This paper should be referenced as:

Zirintsis, E., Dunstan, V.S., Kirby, G.N.C. & Morrison, R. “Hyper-Programming in Java”. In Proc. 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California (1998).

Hyper-Programming in Java

E. Zirintsis, V.S. Dunstan, G.N.C. Kirby & R. Morrison

Department of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland

Email: {vangelis, graham, ron}@dcs.st-and.ac.uk

Abstract

Hyper-programming is a technology only available in persistent systems, since hyper-program source code contains both text and links to persistent objects. A hyper-programming system has already been prototyped in the persistent programming language Napier88. Here we report on the transfer of that technology to an object-oriented platform, Java. The component technologies required for hyper-programming include linguistic reflection, a persistent store, and a browsing mechanism, all of which have been reported elsewhere. The topics of discussion here are the additional technologies of: the specification of denotable hyper-links in Java; a mechanism for preserving hyper-links over compilation; a hyper-program editor; and the integration of the editor and the browser with the hyper-programming user interface. We describe their design and implementation. In total, these technologies constitute a hyper-programming system in Java.

1 Introduction

In persistent systems, programs may be composed and stored in the same environment as that in which they are executed. At the time of program composition, objects accessed by a program may already be available in the persistent store. Since the program itself is a persistent object, it can include direct links to such objects rather than textual descriptions of how to locate the objects. A program containing both text and links to objects is called a hyper-program [1]. Figure 1 shows an example hyper-program that contains a link to another persistent object. The analogy with hyper-text is made by the links being displayed as buttons. The programmer can follow these links using a browsing tool, to reveal further hyper-programs and other data structures.

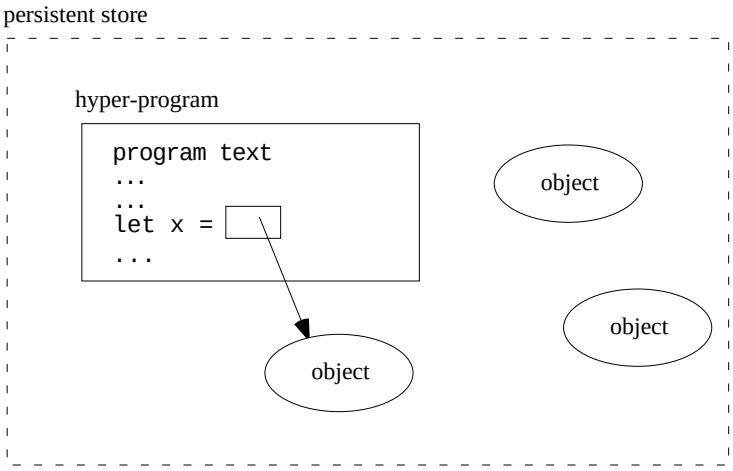

Figure 1. Example hyper-program

The benefits of hyper-programming are discussed in [1-3]. They include:

- being able to perform program checking early
- support for source representations of all object closures
- being able to enforce associations from executable programs to source programs
- availability of an increased range of linking times

- increased program succinctness
- increased ease of program composition

A hyper-programming system has already been prototyped in the persistent programming language Napier88 [4]. Here we report on the design and the transfer of that technology to an object-oriented platform, a persistent form of Java, PJama [5]. From our experience with Napier88 we identify the following as the requirements for providing a hyper-programming system in Java [6]:

- a persistent store with root(s), reachability and referential integrity
- linguistic reflection as a programming technique
- a persistent store browser
- a specification of denotable hyper-links in Java
- a mechanism for preserving hyper-links over compilation
- a hyper-program editor
- a graphical user interface

Some of this technology, including a persistent store for Java – PJama [5], linguistic reflection [7, 8] and a browsing mechanism [9], has been reported elsewhere. Here we concentrate on the additional technologies of: the specification of denotable hyper-links in Java; a mechanism for preserving hyper-links over compilation; a hyper-program editor; and the integration of the editor and the browser with the hyper-programming user interface. In total these technologies constitute a hyper-programming system in Java.

2 Denotable Hyper-Links in Java

The first step in designing a hyper-programming system is to define the denotable values that may be hyper-linked. In Napier88, a hyper-link can be made to any data value. This is captured by the syntactic production *name* in the Napier88 formal definition [4] and includes both values and locations that contain values.

In Java not all denotable values are first class values, for example methods, and there is no simple production rule to capture this in the syntactic definition. We define the denotable values that can be hyper-linked in Java as: objects; classes; interfaces; arrays; array elements; static members; non-static members; and constructors. Furthermore, links may be made to both values and locations that contain values (such as fields and array elements) where appropriate.

Table 1 illustrates the Java denotable values that can be hyper-linked and their corresponding syntactic productions [10].

Hyper-link To	Production
class	ClassType
primitive type	PrimitiveType
interface	InterfaceType
array type	ArrayType
object	Primary
primitive value	Literal
(static) field	FieldAccess
(static) method	Name
constructor	Name
array	Primary
array element	ArrayAccess

Table 1. Java hyper-links and productions

The Napier88 hyper-programming system allows a hyper-link to be inserted anywhere in a program whether it is a syntactically legal use or not. Illegal uses will result in compilation errors. The same is true in our present Java system but we intend to incorporate a parser into the editing system to direct syntactically legal insertions of hyper-links. In the context of Table 1, if a hyper-link cannot be parsed as its equivalent production then it is syntactically

illegal. If it can then its use is context sensitive with respect to the surrounding hyper-program. In that sense the equivalence between hyper-link and production is necessary but not sufficient for legal syntactic construction. For example, a hyper-link can appear legally at a position corresponding to the production *Name* where it denotes a constructor, but not where it denotes a package name, since packages cannot be linked to.

3 Representing Hyper-Programs

The Java hyper-programming system uses three different representations for hyper-programs at various stages of the program development process. The *editing* form is optimised for editing, including fast selection, insertion and deletion of both text and links. The *storage* form is optimised for storage and is used in preserving hyper-links over compilation. The *textual* form is designed for use with a standard Java compiler.

Translation between the editing form and the storage form takes place when the hyper-program editor accesses or stores a hyper-program in the persistent store. The textual form is generated by a new hyper-program compiler method *compileClass (HyperProgram)* which takes an instance of the storage form as a parameter. By performing this translation the method can call any standard Java compiler to perform the compilation.

In the rest of the paper we will illustrate the implementation of hyper-programming using the hyper-program shown in Figure 2, defining the public class *MarryExample*.

```
public class MarryExample {
        public static void main(String[] args) {
                Person.marry ( vangelis , mary );
        }
}
```

Figure 2. An example hyper-program in Java

The body of the method *main* contains a link to the static method *Person.marry*, and links to two persistent instances of class *Person*, which is partially defined in Figure 3.

```
public class Person {
    private String name;
    private Person spouse;

    public static void marry (Person a, Person b) {
        a.spouse = b;
        b.spouse = a;
    }
    … // Other methods
}
```

Figure 3. Definition of class *Person*

3.1 The Storage Form

The storage form of a hyper-program is represented by the class *HyperProgram*. Part of the definition is shown in Figure 4. It contains a string and a vector of *HyperLinkHP* instances. The string contains the textual part of the hyper-program while the vector contains references to the hyper-linked entities.

```
public class HyperProgram {
    protected String theText;
    protected Vector theLinks;
    // Other fields

    public HyperProgram() {
        theText = "";
        theLinks = new Vector(); }
```

```
    public HyperProgram (String theText) {
        this.theText = theText;
        theLinks = new Vector(); }

    public HyperProgram (String theText, Vector theLinks) {
        this.theText = theText;
        this.theLinks = theLinks; }

    public String getTheText () {
        // Returns the textual part of the hyper-program
    }

    public Vector getTheLinks () {
        // Returns the vector containing HyperLinkHP instances
    }

    // Other methods
}
```

Figure 4. The hyper-program storage form

The example hyper-program is shown in this form in Figure 5.

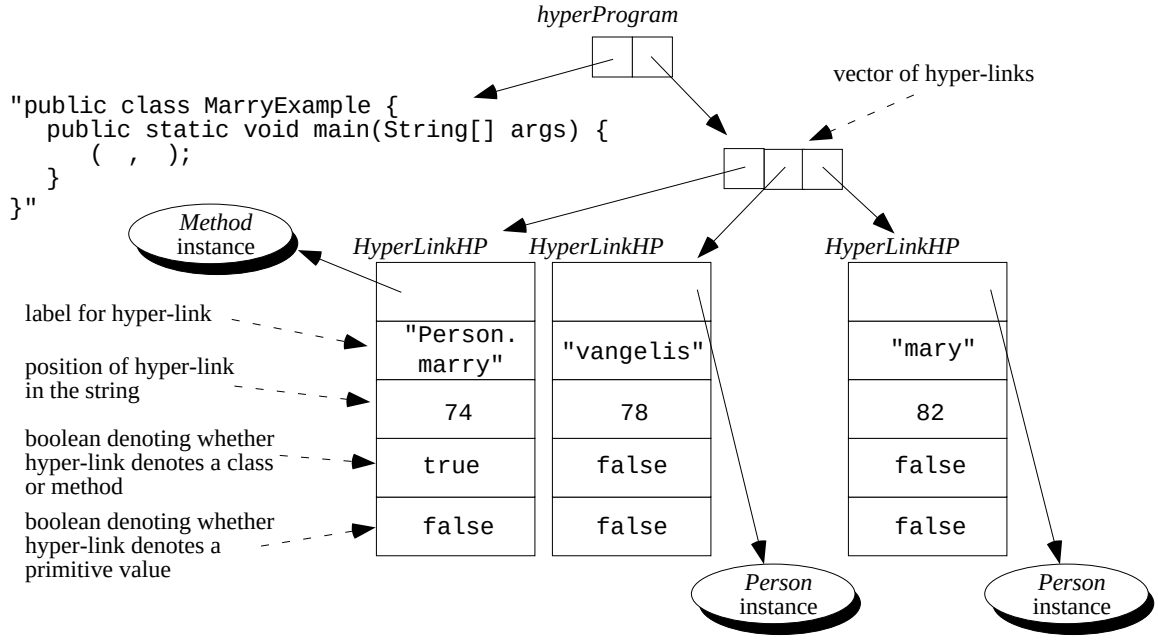

Figure 5. An instance of the hyper-program storage form

The class *HyperLinkHP*, illustrated as part of Figure 5, is used to represent hyper-links and is defined in Figure 6. The use of the field *hyperLinkObject* depends on the kind of hyper-link. In our example, for the link to the static method *Person.marry*, it refers to the instance of class *Method* representing the method, and for the links to the objects it refers to the objects themselves.

```
public class HyperLinkHP {
       protected Object  hyperLinkObject;
       protected String  label;
       protected int     stringPos;
       protected boolean isSpecial;
       protected boolean isPrimitive;
       // Other declarations and initialisations
```

```
        public HyperLinkHP (Object hyperLinkObject, String label,
                            int stringPos, boolean isSpecial, boolean isPrimitive) {
                this.hyperLinkObject = hyperLinkObject;
                this.label =           label;
                this.stringPos =       stringPos;
                this.isSpecial =       isSpecial;
                this.isPrimitive =     isPrimitive;
        }
        // Other constructors

        public Object getObject ()      { return hyperLinkObject; }
        public String getLabel()        { return label; }
        public int getStringPos()       { return stringPos; }
        public boolean getIsSpecial()   { return isSpecial; }
        public boolean getIsPrimitive() { return isPrimitive; }
        // Other methods
}
```

Figure 6. The representation of hyper-links

4 Compiling and Running Hyper-Programs

In type safe linguistic reflection [7, 8, 11], the executing application generates new program fragments in the form of source code, invokes a dynamically callable compiler, and finally links the results of the compilation into its own execution. We use this technique to process a hyper-program: a textual equivalent is generated and compiled, the resulting class is loaded, and can then be instantiated using Java core reflection.

4.1 The Textual Form

Standard Java compilers operate on textual source programs rather than hyper-programs. To enable a hyper-program to be compiled with such a compiler, it is first translated into a purely textual form in which each hyper-link is replaced by an equivalent textual denotation.

To ensure that every hyper-link has such a textual form, the system records a reference to each hyper-program submitted for translation, in a password-protected location in the persistent store. The hyper-linked entities will thus remain accessible by the compiled form even if the original hyper-program is discarded. The textual denotation of an individual hyper-link is an expression that will retrieve the hyper-linked entity from the password-protected data structure, and the password protection prevents any accidental or malicious tampering with the data structure.

The precise form of the retrieval expression generated for each hyper-link depends on the kind of the hyper-link (object, class, method etc), but always includes a unique identifier allocated to each hyper-program when it is processed, and the index of the hyper-link within the hyper-program. Examples are given in Section 4.2.

In our current implementation, no hyper-program that is translated and compiled can be subsequently garbage collected, since a reference to it is stored in the implementation data structure. To overcome this problem, weak references will be used in the next version (requiring JDK 1.2) so that hyper-programs may be garbage collected once no user references to them remain. This is illustrated in Figure 7.

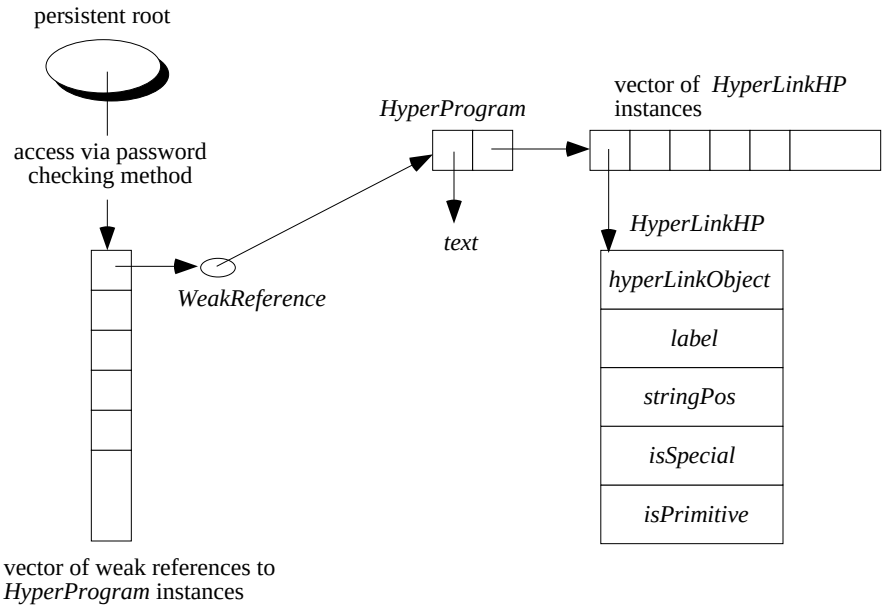

Figure 7. The hyper-program storage data structure

To recap, a reference to each hyper-program being compiled is placed in the persistent store at a known (textually denotable) location. Using this, a textual form can be generated for each hyper-link. Once compiled, the execution of the textual form allows access to the original hyper-links via the password protected data structure.

4.2 Example of Textual Form

The textual form of a hyper-program is generated by the method *generateTextualForm* of the *DynamicCompiler* class (see Figure 9), using information recorded in the storage form. A textual equivalent is generated for each hyper-link, with a structure depending on the kind of hyper-link. For example, the string generated for a hyper-link to an object has the form:

```
((class name) DynamicCompiler.getLink (secret password,
                                       unique id for hyper-program,
                                       unique id for hyper-link).getObject())
```

The static method *getLink* retrieves a specified *HyperLinkHP* instance from the persistent data structure of Figure 7, taking as parameters the password and identifiers for the hyper-program and hyper-link. The call to the *getObject* method returns the hyper-linked object itself, which is then cast to its specific class. The entire string thus gives an access path to the hyper-linked object that may be evaluated correctly at run-time.

A link to a static method does not require any persistent objects to be retrieved; the string simply has the form:

```
fully qualified method name
```

Figure 8 shows the resulting textual form for the example hyper-program.

```
1     “import compiler.DynamicCompiler;
2     import Person;
3     public class MarryExample {
4       public static void main( String[] args ) {
5         Person.marry(
6           ((Person) DynamicCompiler.getLink ("passwd", 0, 1).getObject()) ,
7           ((Person) DynamicCompiler.getLink ("passwd", 0, 2).getObject()) );
8         }
9     }”
```

Figure 8. An instance of the hyper-program textual form

The details of the hyper-link textual equivalents in lines 5-7 are generated as follows:

- The name of the static method, the string `Person.marry` in line 5, is generated by combining the names of the method and its defining class. The former is obtained by calling the method *getName* on the *Method* instance recorded in the corresponding *HyperLinkHP* instance, and the latter by calling *getDeclaringClass* on the *Method* instance, followed by *getName*. Note that a hyper-link to an object of class *Method* would also be represented by a *Method* instance in the *hyperLinkObject* field; the value of the field *isSpecial* is used to distinguish between the possible interpretations.
- The string `(Person)` defining the class casts in lines 6 and 7 is obtained by calling the *getClass* method on the objects recorded in the corresponding *HyperLinkHP* instances, followed by *getName*.
- The password used in the calls to *getLink* in lines 6 and 7 is built into the system. The hyper-program and hyper-link indices are the offsets in the respective persistent vectors.

4.3 Compiling Textual Form

After generating the textual form, the system calls a standard Java compiler dynamically, to compile the textual form into a class that is equivalent to the original hyper-program. Although no facilities for dynamic compilation are provided directly by the Java environment, it is possible to implement a class that provides such facilities. Figure 9 shows several compilation methods provided by the class *DynamicCompiler*.

```
public class DynamicCompiler {
    // Field declarations and initialisations

    public static Class[] compileClasses (String[] classNames, String[] classDefns)
                            throws Exception {
        boolean compiled = false;
        try { // Direct invocation of the standard Java compiler
        } catch (Exception e) {} // Ignore errors
        if (! compiled) {
            // Direct invocation of compiler failed. Fork an operating system process
        }
        // Create a class loader and use it to load class
        ...
    }
    public static Class compileClass (String className, String classDefn) throws Exception {
        // Compiles a single class using compileClasses above
    }
    public static Class[] compileClasses (HyperProgram[] hps) throws Exception {
        String[] classNames = new String[ hps.length ];
        String[] classDefns = new String[ hps.length ];
        for (int i = 0; i < hps.length; i++) {
            addHP (hps [i], "passwd");
            classNames [i] = hps [i].getClassName ();
            classDefns [i] = generateTextualForm (hps[i]);
        }
        Class[] result = compileClasses (classNames, classDefns);
        return result;
    }
    public static Class[] compileClass (HyperProgram hp) throws Exception {
        // Compiles a single hyper-program using compileClasses (HyperProgram[])
    }
    public static String generateTextualForm (HyperProgram hp) {
        // Generates the textual form required by the compiler
    }
    public static HyperLinkHP getLink (String password, int hpIndex, int hlIndex) {
        // Returns representation of a given hyper-link.
    }
    private static void addHP (HyperProgram hp, String password) {
        // Adds hp to the persistent vector of hyper-programs (if not already present)
    }
    // Other methods
}
```

Figure 9. The DynamicCompiler class

The *DynamicCompiler* class includes methods for generating the textual form. The *generateTextualForm* method takes a *HyperProgram* instance as parameter and returns a string. The methods *getLink* and *addHP* retrieve a given *HyperLinkHP* instance, and add a *HyperProgram* instance to the persistent vector, respectively. Both require a password to prevent unauthorised access to the vector containing the hyper-programs.

The *compileClasses(String[], String[])* compiler method takes an array of source code strings defining a number of classes and attempts to compile them by invoking the standard Java compiler directly as a Java class. If this fails, then a new operating system process is forked to call the Java compiler. If the compilation is successful, the result is an array of instances of class *Class*, otherwise an exception is thrown.

The first mechanism has the advantage of fewer run-time overheads. The disadvantage is the reliance on knowledge of the Java implementation, in particular of the compiler interface and of which package contains the compiler. Thus a change in the Java implementation—such as placing the compiler in a different package or re-implementation of the compiler in a different language—would prevent this approach from working. The disadvantages of the second mechanism are that significant additional run-time resources are involved in creating a new instantiation of the JVM, and that it is more platform-specific.

Other compiler methods provided take *HyperProgram* instances as parameters and compile them to instances of the class *Class*. This involves conversion from the storage form to the textual form, which is then passed to the string compilation method as described before.

Dynamic compilation, if successful, creates class definitions in the form of byte code sequences (.class files). Using the example, compilation will result in the creation of the file *MarryExample.class*. To be useful this must then be loaded into the running system and converted to a *Class* object. This is achieved by using a subclass of the class *Classloader*—details are given in [7]. Once a class has been loaded at run-time, instances of the class can be created using the *newInstance* method.

5 User Interface

The user interface to the hyper-programming system has two components: the hyper-program editor, which is used to construct and edit hyper-programs, and the object/class browser, which is used to select the persistent data to be linked into the hyper-programs.

5.1 The Hyper-Program Editor

The hyper-program editor is designed to support the following requirements:

- basic editing facilities;
- support for embedded hyper-links;
- drag and drop of hyper-links and text; and
- multiple fonts, sizes, styles and colours (faces).

The editor is implemented in three layers, as shown in Figure 10.

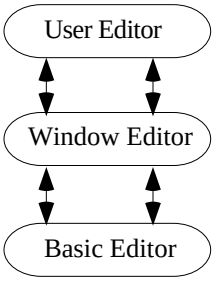

Figure 10. The hyper-programming editor layers

This allows implementations of the different layers to be changed independently. The envisaged use of each layer is as follows:

- The basic editor stores and manipulates text and hyper-links. It supports basic operations such as insertion, cutting and pasting of text and links.
- The window editor provides an API for the graphical display and editing of the contents of a basic editor. It supports multiple fonts, sizes and colours.
- Various higher-level user editors may be constructed using the window editor API. One, the hyper-program editor, is pre-defined.

5.2 The Editing Form

The hyper-program editing form is the data structure used in the basic editor. It is similar to the storage form but is optimised for editing operations. An example is shown in Figure 11.

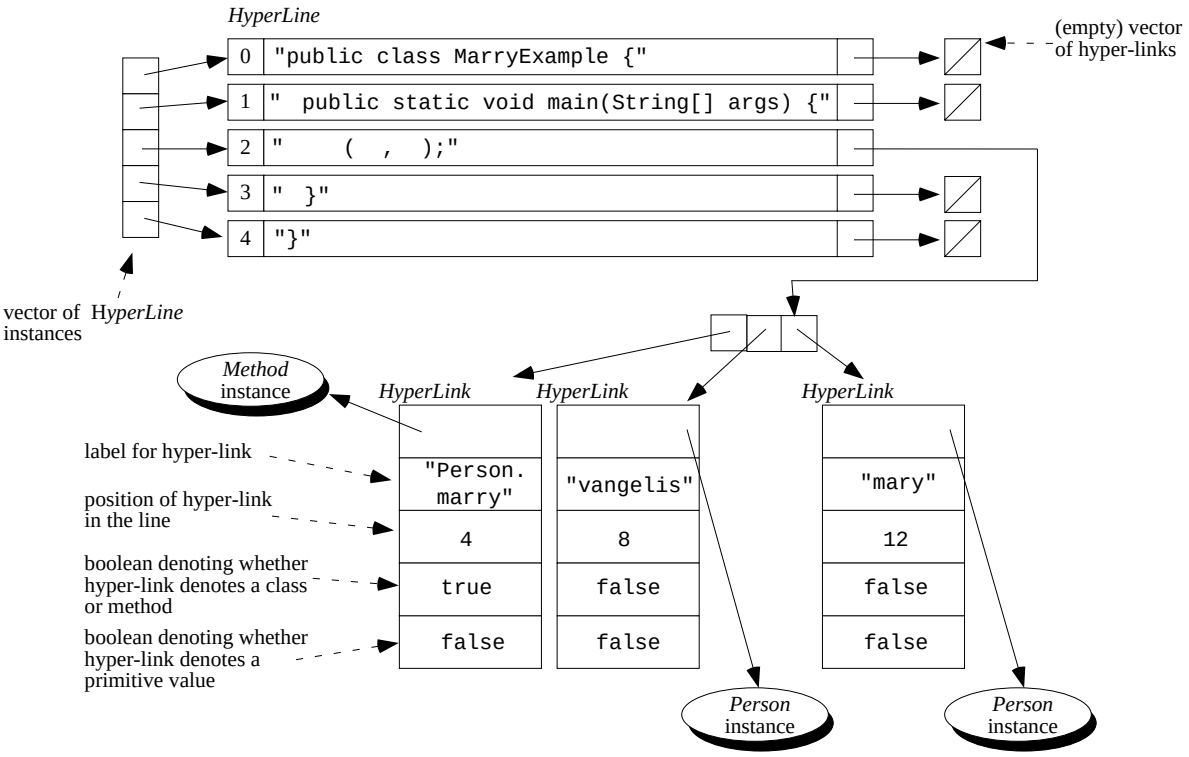

Figure 11. An instance of the hyper-program editing form

The editing form is designed to be efficient in: memory and disk storage requirements; I/O transfers; performing operations such as basic editing and navigation; and hyper-link manipulation. The textual part of each line is kept in a separate string. The position of each hyper-link is defined by a pair of values (line number, offset).

5.3 The Object/Class Browser

The OCB browser [9] was initially designed in response to a need identified by the developers of the PJama persistent Java implementation [5]. It was recognised, however, that most of its facilities would also be useful in hyper-programming. 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 OCB browser has the following design aims:

- to provide portability by implementing in Java;
- to allow control from running Java programs through a class interface and call-back 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.
- to support hyper-programming in Java.

5.4 User Interface Example

5.4.1 Composing Hyper-Programs

Figure 12 illustrates the hyper-programming user interface. It shows a hyper-program editor window, containing text and three links, and an Object/Class Browser (OCB) browser window. As values are discovered in the persistent store using OCB, they may be linked into the editor source. The first link, to the static method *Person.marry*, is displayed with a textual label. The other links, to instances of Person, are displayed with image labels. The OCB window shows a representation of an instance of *Person*.

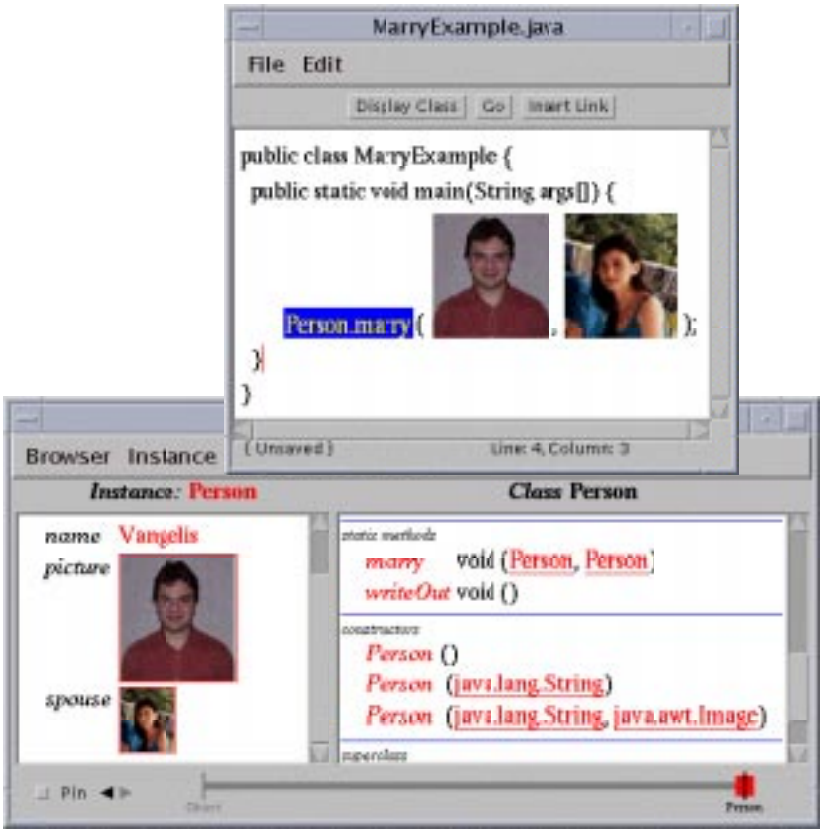

Figure 12. The hyper-programming user interface

The programmer composes a hyper-program by a combination of typing text into the editor window and inserting links to existing data. In Figure 12, the browser window displays an instance of the class *Person* in the left panel and the static method *marry* in the right panel. A hyper-link may be inserted by positioning the mouse over a denotable entity in the browser window and pressing the right-hand mouse button, or by pressing the *Insert Link* button in the editor window. In the first case a hyper-link to the selected entity is inserted into the front-most editor window, while in the latter case a hyper-link to the object displayed in the front-most browser window is inserted into the selected editor window. Figure 12 shows the display after a link to the method *marry* has been inserted. Where appropriate, the user can select whether to link to a value or the location containing the value, by pressing the right-hand mouse button over the right or left half of the panel respectively. In a future version, we plan to support insertion of hyper-links by drag and drop.

Each inserted hyper-link is displayed as a button in the editor window. These buttons can be moved and edited in the same way as the text. The names or images displayed on the buttons can be changed and are not significant to the

semantics of the hyper-program. If the programmer presses a button, the associated entity is displayed in the top-most browser window.

5.4.2 Compiling, Loading and Executing Hyper-Programs

Once a hyper-program is completed it may be translated, compiled and the resulting classes loaded. The programmer may then either display the principal class in the browser, or execute the main method of the principal class directly[1]. These options are selected using the *Display Class* and *Go* buttons respectively.

If compilation fails, an error message is displayed. In the current version the error is described in terms of the translated textual form, which may not be comprehensible to the programmer. In a future version, we plan to display error messages in terms of the original hyper-program.

6 Future Work

The printing of hyper-programs and the transferring of hyper-programs from one system to another is hindered by the presence of hyper-links. It is, however, possible to translate each hyper-program into HTML, representing the hyper-links as URLs. This was done to publish the Napier88 compiler source, which is itself a hyper-program, and it is our intention to do the same for Java.

The hyper-programming system described here is a further step towards our goal of an integrated programming life cycle using *hyper-code* [12]. The hyper-code abstraction allows a single program representation form, the hyper-program, to be presented to the programmer at all stages of the software development process. In constructing a program, the programmer writes hyper-code. During execution, during debugging, when a run time error occurs or when browsing existing programs, the programmer is presented with, and only sees, the hyper-code representation. Thus the programmer need never know about those entities that the system may support for reasons of efficiency, such as object code, executable code, compilers and linkers. These are maintained and used by the underlying system but are merely artefacts of how the program is stored and executed, and as such are completely hidden from the programmer. This permits concentration on the inherent complexity of the application rather than that of the underlying system.

7 Conclusions

The aim of this paper is to report on the transfer of hyper-programming technology from the persistent programming language Napier88 to an object-oriented environment, and in particular the orthogonally persistent Java system, PJama. The component technologies required for hyper-programming include linguistic reflection, a persistent store, and a persistent store browsing mechanism. All of these have already been designed, implemented and published. The additional components reported here are: the specification of denotable hyper-links in Java; a mechanism for preserving hyper-links over compilation; a hyper-program editor; and the integration of the editor and the browser with the hyper-programming user interface.

The differences in implementing hyper-programming in Napier88 and PJama revolve around the amount of code that is implemented in the underlying systems and the amount implemented in the languages themselves. In Napier88, the implementors had full control of the system and could implement at any level of abstraction that they chose. In fact, they took the decision to implement as much as possible in the language, providing only the essentials in the underlying system. Interestingly enough, the provision of the core reflection package and dynamic loaders, together with the language facility for casting from and injecting to class *Object*, allows the same implementation decision to be made in Java. This is either a shared insight based on experience or mere coincidence.

One minor drawback in our implementation is the use of AWT objects, which cannot be made to persist in the current PJama implementation. We hope to overcome this in a future release of PJama with the *Swing* classes.

---

[1] The principal class and the main method may be specified by the programmer; by default they are the first class defined in the hyper-program, and the method ***static void*** *main(String[] args)*.

Linking to persistent objects at program composition time would, at first glance, seem to be at odds with the object-oriented paradigm of delayed binding. Hyper-programming does not, however, reduce the range of linking time but indeed extends it to program composition. Delayed binding made be preserved, where it is necessary, by either writing textual code or by linking to a location that contains an object. In the latter case, when the program is run the object that is currently contained in the location will be the one that is used.

Composition-time linking would also seem at odds with system evolution. In fact it can be argued that hyper-programming simplifies this activity. Evolving any system requires the evolution of programs as well as data. In Java this requires recompilation of the libraries and reconstruction of the files or databases. Since a hyper-programming system can ensure that the hyper-program source text is always available for any persistent class that was created within the system[2], it is possible to write an evolution program that updates the source, re-compiles it and reconstructs the persistent data using linguistic reflection. Indeed, in a transactional system it is possible to do this in a separate transaction while the system is live. These are ideas that we have experimented with in the Napier88 context [13].

Our final conclusion is that the hyper-programming concept does transfer from a higher-order imperative polymorphic persistent language to an object-oriented polymorphic persistent language. In both, hyper-programming allows linking to be performed earlier where advantageous, without loss of flexibility.

8 References

[1] G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, A. Dearle, A.M. Farkas and R. Morrison. Persistent Hyper-Programs. In *A. Albano and R. Morrison (ed) Persistent Object Systems*, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy. Springer-Verlag, 1992, pp 86-106.

[2] A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison and R.C.H. Connor. Persistent Program Construction through Browsing and User Gesture with some Typing. In *A. Albano and R. Morrison (ed) Persistent Object Systems*, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy. Springer-Verlag, 1992, pp 376-393.

[3] R. Morrison, R.C.H. Connor, Q.I. Cutts, V.S. Dunstan and G.N.C. Kirby. Exploiting Persistent Linkage in Software Engineering Environments. *Computer Journal*, Volume 38, Number 1, pages 1-16, 1995.

[4] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby and D.S. Munro. Napier88 Reference Manual (Release 2.2.1). University of St Andrews, 1996.

[5] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis and S. Spence. An Orthogonally Persistent Java™. *SIGMOD Record*, Volume 25, Number 4, pages 68-75, 1996.

[6] R. Morrison, R.C.H. Connor, G.N.C. Kirby and D.S. Munro. Can Java Persist? In *Proc. 1st International Workshop on Persistence for Java (PJW1)*, Drymen, Scotland, 1996.

[7] G.N.C. Kirby, R. Morrison and D.W. Stemple. Linguistic Reflection in Java. *Software—Practice & Experience*, Volume 28, Number 10, pages 1045-1077, 1998.

[8] D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson and S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. ESPRIT BRA Project 3070 FIDE Report FIDE/92/49, 1992.

[9] G.N.C. Kirby and R. Morrison. OCB: An Object/Class Browser for Java. In *Proc. 2nd International Workshop on Persistence and Java (PJW2)*, pages 89-105, Half Moon Bay, California, 1997.

[10] J. Gosling and H. McGilton. The Java™ Language Environment: A White Paper. Sun Microsystems, Inc, 1995.

---

[2] There are obvious difficulties in dealing with classes imported from outside the system—reconstruction of source by de-compilation of byte codes may be adequate for classes that have not been deliberately obfuscated.

[11] G.N.C. Kirby, R. Morrison and D.S. Munro. Evolving Persistent Applications on Commercial Platforms. In *R. Manthey and V. Wolfengagen (ed) Advances in Databases and Information Systems*, Proc. 1st ACM SIGMOD East-European Symposium on Advances in Databases and Information Systems, St Petersburg, Russia. Springer-Verlag, 1997, pp 170-179.

[12] R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, V.S. Moore and R. Morrison. Unifying Interaction with Persistent Data and Program. In *P. Sawyer (ed) Interfaces to Database Systems*, Proc. 2nd International Workshop on User Interfaces to Databases, Ambleside, Cumbria, 1994. Springer-Verlag, 1994, pp 197-212.

[13] R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby and R. Morrison. Using Persistence Technology to Control Schema Evolution. In *Proc. 9th ACM Symposium on Applied Computing*, pages 441-446, Phoenix, Arizona, 1994.