Teaching Object-oriented Programming using Object Benches: Practical Experience

Jakub Livovský, Miroslav Biňas, Jaroslav Porubän

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia

{jakub.livovsky, miroslav.binas, jaroslav.poruban}@tuke.sk

Abstract. In this paper we summarize our long-term experiences with teaching object-oriented programming in university courses using teaching support tools called object benches. We describe four software tools: BlueJ, Greenfoot, Alice and Visual Studio’s Object test bench. Common feature of those tools is runtime access to objects, their attributes and operations. We present our experience with using object benches in programming courses, we evaluate the above mentioned tools in context of teaching object-oriented programming and compare them with our OAT tool we have developed to support object-oriented programming teaching. The comparison is based on several criteria - solution’s architecture, level of interactivity and domain specificity. As a result of the evaluation, we summarize the advantages and disadvantages of different approaches to design educational tools for teaching of object-oriented programming. We also discuss the main decisions behind the design and development of our OAT tool.

1 Introduction

Object-oriented programming (OOP) is relatively complex programming paradigm when we take into account the number of different concepts used in OOP [1], [2] (e.g. classes, constructors, attributes, operations, polymorphism, inheritance, subtyping). Therefore OOP teachers count on educational tools when learning OOP concepts and paradigm. A lot of research has been done in the field of educational software for teaching OOP [3] and many tools are available today [4], [5]. Amongst them the object benches are the most successful tools in teaching basic OOP principles.

Common features provided by object bench tools are runtime access to objects, their attributes and operations. Using object bench, users can create new instances of classes, access attributes of objects and execute operations on objects. User interface of those tools typically consists of three main components: standard class diagram well known from UML, object bench displaying existing/created objects and inspector providing access to attributes and operations. Using object bench the basic OOP concepts can be explained on practical examples. Object bench’s users can practice class instances creation using constructors and can manipulate created objects and
their relationships. At the same time user can observe impacts of his/her actions directly within the object bench.

2 Experiences with Object Benches

During last few years we used a few object benches in OOP course at our university. Following section characterizes these tools and describes our experience with them.

BlueJ is an integrated Java environment specifically designed for introductory teaching of object-oriented programming [6]. It is developed by La Trobe University (Australia), University of Kent (UK) and Oracle. Its aim is to provide an easy-to-use teaching environment for the Java language. Anyway, its source editor is not comparable with industrial IDE in terms of functionality. There were a few unsuccessful attempts to integrate the BlueJ with professional IDE. According to our five years experience with BlueJ, students want rather use professional IDEs. The side effect of using BlueJ for too long during the courses is that students will start to use it for development of more complex projects later.

Greenfoot is an integrated Java environment based on BlueJ. It is also designed for introductory teaching of object-oriented programming. BlueJ is focused on development of simple 2D games [7]. It provides much more interactivity with objects than BlueJ directly in developed game. We tried Greenfoot for one week long introductory course of simple game development for teenagers and they adopted it very quickly. Because it is based on BlueJ it has same disadvantages as stated in previous section.

Alice is a product of multi-university initiative to create educational software for teaching computer programming in 3D environment [8]. However, complexity of the IDE and 3D world interaction can be seen too complex a for novice programmers.

For one year we also used Object Test Bench (OTB), the integral part of Microsoft Visual Studio (VS). Because of its tight integration with VS, it has the biggest potential of all mentioned tools. The main problem was to achieve interactivity with objects painted on canvas (because of single thread architecture) and support for encapsulation (private fields were visible). OTB VS was probably not seen by authors of Visual Studio as teaching tool but more as a testing tool. If you wanted to use OTB VS, you had to install Team version of VS. However, OTB is no longer supported in VS from version 2010.

After a brief description of object benches, let us compare them from different viewpoints in the following sections. The comparison is concluded in the Tab. 1.

Architecture

The one of the most important thing when comparing object benches is its architecture. An object bench could be developed as a standalone integrated development environment for developing software applications (IDE), an IDE extension (plug-in), or a library. Let us look briefly at these approaches.

The most complex approach is to build an object bench as an IDE. While building the object bench as an IDE requires a lot of development effort it could be customized to the higher level providing all the features required for optimal teaching. Object
bench’s designers throw away any specific features not related directly to teaching of object-oriented programming making it simpler to navigate and use for a novice programmer.

Nowadays, industrial IDEs are very complex tools with lot of features supporting and simplifying the development of software applications, e.g. syntax highlighting, refactoring, code completion, code formatting, software visualization, unit testing, source code versioning. It takes significant development effort when one decides to build such a complex object bench as a standalone IDE. We should take into account that developed IDE should incorporate the today major IDE features. Otherwise students will be disappointed with the quality of the IDE and they will criticize it. When team decides to build object bench as a standalone IDE then teaching benefits from IDE usage must clearly outweigh all other mentioned disadvantages taking the cost of development into the mind. The features for the object bench IDE must be selected very carefully with respect to the most important practices and tools used in the industry and teaching methods at the same time. Finally, designer should also realize that industry tools are constantly evolving, so it will be necessary to evolve the object bench IDE to stay in touch with current trends. In our comparison, the BlueJ and Alice are representatives of standalone IDE approach.

Another approach to build an object bench is to extend an existing IDE with required feature for learning of OOP. Using these approach authors of an object bench can reuse a whole existing IDE infrastructure. This kind of object bench could be quite complicated because it provides all the features from extending IDE. It is good idea to use a possibility of an IDE to hide unimportant features simplifying navigation and understanding of a novice user and use IDE as a platform. Otherwise student can get stuck with all the unnecessary details for learning. The Greenfoot and Object Test Bench for Visual Studio are the representatives of this approach. The Greenfoot is built on the top of BlueJ IDE while OTB for VS is a part of Microsoft Visual Studio.

The last presented architectural approach is to build object bench as a lightweight standalone library. The object bench will be used as a library while running the application allowing introspecting objects and modifying running application. The library itself is an IDE independent and can be embedded into any regular Java application as a common library. On the other side, such a tool will lack the direct navigation to source code and does not contain source code manipulation support. The main benefit is that user can work in any IDE without any restrictions. At the same time tool and IDE can evolve independently.

Of course, it is easy to imagine that architectural approaches could be mixed. It is possible to develop object bench as a library and IDE extension at the same time benefiting from both approaches, i.e. lightweight architecture and source code navigation.

**Interactivity**

This property defines the possibility of changing object’s attributes and executing the object’s method while the application is running in background. Only Object Test Bench for Visual Studio does not support this behavior because of its single threaded implementation.
Domain

Some object benches are suitable for developing of all kinds of applications and some are domain specific. Object benches in our comparison are general-purpose except the Greenfoot designed for building simple interactive 2D computer games and Alice for developing 3D interactive scenarios.

Programming language

Since object benches are primarily used for teaching of OOP they have to support one or more object-oriented programming languages in which students express their solutions. The Java is currently the favorite object bench developer choice.

Visualization

Visualization of objects directly in an application can support the understanding of object programming concepts. The BlueJ and OTB VS do not have any special features/support for visualization of in-application objects directly in the application. On the other side, Greenfoot and OAT both have support for 2D visualization. Finally, the Alice is built around the concept of attractive 3D visualization. Alice’s authors are using the 3D visualization concept as an important motivating factor for learning, providing natural visualization of objects.

Table 1. Comparing object benches

<table>
<thead>
<tr>
<th>Tool</th>
<th>Architecture</th>
<th>Interactivity</th>
<th>Domain</th>
<th>Programming language</th>
<th>Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlueJ</td>
<td>IDE</td>
<td>Yes</td>
<td>general</td>
<td>Java</td>
<td>no</td>
</tr>
<tr>
<td>Greenfoot</td>
<td>IDE extension</td>
<td>Yes</td>
<td>games</td>
<td>Java</td>
<td>2D</td>
</tr>
<tr>
<td>OTB VS</td>
<td>IDE extension</td>
<td>No</td>
<td>general</td>
<td>.Net family</td>
<td>no</td>
</tr>
<tr>
<td>Alice</td>
<td>IDE</td>
<td>Yes</td>
<td>scenarios</td>
<td>alice / Java</td>
<td>3D</td>
</tr>
<tr>
<td>OAT</td>
<td>Library</td>
<td>Yes</td>
<td>general</td>
<td>Java</td>
<td>2D</td>
</tr>
</tbody>
</table>

3 The Object Access Tool

Coming out from our experiences with tools mentioned before, we created the object bench tool called Object Access Tool (Fig. 1) abbreviated as OAT. The OAT was primarily designed for use with 2D Java games. 2D games naturally contain objects with graphic representation and this fact can support student’s projection of objects via physical metaphor. For this purpose the OAT tool contains optional support for visualization of objects in both target application and object bench. Although OAT is primarily designed for 2D games, it is implemented as the universal library and it is possible to bind it to any Java application.

Architecture

For the OAT implementation we considered three different architectural approaches described in the previous sections. Right at the beginning we rejected the approach to create a new IDE. Like we stated before it would be a quite demanding task to
create and maintain an IDE that could compete with industry grade IDEs. Instead, we wanted to take advantage of those professional IDEs and avoid students’ transition from educational IDE to a professional one.

Both two remaining approaches have their advantages and disadvantages. Tight integration with an IDE is a significant advantage of IDE extension approach. In such a setting, user can directly navigate from an object bench to application’s source code and vice versa. It means straight navigation between visualized object in an application and object’s class sources code or navigation between selected attributed/operation in inspector and its source code. Tight integration would also allow object bench developer interact with many IDE functions, e.g. refactoring.

The advantage of a standalone library approach is in its lightweight infrastructure and minimum of dependencies. Library can be used with any development environment and is not dependent on any specific version of an IDE. The library can be embedded into any regular application.

Target application binding

Another important issue of the OAT tool development was the binding strategy to a target application. We considered two different approaches: defining a set of interfaces or use of aspect-oriented programming (AOP).

The main benefit of using AOP is in its composability. Using AOP we don’t have to modify the source code of an inspected application. Object bench application
adapter could be implemented as an aspect that injects all necessary code to target application. The fact that we can’t explicitly define the contract between the OAT and a user application is one of shortcoming of this solution. Our goal was to make a universal tool, which could target various applications. Because the architecture of each application is different, it is not possible to write universal set of pointcuts and advices. They have to be modified for each specific target application. The problem is we can’t formally define what has to be modified and how to modify it. This kind of modification would not be possible without detailed understanding of OAT implementation.

OAT was primarily designed to support teaching object-oriented programming - for example, to test the new classes implemented by student in an existing project. With help of object bench student can create and test instances of his classes without having to modify the existing source code. Using AOP in this case would require the aspect language compiler to be available in student’s environment due to weaving into new classes. It also should be noted that the aspect-oriented programming is not as widespread as, for example, object-oriented programming. A requirement to use AOP for binding with target application could discourage people from using OAT tool.

On the other hand interfaces are providing a method to define communication requirements between object bench and target application. Our solution consists of two interfaces specifying the contract between OAT and target application:

- InspectableApplication defines the communication from object bench to target application. As a consequence it also specifies functional requirements for target application. The target application adapter must implement this interface in order to OAT can access the application.

- ApplicationListener defines the communication from target application to object bench. Target application propagates events and changes to object bench through calling this interface’s operations.

The form of InspectableApplication interface implementation is left to the user. In our sample solution (Fig. 2) we implemented ApplicationListener by the adapter class - ApplicationAdapter. Adapter is extending target application with necessary functionality whilst monitoring changes in application and propagating them to observer implementing ApplicationListener.

Keeping track of all objects existing in target application is one example of a problem in which AOP would be more useful than interfaces. Object bench in OAT needs access to an up-to-date list of application’s object to work with. OAT tool was designed primarily for use with 2D games which is a specific category of target applications. Majority of 2D games contains one class representing the game world. Game world holds a list of actor belonging to this world. In this case it is relatively easy to implement this feature in Adapter.

The use of OAT tool is not limited only to games. In applications without central list of objects where objects are created on many different places, it would be much more complicated to keep track of all objects in adapter. This is where AOP comes in handy. With AOP it is possible to target all constructors executions to monitor creating of objects.
Saving the application state

One common problem of object bench tools mentioned in first section is that they do not keep the state of application. When a change in source code of target application is made, they need to recompile source codes and restart application. Which means the state of application is lost. All objects created with object bench are gone and they have to be recreated again by user.

We are trying to keep the state of application in our solution. Our solution needs to recompile and restart between changes too, but we are using the fact that the target application in our sample solution is a game. The state of game is represented by the state of its actors. We are using serialization for saving the state of actors between restarts. The main problem with using serialization is that our solution allows users to expand the target application. Due to this fact, we cannot rely on compliance of their codes with the naming conventions or the presence of annotations or default constructors.

Three approaches to serialization were considered - XML serialization (Java beans), standard Java serialization and custom serialization solution. The XML serialization is not suitable, because it requires presence of getter / setter pairs of operations with standardized naming. This solution would require modifying the target application so it satisfies this condition and we cannot guarantee this requirement in source codes added by users.

The problem with using Java serialization is that it is impossible to de-serialize saved object the class file was changed between application restarts. Java serialization calculates the hash code of the class files, and if it finds that they have changed, it won’t de-serialize the object. With Java serialization we can restore only objects whose code has not been changed.

We could implement our own solution for serialization into XML. By using reflection we could store the attributes and their values during serialization of objects and
we could re-set them during de-serialization. Thus, we could reconstruct all attributes that have not been changed. If only operations in the class were changed, we would be able to de-serialize the entire object to its previous state. The remaining problem is how to create a new object during de-serialization. User can create a new class without an empty constructor. In this case we would not know which constructor and arguments values to use for de-serialization.

4 Conclusion

In this paper we summarized our long term experience obtained during lecturing object-oriented course with object benches at our university. Object benches are important tool for OOP teaching, but they lack support known from professional IDEs. We presented the design of our object access tool with the aim of minimizing all of the disadvantages of existing object benches we have experiences with. If you decide to use our OAT, you will get traditional object bench tool for your favorite IDE with no extra limitations for target application you want to create.

Acknowledgement

This work was supported by KEGA Grant No. 021TUKE-4/2011 Platform for integration of study guides and tools within the learning process.

References