Visual Zero: A persistent and interactive object-oriented programming environment

J. Baltasar García Pérez-Schofielda,*, Emilio García Rosellób,1, Francisco Ortín Solerc,2, Manuel Pérez Cotaa,3

aFaculty of Computer Science, University of Vigo, Edificio Politécnico, s/n Campus As Lagoas, 32004 Ourense, España, Spain
bDepartment of Computer Science, University of Vigo, Edificio Fundición, s/n Campus As Lagoas-Marcosende, 36200 Vigo, Spain
cComputer Science Department, Calvo Sotelo s/n, University of Oviedo, 33007 Oviedo, Spain

Received 16 March 2006; received in revised form 12 March 2007; accepted 3 November 2007

Abstract

In this article, an ongoing research project held in the Computer Science department of the University of Vigo is described. Its main objective is to develop an interactive object-oriented environment, serving as a vehicle for learning object-oriented programming. It currently consists of a virtual machine, an assembler, compilers for two high-level programming languages, and an educational visual programming environment. Its main characteristics are (a) the support of prototype-based object orientation, which is a model of object orientation that actually wraps the class-based model; (b) the support for object persistence, which simplifies all input/output issues to the minimum, and (c) the availability of a new visual programming environment is an invaluable help. We the authors think that the combination of all of these possibilities, will lead to an interesting, useful tool that would be recommended for object-oriented teaching. The whole system has been successfully employed in a number of different courses, allowing students to concentrate on objects and their relationships from the very beginning, and thus helping them to achieve a high degree of knowledge about the object-oriented programming paradigm.

© 2007 Elsevier Ltd. All rights reserved.

Keywords: Visual programming; Object-oriented paradigm

1. Introduction

To the authors’ knowledge, there are not many studies about the suitability of programming languages for education in object-oriented technology. Considering structured programming, immediately Pascal [1] appears as the paradigm of a programming language designed for education. In the field of object-oriented programming, probably the Blue project [2] is the best known one. The Blue project team made a survey of relevant languages...
and finally they stated that it was needed to create a new one, because the existing languages were not suitable for education. However, they dropped the so-called Blue programming language [3], apparently because of its portability problems (it was only available in Linux and Solaris). Finally, they adopted Java, developing BlueJ [4,5], which includes a programming environment, something considered as very important for learning.

Although we the authors agree with the majority of the conclusions of the Blue project (and other supporting authors [6,7]), we believe that dynamicity (which normally excludes, apart from the Kevo language [8], strict and static typing and type safety [9]) can provide some advantages, such as the simplicity of an homogeneous model [10], which can bring to advanced students a deeper knowledge of object-oriented programming. Moreover, probably type safety is a must for students of the first levels of object-oriented programming [1,2,7], while we defend that dynamicity is more appropriate for higher levels [11].

Dynamic languages historically began their role in the object-oriented programming paradigm with the apparition of Smalltalk [12]. It was designed, as one of its main objectives, to be useful as an educational programming environment. The later apparition of Self [10], was an even more related approach. It presented a model of object orientation that was new at that moment: prototype-based object orientation. Actually this model was intended to wrap the class-based (traditional) model of object orientation, by defining some objects as moulds (prototypes) for other objects. Programming was carried out incrementally and dynamically, so there was no static type verification. Additionally, all programming was done graphically, by means of menus and accepting commands through the use of the mouse.

These kinds of languages were not widely successful because their flexibility and interaction characteristics made them inefficient. However, they were highly appreciated because of their educational capabilities, as well as Smalltalk. Indeed, we the authors defend that this kind of languages are very appropriate for being taught on advanced courses of computer science. The prototype-based model of object orientation is simple, and more important, homogeneous (specially when compared to the class-based model). Unfortunately, these kinds of programming environments (and languages) are not widely available. Some of them are platform-dependent (such as Self [10]). Some present a syntax that is not easy to understand for beginners and even for experienced programmers [2] (such as Smalltalk and Self). Other languages have just been abandoned (such as Kevo [8]).

Also related to education, another interesting feature is the support of any flavour of persistence [13], such as in Smalltalk and Self (however, very simple) and other modern environments such as Initial Object-Oriented Programming Language (IOPL) [7]. Persistence is basically an attempt to allow object saving and restoring in a transparent way. This transparency, in a so complex process that normally involves database managers, is considered [6,7] as an important contribution, smoothing the learning curve for students, as students do not have to care about storage systems. This aspect has been considered marginally by other authors [6,7] as an specific assistance for their environments, while it has been an important topic of research by the authors of this work [14–16], recently focused on education [11,17].

In this paper, an object-oriented, prototype-based virtual machine (VM) called Zero [11,18], specially designed for education, is presented. Dynamicity and prototypes provide higher levels of interaction, as well as a suitable support for persistence. Indeed, persistence (implemented following the container-based model of persistence, already implemented in Barbados [14,15]) is central to this project, used by students to save their work (i.e., their objects for future use) in a highly transparent way. It is even possible to share objects, while traditionally persistent stores are highly self-contained [6,10,12,13,19]. The persistence capabilities of the VM makes these tasks completely transparent, while it is still a multiplatform application, which assures its availability in virtually any operating system, being already implemented in Linux and Windows.

In the next section, the VM, the very heart of the system, is described. Immediately after, the basic capabilities of the system and its simple model of object orientation are presented. Then the visual programming environment and its capabilities are discussed, showing its characteristics (interactive programming) in an example of use that can be used as an exercise in a real class. Finally, related projects are discussed, immediately followed by the conclusions and future work.

2. The virtual machine

Zero was designed around a VM as the heart of the whole programming system. The design of a VM
(in contrast to a monolithic design), greatly guarantees the possibility of including new, independent modules such as compilers of new languages, documenters, etc. The VM itself is kept as small as possible, being then simpler for the designers of this programming system to offer characteristics such as the support of multiple platforms (as it is written in standard C++), and also multiple languages. Objects can thus be created by means of one language and retrieved for modification in another one.

Technically, the Zero VM is a reference-based processor (as shown in Fig. 1). This means that it mainly manages references, which can be copied among its registers. There are four specific-purpose registers (an accumulator, the this register, the register storing exceptions and the register storing the return reference for any method), and four general-purpose registers. An assembler and two high-level programming language compilers produce bytecode to be consumed by this VM. The execution of bytecode can be triggered by a command-line, environmental-free tool (such as Java [20], or C# [21], for example), or using the educative visual environment.

The Zero VM is based on objects, which means that the object is the working unit. Objects are bags of references that point to other objects; also, they store methods that can be executed when a message with its name is received. Specifically, there are no classes. In the model of prototypes implemented [10], objects are created just copying them. Inheritance is provided through a special attribute called parent, which points to the base object. This kind of inheritance is known as delegation [10].

Though this model can be apparently too simple, actually it is able to represent the class-based model [10], i.e., the one supported by “classic” languages such as C++ [22]. When implementing the class-based model, some objects will play the role of classes are being reserved for copying only, which is the way in that new objects are instantiated. This has been experienced in the development of the J-- compiler, which consumes source code in a programming language similar to (actually, a subset of) Java, and is able to produce Zero bytecode.

2.1. Object persistence

There is an important kind of object: the container. Containers are specially designed for storing other objects (even other containers), effectively defining clusters of objects to be stored on disk [14], and allowing access to this objects while being resident in main memory.

There is a special container called Exe, which stores those objects being executed, and that is transient (i.e., not persistent) by construction (as shown in Fig. 1). Any object intended to be persistent must be linked to an existing container, by means of simple references. It is enough for an object to be linked to any other object that is inside a persistent container to become persistent as well. This is called persistence by reachability, which is one of a total of three rules of the so-called orthogonal persistence [13]. The other two rules say that (a) objects must be allowed to persist, regardless of its type, also fulfilled by this system; and (b) persistent and non-persistent objects must be managed the same way, regardless of whether

![Fig. 1. Structure of the Zero virtual machine. At the left of the image the registers appear, with the standard library, which happens to be always reachable. In the middle, there is an hypothetical method being executed, in the Exe container, and finally, to the right, the persistent store.](image-url)
they are persistent or not. This last rule is not fully accomplished by this system, as users must specify in which container the objects must be stored. However, the use of this model has been proved to provide some advantages, such as overall simplicity of the persistence model, and a simple way for the user to organize the persistent store [14,16]. That explains why it was included in Zero, as it is expected to help in education of object orientation [23], since object saving and restoring is one of the more error prone, and tedious issues in all programming languages, in which normally complex subsystems such as database managers must be used.

A proof of the independence of the persistence mechanism to the applications running in the VM or the language used to develop it is the ability to choose the use of the persistent store or not by means of a command-line option. The use of the persistent store is not mandatory, supposing therefore no penalty in the case of executing any application without persistence needs. Moreover, the compilers created for this VM know nothing about persistence: as discussed previously, the decision of objects being persistent is taken just by reachability, while containers are created the same way as any other object.

Containers are the main mechanism allowing Zero to support persistence, presenting a persistent store [24] divided in clusters at low level [14], as well as being represented by plain objects in the object world of the Zero VM, so they can be managed the same way than any other object, while they are actually a clustering mechanism [14,15].

From the user point of view they are similar to directories (of objects, in this case), mixed with the namespace concept in some object-oriented languages such as C++ [22]. As stated, this characteristic makes the container-based persistent system supported by Zero not fully orthogonal, though we the authors defend that this offers more advantages to programmers, as they can organize their persistent store in “folders” (container objects) as if it were similar to a filesystem [14], in contrast to a totally inorganized (orthogonal) persistent store.

Research in persistence has been actively carried out as a search for a homogeneous and transparent system for object persistence [14–16,25]. Persistence is therefore really interesting for educational purposes [11,17].

A transparent persistence system is highly interesting for learning, as it automatically resolves all problems related to storing and retrieving objects. The difference between primary and secondary memory gets fuzzy, as objects are loaded automatically when needed and saved when not. Students can therefore concentrate themselves on the core application instead of creating repetitive and error-prone code for their data objects.

2.2. Reflection

There are various kinds of reflection. Introspection is the capability of an object of reporting about its own members at run time in the form of plain data [26]. Structural reflection is the capability of any object to inspect its members and change them as well [11,12,17,26]. Computational reflection is outside the bounds of this article and consists of changing even the meaning of operators and other elements of a programming language [9,27].

Introspection is now supported (in a more or less complete fashion) by modern languages such as Java [20] and C# [21], and recent revisions of standard C++ [22]. However, structural reflection is only supported in a few object-oriented languages or VMs, related or similar, to the one presented in this document, such as Smalltalk [12], and Self [10].

Actually the Zero VM heavily relies on structural reflection. Indeed, it is central to the visual programming environment, as actually the environment is just a Zero VM with a graphic user interface. The addition of a new attribute to an object, for example, is carried out by sending the message addAttribute to the object (Fig. 6). However, the environment presents to the user some shortcuts for common actions, which triggers the execution of these methods automatically. The benefits of this approach are two fold: firstly, there is a simple and easy mechanism to do these kind of tasks, and secondly, the user is able to verify what operations (messages sent) are being carried out, which is highly educative, because he or she is able to repeat them later without using the automatic mechanism.

2.3. Dynamicity

The programming environment has been designed around interactivity and dynamicity, with the final objective of being useful for education. Objects can be created, erased, and changed in any way on demand, i.e., in execution time.
Complementary, this is possible because of the lack of static verification. This means that the compiler merely does not do any static type checking at compile time. For example, the message “obj.\texttt{f()}” could be wrong if object \texttt{obj} does not have any method called \texttt{f}, but could be correct if that method is created immediately before (for instance) the message is sent, with \texttt{obj.addMethod(’’f’’, v\texttt{args})}. Of course, compile time type verifications are impossible to be carried out, as there is a complete lack of types. A reference is not tied to a type (classes do not exist), and this means that it can be pointing to a string at a given time and to an integer some instructions later.

Lack of compile-time verifications involves positive and negative issues. On one hand, it is positive to have a programming environment, such as the one presented here, in which objects can be modelled quickly, adding attributes, and methods, deleting them, and so on, making the differences among edit, compile and run time very vague. On the other hand, it is difficult to program a large system using the traditional edit–compile–save cycle when nearly no compile–time checks are carried out. That is why a visual programming environment reveals as being so useful. The objective of a system such as this one is thus rapid prototyping, and, specifically, education.

2.4. The standard library

The VM and its companion compilers, and assemblers, would be of little use by themselves. A library including a number of objects, which represent a wide range of data, from very basic units, to complex data structures, is provided. The Internal Standard Library (IntStdLib()) container, includes the Object object, which is the root of the inheritance hierarchy. It also includes the objects String, Int and Float, (there are not primitive values, i.e., C# \texttt{value types} [21]). And finally, it provides objects for complex data structures (Vector, Map). The standard library (in the form of plain Zero \texttt{bytecode}) is needed to be present with the VM executable, or it is just included as another container in the persistent store when it is first used.

The main objective of the library is to provide programmers with a high-level abstraction of common data structures, as well as making possible to deal with the possibilities of the VM, such as getting the time of the system, managing the standard console, and others.

3. Using the basics of the system

The currently available programming languages for Zero are a simplification of Java, called \texttt{J--}, and \texttt{Prowl}, a dynamic language that actually just exposes all the capabilities of the VM without modification, from the perspective of a high-level language. Indeed, the following example has been built by means of \texttt{Prowl}, very suitable for dynamic programs. Since it uses the prototype model directly offered by the VM, it is therefore the better programming language in order to experience the prototype-based model of object-oriented programming. The process for object creation and execution is shown below. The “Hello World!” program (object) is also shown.

3.1. The “Hello, World!” object

Object creation is in this case limited to the \texttt{HelloWorld} object, which sends a message to the \texttt{Zero Console} object, pointed by an attribute of the System object.

Inheritance is expressed by a colon and the name of the parent object immediately following the name of the object (as shown in the first line). Methods are included inside objects using the reserved word \texttt{method} and a minus or a plus sign that precedes the name of the method, declaring a public or private method, respectively. The system objects grants access to the console, allowing to output text with the \texttt{write()} method and to add a new line with \texttt{lf()}. Exception handling is supported automatically inside any method, just by declaring an \texttt{onException} clause (which is not needed in this example and therefore it does not appear in the code below). When an exception is thrown, the control flow jumps to the exception handler of the method. If it is empty, then the exception is just rethrown to the caller method.

```
object HelloWorld :ConsoleApplication
    method +doIt()
    {
        System.console.write( “Hello, World!” );
        System.console.lf();
        return;
    }
endObject
$pwc HelloWorld
```

Prowl—High-level programming language for
3.2. The point of the model

In order to show the capabilities of this model, a simple example will be introduced. Specifically in this section, objects representing cartesian points will be created. They will have attributes for the representation of coordinates and methods for moving the point and obtaining its information.

The following object creates a prototype Point and an instance from it. Objects are created by means of copying another existing object. The latter receive the name of prototype, as it serves as a model for new objects.

```prowl
object Point
  attribute + x = 0;
  attribute + y = 0;
  method + moveTo(a, b)
  {
    x = a; y = b;
    return;
  }
  method + toString()
  {
    reference toret = "(";
    toret = toret.concat( x.toString() );
    toret = toret.concat( "", "");
    toret = toret.concat( y.toString() );
    toret = toret.concat( "") );
    return toret;
  }
endObject
```

However, one problem with this approach is that whenever a copy of any object is made, methods (such as `moveTo()` and `toString()` in `Point`) are copied as well. This is not very efficient, so students are taught to use the following approach.

```prowl
object TraitsPoint
  method + moveTo(a, b)
  {
    x = a; y = b;
    return;
  }
  method + toString()
  {
    reference toret = "(";
    toret = toret.concat( x.toString() );
    toret = toret.concat( "", "");
    toret = toret.concat( y.toString() );
    toret = toret.concat( "") );
    return toret;
  }
endObject
```

The modification from the last version of the program is just one extra object and the old object `Point` (which is still the prototype, now storing the state) renamed to `TraitsPoint` (which now does only store the behaviour). All new copies of `Point` inherit from `TraitsPoint`, so they have access to the methods stored inside it. However, the state is stored in `Point`, so all objects, by means of a copy operation, will have their own state; modelling, in this specific approach, the class-based object-oriented model (as seen in Fig. 2). That's why this model is said to include, or to be able to represent, the class-based model [28]. Students are then able to better understand how object-oriented programming works, from a completely new perspective.

This has shown the common edit–compile–execute cycle using Zero. However, the real strength
of Zero is not within this traditional approach to programming, but rather in a highly interactive, visual programming environment that is presented next.

4. Interacting with the visual environment

The educative programming environment presents a simple interface for the user to interact with. To the left, a panel that can be easily hidden and recovered holds the container explorer, the environment log and the Zero’s standard output. Also, some convenience buttons are provided as shortcuts for some useful routines: object creation, navigation to the parent of the current container, and a check box that will be marked when the VM is working, as a kind of visual indication (Fig. 3).

The VM can easily accomplish the operations supported by these shortcuts by means of structural reflection: for instance, by selecting the container-Owner member of any container, we will navigate to the parent container. By executing the createChild method of the Object object, a new object is created. As discussed before, these shortcuts are provided because their benefits are twofold: they are convenient for day-to-day use (allowing to program visually, without writing a single line of source code), as well as they show how these operations are carried out (a powerful learning assistance).

Indeed, the majority of programming can be carried out visually, by means of user interface elements. This is also done dynamically, i.e., there is no distinction between edit time, execution time and debug time. It is, however, needed to write source code for the business logic. This basically implies that the source code for methods must be still written. Writing source code is not a disadvantage, as the objective of the environment is to assist advanced students to acquire a deep knowledge of object orientation, in contrast to other aimed at novices [29,30].

4.1. Container explorer

The environment presents a main window to the user in which the container explorer is shown (Fig. 3). This explorer allows the user to navigate among containers and explore their objects. Its functionality is simple: double clicking its elements will open them for exploration (regular objects), or directly execute them (objects deriving from the Application object). Objects are visually identified by a folder if they are containers (as they can be understood as folders of objects), by a neutral
graphic if they are regular objects and by a gear if they derive from the Application object. This can be seen in Fig. 3, in which the HelloWorld object is recognized as an application.

Pressing the right button of the menu over any item can open a contextual menu. It contains options that make possible to open any object in the object inspector (see below), and add objects to the current container. It is also possible to modify the view of the items in the explorer.

4.2. Object inspector

Object inspectors are tied to a given object, being possible to have as many explorer instances as the user likes. There are various ways of opening an object inspector. Double clicking a regular object in the container explorer will open it in an object inspector. Also, right clicking an object and choosing explore will achieve the same, desired effect. Finally, the button with the glass in the main window will open an object inspector describing the container in the container explorer, as to the last extent, containers are objects as well. The explorer is divided in three sections, up to down: attributes, methods and inherited methods (shown in Fig. 4). By pressing the right button of the mouse over any of these three panels, a contextual menu will appear letting the user to view an attribute (which would open another object inspector with the object that attribute is pointing to, as seen in Fig. 3), or a method (the code of a method will be displayed, being possible to modify and recompile it, shown in Fig. 5); change the object an attribute is pointing to (Fig. 8); execute a method (Fig. 9), and finally add or delete attributes (Figs. 6 and 7) or methods (Figs. 11 and 12).

4.3. Object creation, message passing and inheritance

Any method of a given object can be explored by double clicking it, as discussed before, as well as it is also possible to execute it. When the user executes a method, the VM is actually sending a message to the object in which the method exists, again by means of reflection. Nonetheless, this process, as well as all other reflective techniques carried out by the VM, is transparent to the user.

Message passing indeed happens whenever users carry out any action. Moreover, by means of reflection, every time the user wants to execute a method a TopLevel object is created, with a doIt() method inside it which actually does the work (i.e., sends the message).

For instance, objects can be created by just clicking in the new button below the container explorer, which just triggers the Object.createChild() method, after asking for the name of the
new object. This new object will be stored in the Exe container. Another example is that new methods and attributes can be created by executing its addMethod or addAttribute methods or choosing the add attribute and add method menu options, which trigger the very same methods (Figs. 6 and 11), opening the dialogue window for the method’s arguments (as seen in Fig. 10).

New methods are added in order to extend the functionality of the objects they are created into. For example, the user could need to create a method for converting a given object’s information
into a string. In order to do that, after triggering the *add method* option of the contextual menu, it is necessary to double click the new method in order to open the method editor and type the instructions (string concatenation, in this case) to do the job.

Inheritance is provided through the *parent* attribute. This attribute, present in every object,
points to the object the first one inherits from. Inheritance is implemented through delegation [10,11,17]. This mechanism mandatorily sends unsatisfied messages to the parent of the object, meaning that delegates its execution on its parent. As the parent attribute is a regular attribute in all senses, it can be changed inside the programming environment as well. This implies another possibility that comes without any penalty: dynamic inheritance.

Dynamic inheritance [10] involves the capability of changing the parent of any object depending on the context. For example, an object Employee could derive from AverageEmployee or DirectiveEmployee depending dynamically on its salary. It could start deriving from AverageEmployee and modify its parent attribute to DirectiveEmployee when its salary grows above a given limit. This can be done interactively (directly modifying the parent attribute in the environment, described in Fig. 8), programmatically (overriding the putSalary() method) or by using the support for dynamic inheritance in the Prowl language, shown below.-

object Employee:
    AverageEmployee (salary < 20000),
    DirectiveEmployee (salary >= 20000)
endObject

When using the environment probably the first possibility is better as they are more interactive, while surely specific support (such as the one in Prowl) is more adequate and less error-prone than the others, especially when using the traditional edit–compile–test programming cycle.

4.4. Method modification

When a method is required to be modified, the code inspector is displayed (as shown in Fig. 5) and then the user can modify its source code. It is possible to compile or discard the code (pressing one of the available buttons), and in case of errors or warnings, these are displayed so the user can modify again the method as he or she likes.

This is similar to the Self-programming environment, where methods can be recompiled interactively, and actually changes the way that software can be developed. Instead of a classic edit–compile–debug cycle, methods can be modified at any time, as well as they can be executed to be tested and modified again if needed, until it is satisfied.

5. An example of use: the addressbook application

An example, a simple addressbook, will be discussed through this section in order to show the
interactive capabilities of this visual environment, as presented previously.

5.1. The common traits of persons

Firstly, a new object will be created to deal with personal information in the addressbook. Once the environment is started, it is possible to press \textit{new}, having \textit{object} chosen in the list under the container explorer, in order to create this object. The environment will trigger the execution of the \texttt{createChild()} method of the \texttt{Object} object, asking for its parameters. Actually, the only parameter needed is the object’s name, so the environment is thus asking...
for the name of the object. By entering “TraitsPerson” (including quotes), the object representing all persons will be created. An object inspector will open, showing the properties of the new TraitsPerson object. The explorer is, as discussed before, divided into three sections, up to down: attributes, own methods and inherited methods. Although the object has been created successfully, it has no information to hold, nor behaviour to offer. It is possible to solve this by selecting the add attribute contextual menu.

Fig. 11. Adding a method through the use of the contextual menu.

Fig. 12. Introducing parameters for the addMethod method, in order to add a new (empty) method to the object.
option (Fig. 6) or by choosing addAttribute in the inherited methods section, pressing the right button of the mouse and selecting execute. A new dialogue box (Fig. 7) will appear asking for the arguments the method will be invoked with. In this case, the name of the attribute must be provided, as well as the object it will point to. The name of this new attribute is “name”, which will be used in order to store the name of the person. So this is the value for the first box, always remembering that it is needed to enter it with double quotes, as it must be interpreted as a new String object. The contents of the attribute to be created are also a string, so for example “John Doe” can be an appropriate value.

The environment automatically creates special values such as strings (enclosed in quotes), numbers (as themselves), and vectors (enclosed in braces), before invoking the method addAttribute. The last step is to press Accept, in order to make the VM execute the given method.

When a method is executed, an object inspector with the properties of the object returned by the execution is displayed. In this case, it is not relevant, so its window can be closed, going back to the TraitsPerson object inspector. The new attribute appears now among the attributes of the object, the one that appears at the top of the explorer. It is possible to double click it in order to see the object it is pointing to, a simple new String object.

The same steps taken for adding the name attribute to the TraitsPerson object, can be taken in order to create the attribute email, using this time the values “email” and “John@doe.com” as arguments.

It would be useful to be able to change the name and the email of the person. That behaviour must be accomplished by a method. In order to add a method to a given object, the contextual menu option add can be used, as well as executing directly the addMethod method inherited from Object (as seen in Figs. 11 and 12). A dialogue box similar to the one that was presented for the attributes will appear. This time, the name of the method, and a vector with the names of the attributes must be provided. The name of the method is “setInfo”, and a vector (‘‘anName’’, ‘‘anEmail’’) must be entered for its arguments. Once created (pressing the Ok button), the new method will appear in the methods section of the object inspector, the one in the middle of the window. Its source code must be entered, which corresponds with the following Zero assembler (instructions can be entered by double clicking the new method): SET aName, ASG name, SET anEmail, ASG anEmail, changing each comma by a carriage return (similar to the method which appears being edited in Fig. 5).

This assembly code just assigns the object pointed by arguments and makes the attributes point to them. If the Compile button is pressed, then that instructions will become the new body of the setInfo method. Once returned to the object inspector for TraitsPerson, it can be executed as any other method in order to verify its behaviour.

5.2. The person prototype and a first instance

Now, using the controls below the container explorer of the main window, a new object, called Person, must be created in order to be able to generate new persons by means of a copy operation. This new and empty object will appear described by an object inspector if it is double clicked in the container explorer. In order to make it derive from TraitsPerson, the parent attribute must be selected, and clicking it with the right button of the mouse, the change option chosen from the contextual menu. Of the scrollable list in the assign object dialog, the TraitsPerson should be chosen (this process is shown in Fig. 8). In order to create new Person objects, it is just needed to press the copy button of the object inspector. After providing the name of the object, an exact copy of Person will be created (which is actually empty: the important concept right now is that it inherits from TraitsPerson). For an example of Person with one of the developers of Zero, the new name could be baltasar.

Once created, the object will appear in an object inspector. It is not linked to any container, so it will be lost as soon as the inspector is closed. Therefore, it must be saved (using the appropriate button in the inspector) to be stored in the current container. As expected, the button Save on the top of the object inspector must be pressed. The object’s name will now appear in the container explorer. Note in this example, the current container is Exe. Therefore, the object will be lost when the VM finishes its execution.

Please take into account that it is possible to avoid separating Person and TraitsPerson, having only one object, namely Person. But each time an object was created, all methods and attributes would be copied at the same time, making the copy very expensive. With this separation, methods exist only in one, shared, object (more or less similar to a
class), while the state would be copied only each time a new object was created [10].

The object baltasar has been created, though its attributes have not been set properly. In order to achieve this, the setData method should be called with the appropriate arguments, which are “Baltasar” and “jbgarcia@pleasenospam.uvigo.es”, respectively.

It should be worth noting that attributes name and email are now part of the object baltasar. This happens because, while not modified, they share the very same attributes with TraitsPerson. However, as soon as they are modified a new instance of these attributes is created in the object. This technique is called copy-on-write, which means that the duplication is done only when necessary.

5.3. The addressbook

Now it is needed to introduce this Person instances in a data structure, allowing some kind of searching. The natural key would be the name of the person, which makes a map (an association between strings and objects) the ideal structure. The standard library has a map, so that it is possible to create the new data structure just by inheriting from the Map object in the IntStdLib container. In order to achieve this, a new object must be created (for example, addressbook) and its parent attribute must be changed (as explained above) to point to Map, just by introducing its reference: Map (library objects are always reachable). The map already provides an add() method that takes a string and an object as arguments. A more convenient add() method, automatically taking the name of the Person object as index for the map would be really useful. Thus, a new method, called “add”, taking only an object as parameter ((‘aPerson’)), must be created in addressbook, with a new body presenting the functionality pursued (shown in Fig. 5):

```plaintext
SET aPerson.name, MSG __this ^add __acc aPerson, RET __acc.
```

This method just takes the name of the object passed as argument and calls the add method in the parent object (that is the meaning of “^”) with this name and the object itself. In order to store the object baltasar in the addressbook, the recently created method of this object must be executed, selecting baltasar from the drop-down list.

5.4. Making objects persistent

Although the main objects of the application have been created, they are stored in the Exe container. This container is always transient, which means that if we end the execution of the VM all the Person objects, the addressbook, etc., will be lost. In order to preserve these objects, they must be included in a persistent container. There are already some persistent containers available in the Zero persistent programming system. The root (psRoot), and the standard library container (psRoot.IntStdLib). None of these are appropriate for the needs of the exercise: the standard library should be only used to store objects like Map, String or Object, while storing objects of user applications in the root container is allowed but not correct: that behaviour would convert the persistent store in a mess of objects. The correct solution is to create a new container. In order to get to the root container, the parent container button must be pressed until the root container is reached (i.e., it is not possible to go up in the hierarchy any more, one click away from Exe). Then, “container” must be selected in the pull-down box and press new. A new container will be created under the root one, after entering its corresponding name that could be “Addressbook”.

Once the container has been created, the recently created objects should be stored inside it. In order to achieve this: the Addressbook container must be entered with a double click in the container explorer and, again in the contextual menu of the container explorer, add objects should be chosen. That will trigger the execution of the add() method of the container object. In the dialogue window, select addressbook from the pull-down menu. The addressbook object will appear now as part of the Addressbook container.

Inserting the addressbook object is enough to have all remaining objects (TraitsPerson and baltasarPerson) inserted as well, just because of the relations among them. The addressbook object stores a reference to baltasarPerson, while baltasarPerson stores a reference (through the parent attribute) to the TraitsPerson object. The VM will follow all these references to determine which objects must be made persistent as well inside the container (this means that the effect will not be seen until the environment is closed and reopened, though the objects can always be inserted manually). On the other hand, Person, the prototype object created as a factory of new persons, will have to be copied explicitly.

If the environment is shut down now, these persistent objects will be stored. When open up later, the Addressbook container will be available.
under the root container, with the previously created objects inside it. These objects are now also available through other programs created with the compilers or the assembler. This shows how easy it is to make an object persistent: just making it reachable from a persistent object does the job. From the point of view of the programmer, there is nothing as simple as this, not needing to worry about any saving/restore mechanism.

If a user interface is provided (which can be built with this programming environment or by any other of the available means), then the addressbook application will also be available for execution just by using the command-line VM.

6. Evaluation

The evaluation of Zero has been carried out by collecting the feedback given by students using the system. This collection has been carried out from 2004 on, within the subject Object Technology of the Computer Science Faculty at the University of Vigo (though Zero has been briefly used in other seminars as well, even at other Universities). This involved groups of 30 students each year, of which all of them completed the surveys. Before and after the seminar, a pretest and a posttest [11] was delivered to students in order to check, basically, whether they (a) had found the system helpful for understanding of object-oriented programming, (b) had found the system useful for learning, (c) which characteristics would they improve. These tests included various multiple-choice questions, some of them aimed at evaluating how students agree with the question, others just to know whether the student was able to answer fundamental questions about the prototype-based model of object orientation.

The kind of questions in these questionnaires is divided in two big groups, though all of them are of the multiple-choice type. The first ones are aimed at stating the knowledge of the student about some specific issues. The second ones give choices in a range from total agreement to total disagreement (three or five possible answers). This last kind of questions is divided as follows: in the pretest (eight questions), five questions have three answers, and one question has five possible answers. In the posttest (thirteen questions), seven had three possible answers, and another three had five possible answers.

Although nobody (0%) initially thought they knew deeply this model, there is a significant increase, in the posttest, of the percentage of people who thought they had some knowledge (with a final percentage of 93% of the total of students [11]). Questions about the utility of these seminars for their careers shows how students give value to the opportunity of being able to deal with an implementation of the kind of system explained in the theoretical classes. They undoubtedly answered “yes” (100%) about the appropriateness of using this software in order to learn the main concepts of this new model, in an empirical way. Another question was aimed to know whether the practical seminar was going to be (or had been), useful (respectively) for the students, and it particularly shows the satisfaction of the students with the practical seminar. In the pretest, the 68.75% of the students think the seminar will be useful for them, while the rest are not sure. Finally, in the posttest, the 100% answer positively the same question; therefore thinking the seminar had been useful for them.

Other questions appear only one time, within the posttest, trying to catch global satisfaction or unsatisfaction. For example, a question asks the student whether he thinks his knowledge about the prototype-based and class-based object-oriented programming has been improved. All students (100%) answered affirmatively. Moreover, when asked about whether this seminar had changed a bit their perception of object-oriented programming, a 75% answered positively, confirming the idea of the prototype object-oriented model being useful for a higher understanding of the object-oriented model in general. Another question asks students what this model of object orientation is good for. In previous years [11], there were an important percentage of people answering that this model was only useful from a theoretical point of view. Since then, the percentage of people giving that answer has decreased importantly, and the considerations of this model of object orientation being useful for teaching and even for some particular applications has grown steadily. This is explained by the fact that Zero has been developed continuously since then. While in the first years a student could only program the VM with a macroassembler, now high-level programming languages, and even an interactive environment, is available. These improvements have been steadily developed following the advices of the posttest in which students were
asked about the less good part of the system, or any part missing in their opinion.

Finally, about the tool itself, another question tried to state whether the system was simple to use. An 81.25% answered that it was simple, another 12.50% thought it was neither simple nor complex, and only a 6.25% thought it was complex.

7. Related work

As a system designed for education, the most similar one is BlueJ [4]. The main difference between these two systems is the dynamicity of Zero and the static strictness of Java. Although each model has its own advantages and disadvantages, we the authors think that the dramatic simplification of the model must be considered as a serious advantage for education. The programming environment developed for BlueJ is probably more advanced than this one, which is at a initial state of development. On the other hand, this environment is already able to support operations that BlueJ cannot support. An example is that in BlueJ, it is necessary to re-create objects from the beginning once any class of the project is recompiled, while that limitation simply does not exist in Zero. There is no support for persistence, either.

IOPL [6,31] is an educative system which uses a flavour of Object Pascal as a programming language, which implies that it is based on classes, and therefore out of scope of this prototype (they defend their system is aimed at novice students). It provides, however, orthogonal persistence through a centralised store in a dedicated server, only for the classes used by the students in their exercises. Apparently, the environment and the programming language are not detachable. This means that it is not possible to use other programming languages (though the system is able to unpars classes to C++, Java and IOPL from their internal representation), and that the student must abandon the system once he or she has earned enough knowledge.

Self [10] is the system in which Zero is inspired, mainly because of its dynamicity and programming environment. However, Self only support the syntax of this language that is not considered by the authors of this work as the ideal one for learning. Its programming environment supports transparent persistence just by dumping the whole memory to disk, and it is actually complex to save an object or group of objects in order to share them with others or make them able to execute in another environment. Finally, it is only available on SPARC hardware.

SmallTalk [12], is a programming system, in many ways a predecessor of Self. However, its programming language is a class-based one, which does not cover the objectives presented for the environment. SmallTalk also shares the same problem about persistence shown by Self.

Oviedo3 [32], is a VM quite similar to Zero, as it has also been partly inspired by Self. However, Oviedo3 presents a class-based model unsuitable for dynamicity and in general, nearly all other requirements expressed in this work.

PJama [19] is a persistent programming system that supports Java as its programming language. Although it is similar to this system in the fact that it supports persistence, the model supported is the pure orthogonal one, in contrast to the container-based model. Moreover, it is not a system designed for education, though obviously an environment similar to BlueJ, with the added capability of providing persistence, could be created on top of it, sharing then the same problems that BlueJ presents.

PerDis [36] presents a model of persistence quite similar to the container-based system, though PerDis is just a middle ware and not a complete programming system. Obviously, it does not have any educational capability.

Eclipse [33], NetBeans [34] and Visual Studio 2005 [35] are development environments growing quickly in popularity due to their refactoring capabilities and model-driven programming support. They are not specifically designed for learning, and though they allow interactive design, they do not allow the possibility of creating or deleting objects, or invoking individual methods, as in BlueJ or Visual Studio, supporting a sort of interactive debugging. In the case of that possibility being developed, they would share the very same problem stated above for BlueJ. Though these environments are highly flexible and support the addition of new features through the use of plug-ins, it is not possible to extend any of those environments in order to support features such as the ones supported by Zero, since they are not designed to support dynamicity in their programming languages. This is very important, as supporting a dynamic language it is possible to edit, compile, debug and run interactively and incrementally. Although in BlueJ (and Visual Studio), it is possible to execute
methods, and create objects, as soon as the user adds or deletes members in a class, all created objects will disappear, as recompilation is needed. This recompilation requirement clearly draws the line between what can be done in static programming environments, and in dynamic programming environments.

8. Conclusions

Zero’s visual programming environment has been presented here along with the VM capabilities and some issues of the object-oriented model supported. The highly interactivity and dynamicity of this system, makes it possible to maximize the possibilities for the student to learn how the object-oriented model works. The environment allows to do part of the programming task of application creation visually, sending messages to objects by means of the user interface. The possibility of inspecting objects makes the need of an input/output library disappear, until the application being developed is finished. Thus, programmers can just concentrate their efforts on the business logic of the application. Finally, the lines among the phases of editing, compiling and running vanish, as the programmer can test and modify the recently created method interactively, until he or she is satisfied with the result.

The use of a VM easily allows to create various programming languages, as well as other modules, such as the visual environment itself, documenters, etc. It permits to have a core that easily allows its port to new platforms, while the remaining modules are more or less independent. Indeed, it is now available for Windows and Linux.

This is to be added to the object persistence capabilities of the VM, making the programming experience a mixture of database and object-oriented programming. There is no need of using complex database managers, as the system itself is in charge of saving the required objects and restores them back when needed. This is very homogeneous, as it appears to be natural to the user, as well as simple.

This system has already been used in various subjects of the Computer Science Faculty of the University of Vigo, as well as in some doctoral courses at the University of Vigo and the University of Oviedo. A pair of questionnaires was delivered to each student, one before using the system and the last one after its use. The results extracted of these questionnaires are compiled elsewhere [11]. Basically, the experience of using this programming system was positively rated by various courses of students, while a higher understanding of the concepts of object-oriented programming was clearly demonstrated.

9. Future work

The educational environment presented here has been developed only for Windows, which means that while the system itself is multiplatform, this application is not. Efforts are carried out in order to create a new environment with a multiplatform user interface toolkit, so it can be made available to all users.

Another limitation is that the environment does only support by now the “natural” code of the VM (Zero assembly) for method modification. The code inspector just disassembles the method, allows modifications in its source code, an then assembles it back again. This is a hard limitation, provided that Zero is a multilanguage VM. We expect the new environment will deal with any programming language supported by Zero, for method bodies.

References


