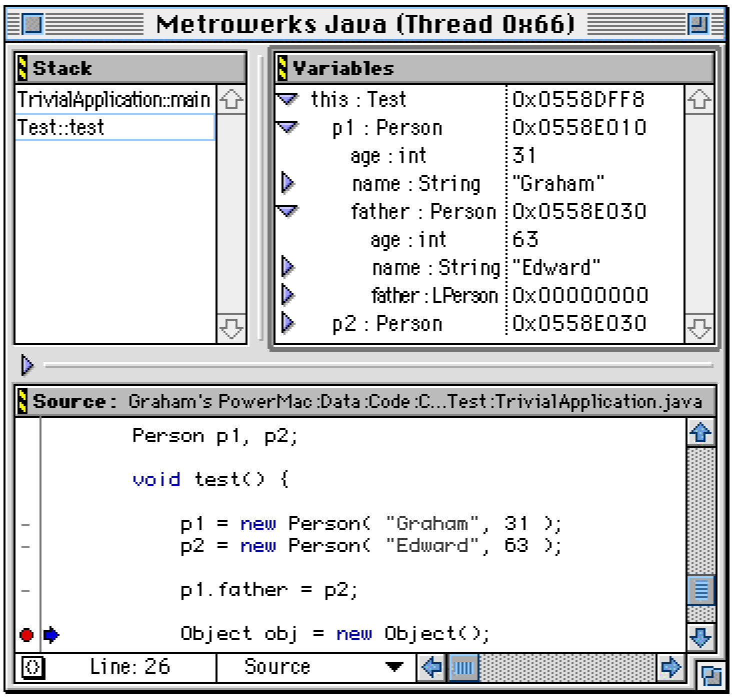
2.1 Commercial Programming Environments

Several commercial products offer integrated programming environments which support the visualisation of the state of an executing Java Virtual Machine. Although these run on non-persistent Java systems, there is considerable overlap with the facilities needed for persistent store visualisation. Such products include Metrowerks CodeWarrior [Met97] and Symantec Visual Café [Sym96] which also support other languages such as C, C++ and Pascal.

CodeWarrior provides a *Hierarchy* window which displays the class tree of the Java program being edited; clicking on a node in the tree brings up the source code for that class. During execution a symbolic debugger may be used to examine local variables accessible by a thread at a particular break-point, and the states of objects reachable via those variables. Similar facilities are available in Visual Café.

The class hierarchy display and symbolic debugger in CodeWarrior are accessed interactively by the programmer via the programming environment user interface. These facilities are only available if the program has been compiled with appropriate flags set. In contrast, OCB can be accessed through an API by any normally-compiled running Java program and made to display objects and classes in scope at the current point of execution. It can also be invoked as a stand-alone Java application to display persistent objects and classes. Since it is written solely in Java, OCB is also relatively portable, whereas CodeWarrior and Visual Café are available in multiple versions tailored to particular platforms, involving greater porting effort.

CodeWarrior[2] does not make it easy to distinguish object graph structures. For example, the *Variables* pane in Figure 1 shows an object of class *Person*. The variable *p1* contains the object at the program break-point indicated by the arrow in the *Source* pane. At this point *p1* and *p2* denote separate instances of *Person*, and *p1.father* is set to *p2*. This structure is not clear from the display, which shows the objects referenced from the current object in the form of a nested list. In particular, *this.p1.father* denotes the same object as *this.p2*, a fact that can only be deduced from the list by noticing that they both have the same address. Presumably the need to make such deductions is the reason for including addresses in the display, which seems an odd mixing of abstraction levels for a language with automatic memory management. Also, in this user interface class names used in the *Variables* pane are not linked to their definitions; it would be useful to able to click on a class name and bring up its definition directly.


Metrowerks Java (Thread 0x66)

Stack
```
TrivialApplication::main
Test::test
```

Variables
```
this : Test              0x0558DFF8
  p1 : Person            0x0558E010
    age : int            31
    name : String        "Graham"
    father : Person      0x0558E030
      age : int          63
      name : String      "Edward"
      father : LPerson   0x00000000
  p2 : Person            0x0558E030
```

Source : Graham's PowerMac:Data:Code:C...Test:TrivialApplication.java
```
    Person p1, p2;

    void test() {

        p1 = new Person( "Graham", 31 );
        p2 = new Person( "Edward", 63 );

        p1.father = p2;

        Object obj = new Object();
```
Line: 26   Source

Figure 1. CodeWarrior Java debugging window

2 Professional Release 1 at the time of writing.

In contrast, the OCB browser does not display any address information. Indeed, since it is implemented as a standard Java program it cannot obtain addresses. OCB allows objects to be distinguished on the basis of identity, so that for the example above it would be obvious to the user that *this.p1.father* and *this.p2* denote the same object, since they would be represented by the same independent window.

The Java debugger *jdb*, part of Sun's Java release, can also be used to browse object states at a program break-point. It makes no attempt to hide address information, and is command line based. In common with the CodeWarrior debugger, the states of the objects reachable by a thread can only be displayed if the running program has been compiled with a 'debug' flag set.

The O_2 object-oriented database system provides a graphical browser which allows the state of persistent and transient O_2 objects to be displayed. Its facilities are similar to those of OCB, except that it does not allow the display of inherited attributes to be controlled in the same way—OCB's style of handling this is described in Section 3.2. Indeed the authors do not know of any other browsers for inheritance-based languages which provide similar control[3].

2.2 Research Systems

Two previous object browsers with which the authors were involved are those for the persistent languages PS-algol [DB88] and Napier88 [KD90]. Similarly to OCB these can be invoked either from a running program or as a stand-alone application. The Napier88 browser displays object graph structures in the form of icons linked by directed edges.

Various other systems have provided graphical object browsers with which OCB shares some similarities in display style, for example Smalltalk-80 [GR83], Trellis/OWL [OHK87] and Cedar [Tei84].

2.3 OCB Design Aims

The main differences between OCB and the related work described above arose from the following specific design aims identified for OCB:

- to provide portability by implementing in Java;
- to allow control from running Java programs through a class interface and callback methods which allow the programmer to specify actions to be performed in response to user interaction;
- to support the visualisation of object sharing and identity, and to allow simple navigation between related objects and classes;
- to allow the graphical display format to be customised for specific classes, including the temporary hiding of superclass fields and methods.

3 OCB User Interface

3.1 Instances and Classes

This section uses the augmented definition of *Person* shown in Figure 2.

[3] OCB's style is not directly applicable to O_2 anyway, since O_2 supports multiple inheritance in contrast to Java's single inheritance.

```
public class Person implements Cloneable {
    private int     age;
    public String   name;
    public Address  address;
    public Person   mother, father;
    public Person[] children;
    public static int numberOfPeople=0;

    Person() {…}
    Person( String n, int a ) {…}

    public Date dateOfBirth() {…}
}

public class Address {
    public String street, town;
    public int  number;

    Address( int n, String s, String t ) {…}
}
```

Figure 2. Definition of class *Person* used in example

Figure 4 shows how OCB displays an instance of the class *Person*. The *Instance* pane on the left displays details of the instance's fields, while the *Class* pane on the right displays details of the class.

In the *Instance* pane the values of any fields with primitive types are displayed next to the field names, while the values of object type fields are represented by boxes containing the appropriate class names. The user may also customise the OCB display by defining alternative textual representations to be used for instances of particular classes, as described in Section 4.3. In the example a customised representation is used for strings, as illustrated for the *name* field.

The *Class* pane displays details of methods, constructors and fields, and the values of static fields. It also displays the superclass and any interfaces implemented by the class. Class and interface names are identified by underlining.

The various modifiers applicable to class members are indicated by coloured squares displayed next to member names. In the default colour coding public members are indicated by transparent squares and private members by red squares, hence the only visible squares in the example are for the *age* field. Colour coding is not used for the *static* modifier, since static members are already displayed in separate regions from non-static members. To assist the user in remembering the colouring scheme, a panel showing the current scheme is displayed whenever the user clicks a mouse button over one of the coloured squares, as illustrated in Figure 3.

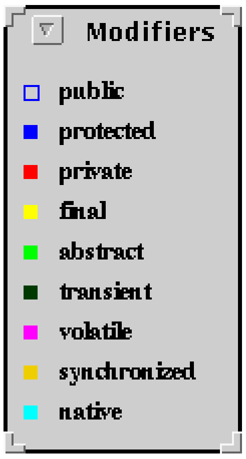
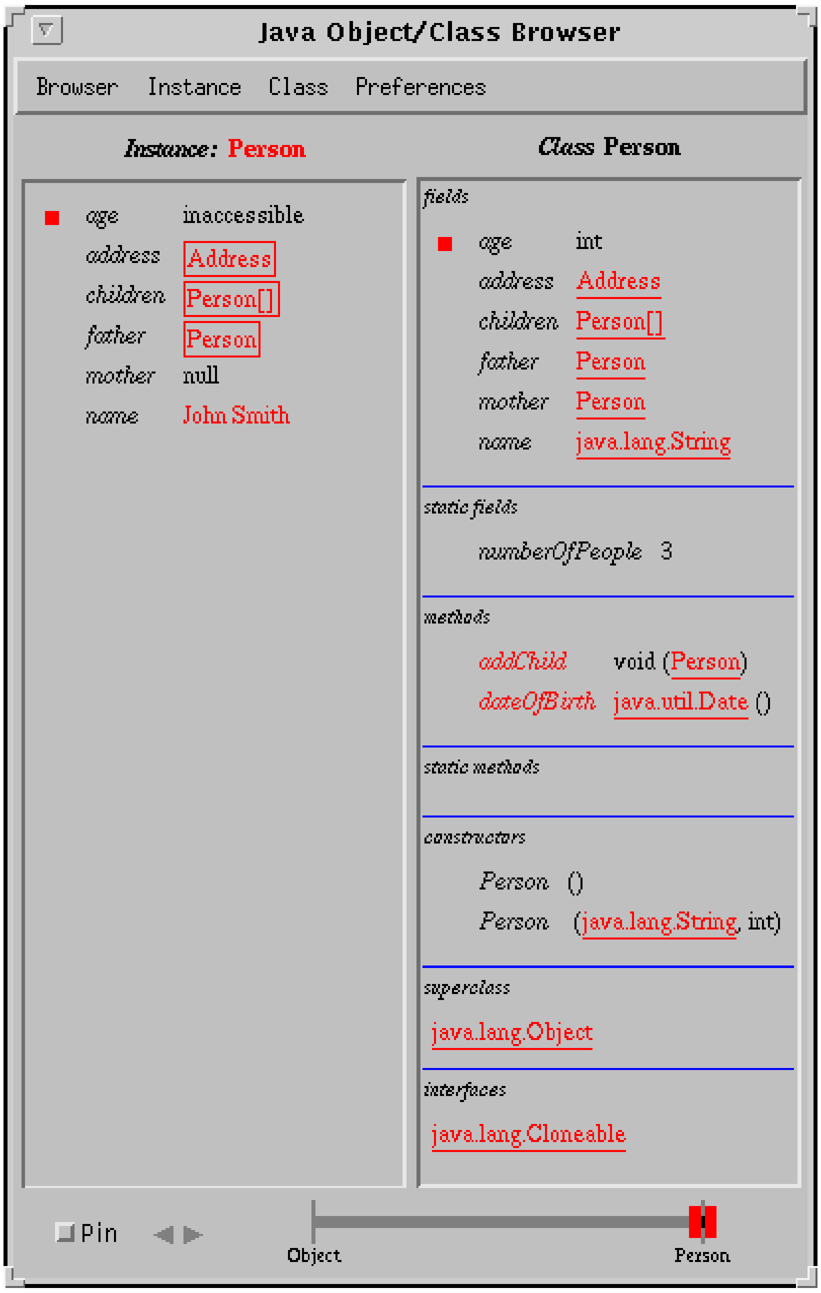


Figure 3. Modifier colour code prompt panel



Instance: Person

🟥 *age*	inaccessible
address	Address
children	Person[]
father	Person
mother	null
name	John Smith

Class Person

fields

🟥 *age*	int
address	Address
children	Person[]
father	Person
mother	Person
name	java.lang.String

static fields

numberOfPeople 3

methods

addChild	void (Person)
dateOfBirth	java.util.Date ()

static methods

constructors

Person	()
Person	(java.lang.String, int)

superclass

java.lang.Object

interfaces

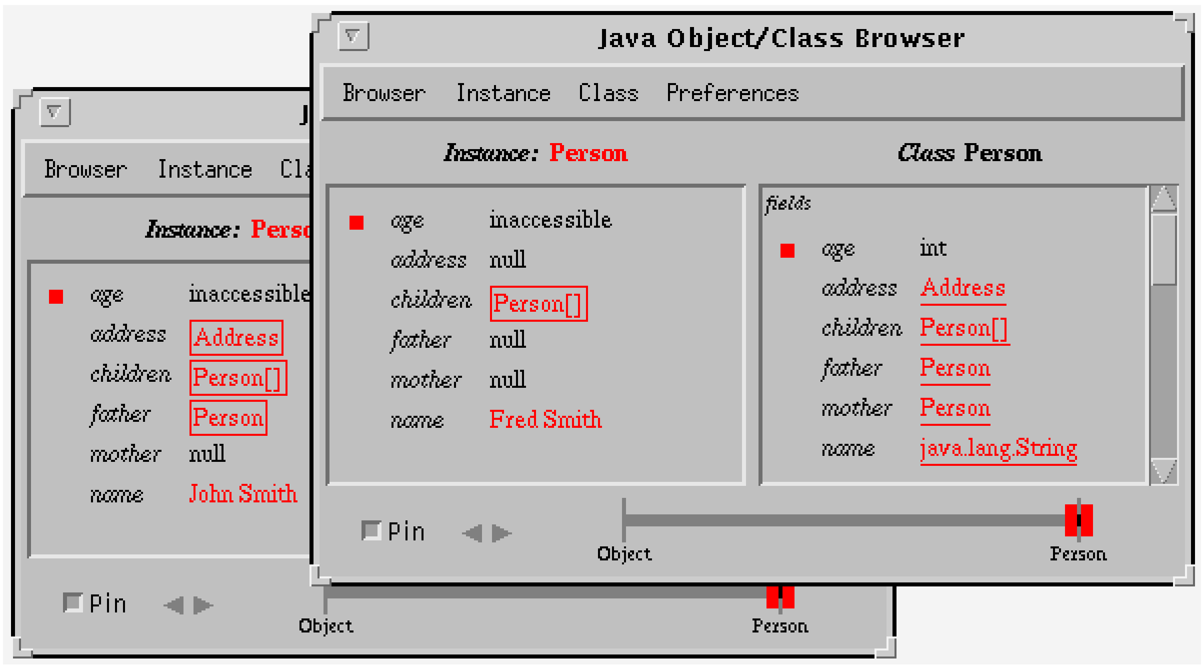
java.lang.Cloneable

Figure 4. Display of an object and its class

The user may click on various parts of the display to navigate an inter-object or inter-class link. Display regions which are sensitive in this way are differentiated by being displayed in red, as well as by the cursor which changes to a hand icon when it passes over them. Sensitive regions are used to denote values of object type fields, both in standard boxed format and customised string format, and class names which are underlined.

When the user clicks on an object value, the corresponding object and its class are displayed in turn. When the user clicks on a class name, the corresponding class is displayed. In the latter case, the OCB continues to display the current instance if the new class is the same as, a superclass of, or an interface implemented by the current class. Otherwise the new class is displayed on its own in the *Class* pane with no corresponding instance in the *Instance* pane.

The user may control whether or not a new window is created when new information is displayed. The default behaviour is that no new window is created, with the new information replacing the existing information within the current window. Alternatively, the user may pin a window by checking the *Pin* checkbox in its lower-left corner. While this is checked, new windows will be created whenever new information is displayed. New windows are initially unpinned. Each time a link is followed in an unpinned window the currently displayed object and class are pushed onto an internal stack. The user may navigate backwards and forwards in the stack using the arrow buttons next to the *Pin* checkbox. This style of browsing combines that used by web browsers such as Netscape Navigator [Net97] with the pinning mechanism of Sun Microsystems' Open Look graphical user interface [Sun89]. Figure 5 shows the display obtained by pinning the first *Person* instance and then clicking on its *father* field.



Instance: Persc

age	inaccessible
address	Address
children	Person[]
father	Person
mother	null
name	John Smith

Instance: Person

age	inaccessible
address	null
children	Person[]
father	null
mother	null
name	Fred Smith

Class Person

fields

age	int
address	Address
children	Person[]
father	Person
mother	Person
name	java.lang.String

Figure 5. Example of pinning

There is a one-one correspondence between pinned windows and the identities of the objects they are displaying. This means that the user can detect object sharing simply. For example, in Figure 5, if the *father* field of another object referring to *Fred Smith* is selected, the window displaying the *Fred Smith* object is brought to the front, rather than a new window being created.

Each window also contains menus which allow the user to select particular objects or classes to be browsed, adjust the colour coding used for modifiers, and to load and save preferences.

3.2 Viewing Inherited Members

In the examples shown earlier, only the fields and methods defined in the instance's most specific class were displayed, omitting members inherited from superclasses. For example in Figure 4 the methods inherited from class *Object* are not shown (and *Object* defines no instance fields). This is satisfactory if the methods inherited from *Object* are not of interest to the user, since the display remains relatively compact and focused on the application-specific class *Person*. The default behaviour of OCB is thus to display only the members of the most specific class, hiding any members defined in superclasses. These hidden inherited members may then be revealed under user control.

Alternatively a user may wish to view an instance through a superclass view, for example viewing a *Student* as a *Person*, requiring members defined in the most specific class to be hidden. Both requirements are met by allowing the user to set a *top* class and a *bottom* class to be applied to an OCB view, subject to the constraint *top* *bottom* *C*, where *C* is the most specific class of the displayed instance, and means "is an ancestor class of, or is equal to".

The effect is, informally, to hide any members which are defined outside the part of the class hierarchy bounded by *top* and *bottom*. More precisely, a member of *C* is displayed if and only if it is defined or over-ridden in a class *X* such that *top*   *X*   *bottom*. Furthermore, if a member which is displayed is over-ridden in a subclass of *bottom*, it is displayed in the form applicable to *bottom* rather than to *C*. To illustrate this, consider the (somewhat contrived) class definitions in Figure 6:

```
public class Animal {
    public String name;
    public int age;
}

public class Person extends Animal {
    public Person spouse;
}

public class Student extends Person {
    public Student spouse;
    public int number;
}
```

Figure 6. Example class hierarchy

This gives the class hierarchy *Object*   *Animal*   *Person*   *Student*. Table 1 defines the effects of setting *top* and *bottom* to various classes for a displayed instance of class *Student*.

top	***bottom***	***result***
Student	*Student*	The fields *spouse* and *number* are displayed, but no methods. This is the default setting (Figure 7).
Object	*Student*	The fields *name*, *age*, *spouse*, *number* are displayed, as are all of the methods inherited from *Object* (Figure 8).
Animal	*Student*	The fields *name*, *age*, *spouse*, *number* are displayed, but no methods (Figure 9).
Animal	*Person*	The instance is viewed as a *Person* and methods inherited from *Object* are hidden; the fields *name*, *age* and *spouse* are displayed, while *number* is hidden. The value of the *spouse* field is displayed as a *Person* rather than as a *Student* (Figure 10).

Table 1. Effects of various settings for *top* and *bottom*

The values of *top* and *bottom* can be set by manipulating the double-thumbed slider at the bottom of the display. The allowed positions on the slider range from class *Object* at the left, to the most specific class of the displayed instance at the right. The left thumb controls the setting of *top* and the right thumb the setting of *bottom*. Initially both are set to the most specific class. The example in Figure 7 shows both *top* and *bottom* set to *Student*: only members defined in that class are displayed.

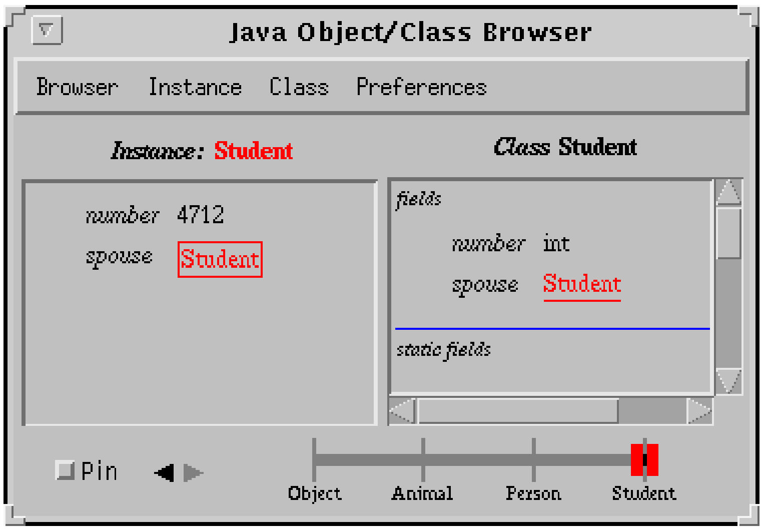


Figure 7. Viewing members of class *Student* only

The example in Figure 8 shows *top* set to *Object* and *bottom* remaining set to *Student*. This makes visible all of the members available in class *Student*, i.e. those defined in any of its ancestor classes.

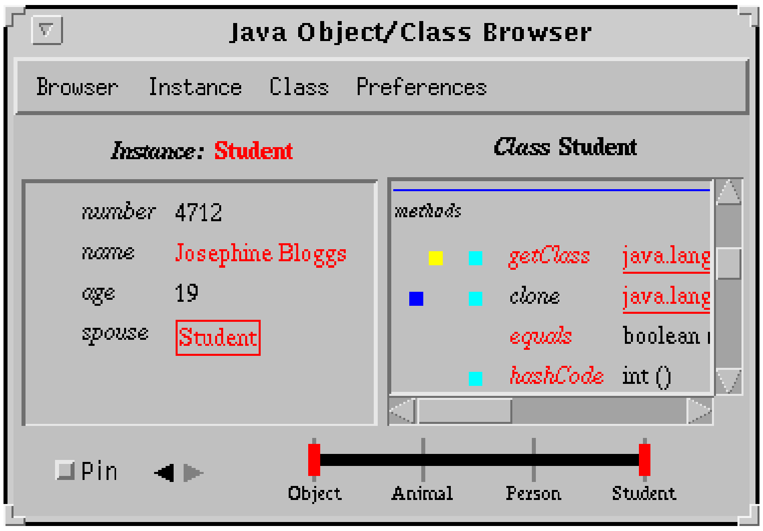
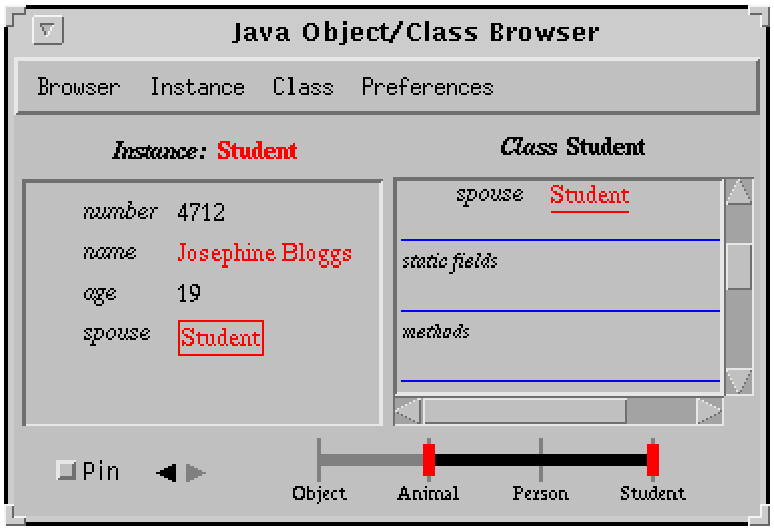


Figure 8. Viewing all members

The example in Figure 9 shows *top* set to *Animal* and *bottom* remaining set to *Student*. This filters out the members inherited from class *Object*.



Instance: Student

number 4712
name Josephine Bloggs
age 19
spouse Student

Class Student

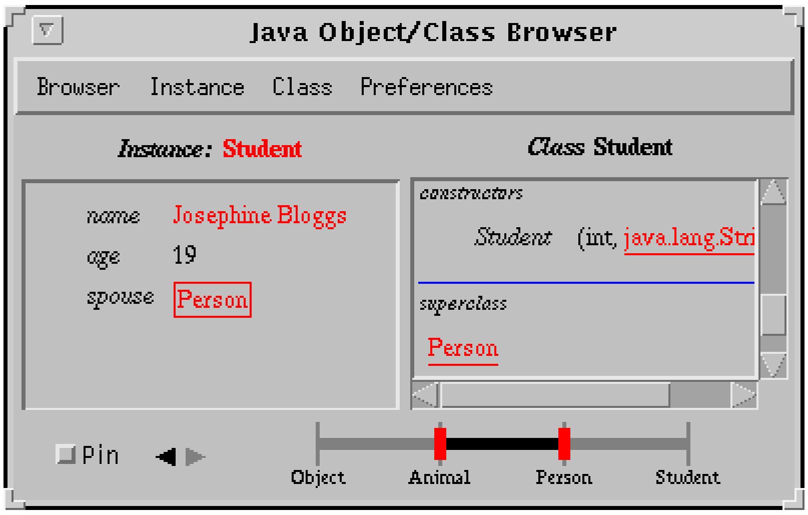
spouse Student

static fields

methods

Figure 9. Viewing only members defined in and below class *Animal*

The example in Figure 10 shows *top* set to *Animal* and *bottom* set to *Person*. This filters out members inherited from *Object*, and also those defined in *Student*, effectively viewing the *Student* instance as a *Person*. Note that the value of the *spouse* field is now displayed as a *Person*.



Instance: Student

name Josephine Bloggs
age 19
spouse Person

Class Student

constructors

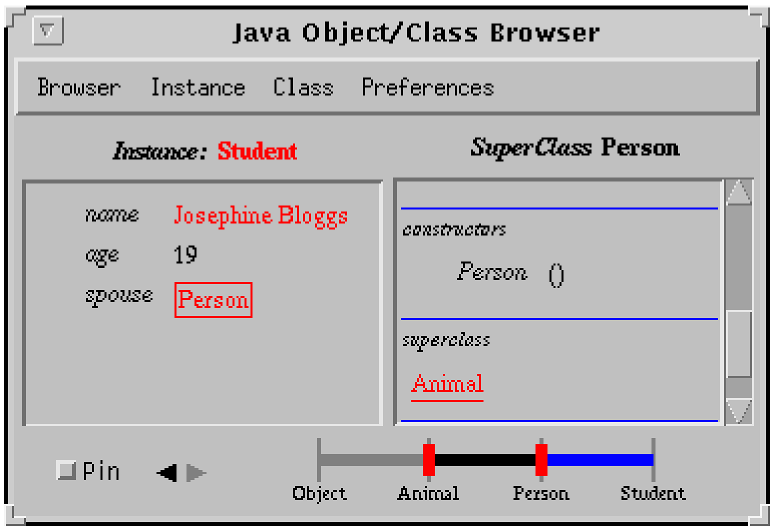
Student (int, java.lang.Stri

superclass

Person

Figure 10. Viewing instance of *Student* as a *Person*

The example in Figure 11 shows the view obtained when the superclass link to *Person* visible in Figure 10 is selected. The instance display remains the same, while the superclass *Person* is displayed subject to the same filtering out of *Object* members. Since it would now result in an inconsistency if the user were allowed to set *bottom* to *Student*—because class *Person* is being displayed—the right slider thumb cannot now be moved past *Person*[4].



Instance: Student

name Josephine Bloggs
age 19
spouse Person

SuperClass Person

constructors
Person ()

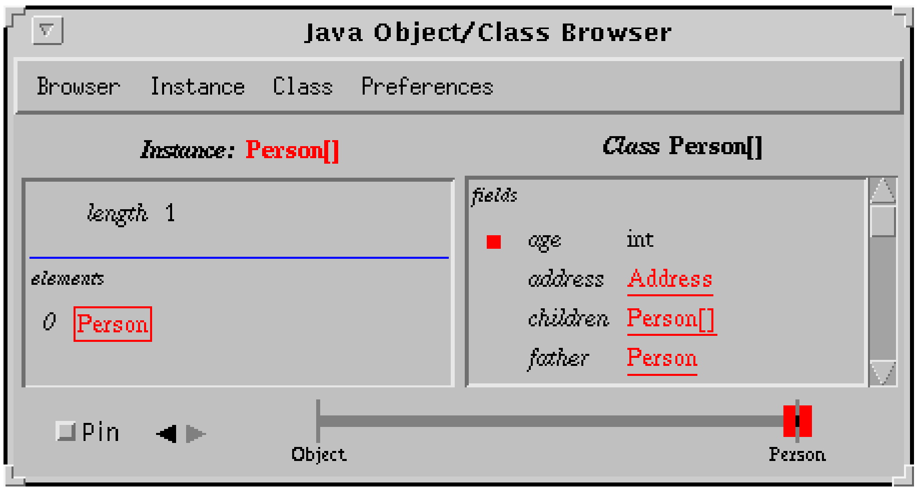
superclass
Animal

Figure 11. Viewing a superclass of *Student*

In the current implementation, the initial settings for *top* and *bottom* when an object is first displayed are the object's most specific class. As discussed in Section 6, it is planned in a future version of OCB to allow the default settings to be specified on a per-class basis.

3.3 Arrays

For array objects the *Instance* pane displays the *length* field and a scrolling list of the array components. The *Class* pane displays the class of the array elements. The example in Figure 12 shows the OCB display after the user has clicked on the box containing *Person[]* in the *children* field of Figure 4.



Instance: Person[]

length 1

elements
0 Person

Class Person[]

fields
🟥 age int
address Address
children Person[]
father Person

Figure 12. Display of an array

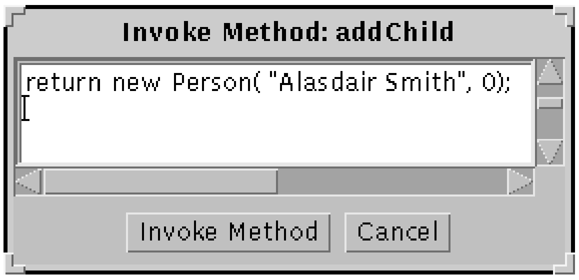
[4] Although not visible in the monochrome screen-dump, the section of the slider to the right of *Person* is now drawn in a different colour to indicate this.

3.4 Invoking Methods

As well as passively displaying the current state of objects and their classes, OCB can also be used to interact with objects by invoking their methods. A method is invoked by clicking on the appropriate name in the *methods* region of the class pane. Depending on the method signature, parameter values for the method call may be required. OCB then displays a dialogue requesting the user to enter code defining the parameter values. For each parameter the user provides a fragment of code which, when executed, will produce a suitable parameter value. For example when the *addChild* method shown in Figure 4 is clicked on, a dialogue requesting a single parameter is displayed. The text area initially contains the code

```
return new Object();
```

which is then edited by the user, as illustrated in Figure 13.



```
return new Person( "Alasdair Smith", 0);
```

Figure 13. Parameter value dialogue for method invocation

When the user presses the *Invoke Method* button, OCB compiles and executes the code to create the required parameter value, which is then used to invoke the *addChild* method. If the code entered is invalid, or its execution does not produce an instance of *Person*, an error message is displayed. For methods with multiple parameters the dialogue contains a separate code area for each one.

In the current implementation, the parameter creation code entered by the user can only refer to classes which already exist and are accessible through the normal classpath. As described in Section 6, it is planned in a future version of OCB to allow the user to define new classes at the point of method invocation which can be used in creating the parameter values.

3.5 Browser Menus

Several OCB facilities are accessed via the menus. The *Browser* menu simply allows the current window to be closed. The *Instance* menu contains an entry which gives access to persistent objects; the details of the operation depending on which persistent Java system is being used. For PJama it displays a dialogue listing the names of the current persistent roots. When one is selected the corresponding object is displayed.

The *Class* menu provides access to persistent classes in a similar way; it also contains an entry which allows the user to enter any fully qualified class name, in response to which the corresponding class is displayed. Finally the *Preferences* menu contains entries which allow the user to alter the colour coding used to display modifiers, and to load and save colour and class customisation settings from a file or from the persistent store.

4 Program Interface

4.1 Invoking OCB

A Java program may create an OCB window by instantiating the class *ocb.OCB*, which implements the interface *ocb.OCBInterface* defined in Figure 14.

```
package ocb;
import java.awt.Container;

public interface OCBInterface {
    public void displayObject( Object anObject );
    public void displayClass( Class aClass );
    public Object getDisplayedObject();
    public Class getDisplayedClass();
    public void clear();
    public void clear( String paneName );
    public void close();
    public Container getOCBContainer();
    public void addCallback( Callback cb );
    public void removeCallback( Callback cb );
    public Callback[] getCallbacks();
    public void addCustomDisplay( Class theClass, DisplayAsString customDisplay );
    public void removeCustomDisplay( Class theClass );
}

public interface DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException;
}

public interface Callback {
    public boolean callback( Object theObject, OCBInterface ocb );
    public boolean callback( Class theClass, OCBInterface ocb );
    public boolean callback( Object theObject, String name, OCBInterface ocb );
    public boolean callback( Class theClass, String name, OCBInterface ocb );
}
```

Figure 14. Interface *ocb.OCBInterface*

The constructors provided by class *ocb.OCB* are shown in Figure 15:

```
package ocb;
import java.awt.*;

public class OCB implements OCBInterface {

    /** Creates an OCB in a new frame of default size. */
    public OCB() {…}

    /** Creates an OCB in a new frame with the given position and size. */
    public OCB( Point position, Dimension size ) {…}

    /** Creates an OCB in the given container. */
    public OCB( Container parent ) {…}

    …
}
```

Figure 15. Constructors of class *ocb.OCB*

The first constructor creates an OCB which displays its information in a new *Frame* (an independent window), as does the second. The third constructor creates an OCB which displays information within a given existing *Container*. Once an instance of class *ocb.OCB* has been created, objects and classes may be displayed by invoking its *displayObject* and *displayClass* methods.

4.2 Callbacks

The user may register callback methods with an OCB instance to be called whenever an object, class or member is selected. When used in a persistent Java system these callbacks persist between sessions. The example in Figure 16 shows the registration of a callback which writes out a message whenever a class member is selected in the OCB:

```
import ocb.*;

public class MyCallback implements Callback {
    public boolean callback( Object theObject, OCBInterface ocb ) { return true; }
    public boolean callback( Object theObject, String name, OCBInterface ocb ) {
        return true; }
    public boolean callback( Class theClass, OCBInterface ocb ) { return true; }
    public boolean callback( Class theClass, String name, OCBInterface ocb ) {
        System.out.println( "class member " + name + " selected" );
        return false;
    }
}

public class Test {
    public static void main( String args[] ) {

        OCB myOCB = new OCB();

        myOCB.addCallback( new MyCallback() );

        Person john = new Person( "John Napier", 447 );
        myOCB.displayObject( john );
    }
}
```

Figure 16. Registering a callback

The *ocb.Callback* interface provides methods to be called when an object, class, object member or class member is selected. Each method returns a boolean which specifies whether other registered callbacks should also be called on this occasion.

4.3 Customising the Display

The user may associate a customised display method with a particular class. This is then used by OCB whenever it displays a field value of that class. The method takes as its parameter the class instance and returns a string representing that instance, which is then displayed in place of the default boxed class name.

Customised display methods are registered as instances of a class that implements the interface *DisplayAsString*:

```
public interface DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException;
}
```

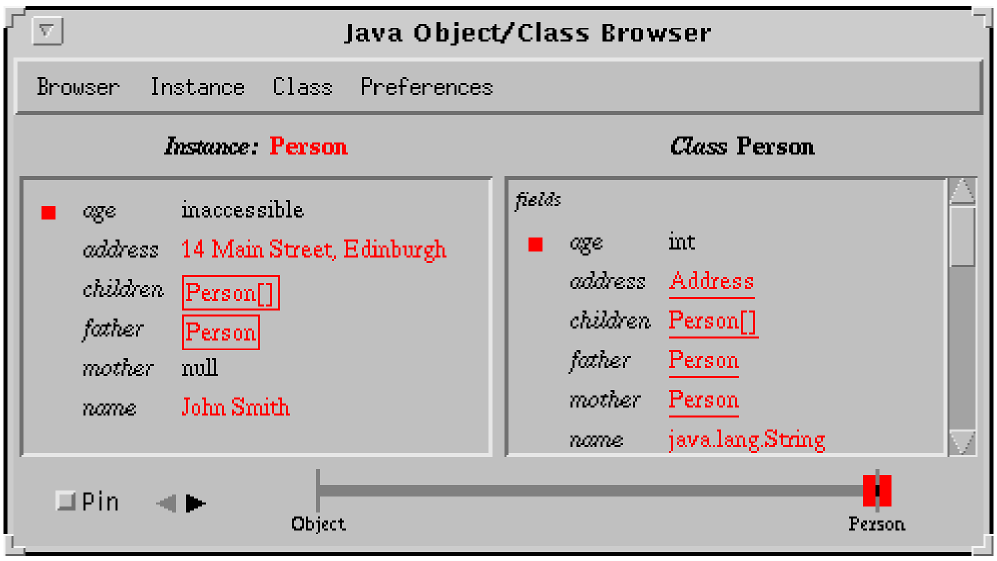
Figure 17 shows how a custom display method for class *Address* could be registered with an *OCB* instance.

```
import ocb.*;

public class AddressDisplayer implements DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException {
        if ( theObject instanceof Address ) {
            Address objectAsAddress = (Address) theObject;
            return String.valueOf( objectAsAddress.number ) + " " +
                objectAsAddress.street + ", " + objectAsAddress.town;
        } else throw new WrongClassException( "class Address expected" );
    }
}
```

```
public class Test2 {
    public static void main( String args[] ) {

        OCB myOCB = new OCB();

        try {
           myOCB.addCustomDisplay( Class.forName( "Address" ), new AddressDisplayer() );
        } catch (ClassNotFoundException e) {
           System.out.println( "Couldn't get class for Address" );
        }

        Person john = …   // Create Person instance as in Figure 2.
        myOCB.displayObject( john );
    }
}
```

Figure 17. Customisation of OCB

Figure 18 shows the display of the *Person* object with the customised display of the *Address* field.



Instance: Person

🟥	*age*	inaccessible
	address	14 Main Street, Edinburgh
	children	Person[]
	father	Person
	mother	null
	name	John Smith

Class Person

fields

🟥	*age*	int
	address	Address
	children	Person[]
	father	Person
	mother	Person
	name	java.lang.String

Figure 18. Example customised OCB display

The motivation for this somewhat cumbersome mechanism is to allow OCB customisation without the need to alter existing classes. A second simpler mechanism may be used if display by OCB can be taken into consideration at the time of class definition: where a class overrides the method *toString*, which is inherited from class *Object*, that method will be used to produce the custom string. Figure 19 shows the definition of such a subclass of *Address*. Instances of this class will be displayed in the customised form without the need to register with the *OCB* instance.

```
public class CustomAddress extends Address {
    CustomAddress( int a, String s, String t ) {
        super( a,s,t );
    }
    public String toString() {
        return String.valueOf( number ) + " " + street + ", " + town;
    }
}
```

Figure 19. Class with customised display method

5 Implementation

OCB is implemented completely in Java and requires release JDK 1.1 or later. It uses the core reflection classes in *java.lang.reflect* to discover details of the classes and objects passed to it. The graphical display is constructed using the *awt* toolkit.

In common with many Java programmers, the use of *awt* was often a frustrating experience, particularly due to the inconsistencies between implementations on various platforms. The core reflection classes, in contrast, were found to be particularly easy to use. They appear to be well designed and to provide all the functionality needed to display object states. It could be argued, though, that for full reflection the classes should allow the retrieval of method source code, although this would present significant implementation difficulties. A security mechanism would also be required in order to suppress source code in situations where public access to it was not desirable.

The core reflection classes do not, however, support the introduction of new code into the running system, the need for which was described in Section 3.4. This requires dynamic access to the Java compiler, that is, an executing Java program needs to be able to invoke the compiler, passing it a source program representation and receiving compiled classes in return. Some techniques for achieving this are described in detail in [KMC+97]. Briefly, given the core reflection classes currently provided this involves either forking an operating system process to perform the compilation, or relying on the availability of a Java compiler implemented in Java which can be invoked directly. The details of the former option depend on the platform, and it may not even always be possible. The latter option depends on access to non-core classes which may not always be present.

The need for dynamic compilation means a degree of platform dependence. Currently OCB fails gracefully by interrogating its environment before attempting to perform compilation; if it detects that compilation is not possible it simply disables dynamic method invocation. A more satisfactory solution would be for compilation support to be added to the core reflection classes. For example a class *Compiler* could be provided, removing the need for ad-hoc solutions:

```
public class Compiler {
   public Class[] compile( Source source, Class[] imports ) throws …
   …
}
```

Here the method *compile* takes a source code representation, which could be a string or a more structured representation, and an array of classes used by the code, and returns an array of compiled classes.

At the time of writing (July 1997) OCB has been fully implemented as described in this paper, with the exception of access to persistent objects and classes through the *Instance* and *Class* menus, customising colours, and the loading and saving of customised settings. OCB is freely available at the following URL:

`http://www-ppg.dcs.st-and.ac.uk/Java/OCB/`

6 Further Work

6.1 Static Visualisation

Two relatively minor enhancements are planned for the next version of OCB:

- to allow customisation of the default settings for the *top* and *bottom* class when an instance is first displayed: allow a top/bottom pair to be specified for a particular instance or for all instances of a particular class;

- to allow arbitrary new classes to be defined at the time that an object method is invoked: these classes could then be used in the creation of the parameter values to be passed to the method. For example a subclass of *Person* called *Child* could be defined dynamically and a new instance of it passed to the *addChild* method.

The issue of assisting the user in discerning the structure of linked object graphs was raised earlier. The currently implemented tool does support this to a limited extent, in that when pinned displays are used there is a one-to-one mapping between objects and OCB windows. However this becomes unsatisfactory when more than a few objects are displayed simultaneously. One possible enhancement is to provide an additional ‘overview’ window containing iconic representations of the objects encountered, connected by edges showing the inter-object references, in the style of the PS-algol and Napier88 browsers [DB88, KD90]. Other possibilities include options to display all persistent roots, and to display the results of queries executed over the persistent store.

The class customisation mechanism described in Section 4.3 supports only customised string representations for particular classes. It would be relatively simple to extend this to allow the specification of graphical representations which would be displayed in place of the standard boxed class name representation, perhaps by renaming the interface *DisplayAsString* to *CustomiseDisplay*, with the methods:

```
public interface CustomiseDisplay {
    public String objectToString( Object theObject ) throws WrongClassException;
    public Image  objectToImage( Object theObject ) throws WrongClassException;
}
```

In this case the *objectToImage* method would be called if the *objectToString* method returned *null*. Another possibility would be for instances of the customised class appearing in fields to be displayed using nested OCB windows, so that the entire field value would be displayed within the parent window. This option could be indicated by having both methods return *null*.

As mentioned in the previous section, a possible addition to the core reflection classes would be the ability to obtain source code from a given method representation. If this were supported the OCB display could then be adapted to show the source of each method in the class pane. Clearly this would raise security issues with regard to controlling which users were allowed to access source code; it would not be acceptable to allow universal access.

One of the issues considered in the design of OCB was whether the display of an object and its members should respect the access modifiers specified in its class definition, for example whether a *private* member should be visible. More generally, should the user be able to discover or affect any aspect of the Java system that they would not be able to do by writing an appropriate program? However, it turned out to be a non-issue: because OCB is implemented solely in Java, by definition it can only perform actions permitted by the language rules. While giving a clean and relatively safe tool—a user cannot do anything using OCB that they could not do anyway—this is too restrictive for debugging purposes. For example it would be useful if all the members defined in a certain package were accessible by OCB while the package was under development. This suggests a refinement of the core reflection facilities to support over-riding of language security mechanisms in limited circumstances.

6.2 Dynamic Visualisation

The current functionality of OCB is to display a snapshot of the state of an object at one particular moment. It could also be enhanced to provide an active monitoring capability, whereby an object could be polled regularly by an OCB instance, with any changes being reflected in an updated display window. This would make OCB more useful as a debugging tool.

As mentioned earlier, the display of persistent objects and their classes is only part of what is necessary for the visualisation of a persistent Java system: the programmer may also need to examine the states of the threads executing at a particular time. Since threads are Java objects it is possible to pass a thread to the current OCB implementation for display. However this does not reveal very much interesting information since most of the thread’s state is not publicly accessible.

An obvious avenue for development of OCB is to extend it with symbolic debugging capabilities, allowing threads to be started and stopped, break-points set etc. Clearly given the current core classes this would have to be implemented using a non-standard JVM. More interestingly, perhaps the core reflection classes could be augmented to allow sufficient safe introspection into the dynamic state of the system for such facilities to be implemented in “100% Pure Java™”. There is considerable existing work on Meta-Object Protocols to support this style of reflection in other languages [KRB91, MJD96].

6.3 Implementation

The efficiency of the implementation could be improved by caching the results of various calculations internally. Each time an object is displayed the structure of its class is traversed, involving a significant number of calls to the core reflection classes. The window layout is then calculated on the basis of that structure. Some results from both of these stages are dependent only of the class of the object and could thus be cached on a per-class basis. Similarly, it is quite common for the user to press the *back* arrow to return to the previously viewed object. In the current implementation this results in a complete recalculation, whereas the complete display state could be cached, keyed by the identity of the object. Where OCB is used with a persistent Java system, both forms of cache could persist across multiple user sessions.

7 Acknowledgements

This work is partially supported by the EPSRC under grant GR/J 67611 “Delivering the Benefits of Persistence to System Construction” and by a gift from Sun Microsystems. We thank Malcolm Atkinson, Laurent Daynès, Mick Jordan, Dave Munro, Michael Van De Vanter and the anonymous reviewers for their helpful suggestions.

8 References

[ADJ+96] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. & Spence, S. “An Orthogonally Persistent Java™”. SIGMOD Record 25, 4 (1996) pp 68-75.

[DB88] Dearle, A. & Brown, A.L. “Safe Browsing in a Strongly Typed Persistent Environment”. Computer Journal 31, 6 (1988) pp 540-544. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1988.html#safe.browsing*.

[DHF96] Dearle, A., Hulse, D. & Farkas, A. “Operating System Support for Java”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996).

[GN96] Garthwaite, A. & Nettles, S. “Transactions for Java”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996).

[GR83] Goldberg, A. & Robson, D. **Smalltalk-80: The Language and its Implementation**. Addison Wesley, Reading, Massachusetts (1983).

[KD90] Kirby, G.N.C. & Dearle, A. “An Adaptive Graphical Browser for Napier88”. University of St Andrews Technical Report CS/90/16 (1990). URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1990.html#napier.browser*.

[KMC+97] Kirby, G.N.C., Morrison, R., Connor, R.C.H. & Stemple, D.W. “Linguistic Reflection as a Paradigm for Program Generation in Java”. Submitted for publication (1997).

[KRB91] Kiczales, G., des Rivières, J. & Bobrow, D. **The Art of the Metaobject Protocol**. MIT Press, Cambridge, Massachusetts (1991).

[MCK+96] Morrison, R., Connor, R.C.H., Kirby, G.N.C. & Munro, D.S. “Can Java Persist?”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996) pp 34-41. URL: *http://www-ppg.dcs.st-and.ac.uk/ Publications/1996.html#java.persistence*.

[Met97] Metrowerks Inc “Metrowerks CodeWarrior”. (1997). URL*: http://www.metrowerks.com/*.

[MJD96] Malenfant, J., Jacques, M. & Demers, F. “A Tutorial on Behavioral Reflection and its Implementation”. In Proc. Reflection 96, San Francisco (1996) pp 1-20.

[Net97] Netscape Communications Corporation “Netscape Navigator”. (1997). URL*: http://www.netscape.com/*.

[OHK87] O’Brien, P.D., Halbert, D.C. & Kilian, M.F. “The Trellis Programming Environment”. In Proc. International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’87), Orlando, Florida (1987) pp 91-102.

[Sun89] Sun Microsystems **Open Look™ Graphical User Interface Functional Specification**. Addison-Wesley, Mountain View, California (1989).

[Sym96] Symantec “Symantec Visual Café”. (1996). URL*: http://www.symantec.com/*.

[Tei84] Teitelman, W. “A Tour Through Cedar”. IEEE Software, April (1984) pp 44-73.