Using the methodology *Design Patterns First* by prototype testing with a user

Rudolf Pecinovský

Abstract:

The agile methodologies are more and more popular. One of their traits is embracing users in the development team. To be really useful users should be able not only to evaluate the design of a user interface but they also have to understand the programming techniques of business processes. If we could show users not only the results, but also how *the program solves the particular processes, we could obtain from them feedback of much higher quality and we could discover some errors much sooner. The article shows how we can teach user to behave correctly inside our program and work here with the program's particular instances – perform instance level modeling and testing. It shows how use the methodology* ***Design Patterns First*** *to teach him to analyze the program behavior without learning how to program.*

Key words: *Design Patterns First, prototype testing, methodology*

1 Our goal

Nowadays the agile methodologies of program development are more and more popular. One of their traits is their attempt to embrace users in the development team. To be really useful users should be able not only to evaluate the design of a user interface but they also have to understand the programming techniques of business processes. Only then they would be able to act valuably as opponents in the design.

Usually the future users discuss with analysts, who transforms the users' inputs into a form understandable by programmers. When the programmers then design the solution, it is often useful that users verify that it is right or suggest where it should be enhanced. Presentation of only the resultant behavior can reveal many mistakes and errors however there are many others that remain hidden because the used input data configuration doesn't allowed them to appear.

If we could show users not only the results of the programmed processes but also how the program solves the particular processes, we could obtain from them feedback of much higher quality and we could discover some errors much sooner. However for this activity we have to teach users to behave correctly inside our program and work here with the program's particular instances – perform instance level modeling and testing.

However if we want to show users the entrails of our programs, we should teach them first to be able to understand at least general features of them. Our experience has shown that we can introduce users to basic principles of OOP in a similar way that we use to introduce students in introductory courses of object oriented programming. However we should not reveal this to users because they become suspicious that we want them to learn programming, which they rebel against.

We need to persuade them that everything we do during their training is only playing with the programs, because they understand better than we do the purpose of the developed program and therefore they can give us valuable feedback. They don't need to write a single line of code. We only need them to be able to recognize, where the behavior of the program

differs from that expected and suggest to us, what we need to make better and what behavior we need to modify to better correspond with the requirements.

In this task two things significantly help us:

- ☞ the methodology of teaching, which we used to introduce students into the world of OOP and which begins with playing with objects and simulating their behavior on a model (see [4], [6]),
- ☞ programs developed originally for teaching introductory courses of object oriented programming and supporting the above mentioned simulation of program behavior including instance level modeling and testing.

However we will not (in contrast to students) teach the users syntax of a language. We teach them only to think in the object oriented paradigm and to understand the function of developed programs.

In teaching students we use the methodology *Design Patterns First* (see [4], [6]), which were created as an improvement of the older methodology *Object First* (see [2], [3]). The methodology *Design Patterns First* also starts with the interactive mode where students work with a prepared program as if they are part of it, that means as if they are one of program's objects and so are able to send messages to other objects. However immediately after the introduction to objects this methodology, in opposition to the *Object First* methodology, continues with introduction to the next important term, which is the interface. This knowledge then allows us to incorporate teaching of design patterns at the very beginning of the courses, which means, even before any attempt to write a program in any programming language. Students gradually create instances of classes, which implement different interfaces and learn how these objects can be passed as parameters of messages and how we can, thanks to the interface, unify several types of objects under a common roof to be passable as parameters of sent messages.

We act in a similar way at introductory meetings with users. After a very short introduction of object oriented paradigm we introduce users to their program and show, which objects act there and which messages these objects send one to another. Users then often bring additional specifications to the original assignment. These can be quickly incorporated and at the next meeting these enhancements can be demonstrated.

Let's show how we could introduce the users to the world, where programmers think, without teaching them programming. Let's show how to teach them the key programmatic constructions and so make them a really useful development team member.

The following text is therefore prepared as a very short teaching text for the user, who we want to introduce to an object oriented world of programmers developing programs on the Java platform. It can therefore serve as a manual for the person who cares about the functional incorporation of a user into the development team. If a section containing an explanation for the teacher is included, it will be typeset in italics.

2 The object oriented world

Object oriented programming is built on the realization that every program is a simulation of a real or a virtual world.

2.1 Objects

Each world consists of **objects**. If the program should simulate the actions in this world, it has to be able to work with objects. In real life we are willing to take as objects persons, animals and things. However, OOP generalizes this understanding and takes as objects also properties (colors, aromas …), events (connection, interrupt …), states (quiet, movement …) and generally **everything that we name with a noun**.

2.2 Classes

In larger programs there are thousand and tens of thousands of objects. If we want to effectively work with them, we have to sort them. If you have all your papers on your table in one heap, you can't work well with them. You will probably first arrange the papers into some thematic groups.

In larger programs there are thousand and tens of thousands of objects. If we want to effectively work with them, we have to sort them. If you have all your papers on your table in one heap, you can't work well with them. You will probably first arrange the papers into some thematic groups.

Several programming languages introduce a special type of objects called classes. Each class defines the common properties and features of its objects (instances) and besides it also offers tools for creating its objects (instances). So we can take classes as templates for their instances as well as factories for creating their instances.

2.3 Messages

In the real word everything that happens is a result of interactions between objects – one object affects the other and it reacts with it. In OOP we simulate these interactions by sending messages.

In the context of messages it is suitable to mention, that the part of a program, which defines the reaction of an object to a message, is called **method**. Programmers often don't talk about sending messages, but about calling methods. Both have the same meaning, its usage depend only on the actual context. When we analyze the problem at a level of abstraction, which is near to the user, we prefer the term **sending message**, during analysis at a level of abstraction near to programmer we prefer the term **calling methods**.

Objects don't react to any message, but only to messages, for which the programmers have defined the appropriate method. Each class defines a set of methods and "equips" its instances with them. So, all instances of a given class are able to react to the same set of messages.

2.4 Instances and references to them

When we send a message, we should always exactly determine, which instance is the addressee. In modern programming languages we get to instances through references to them. If we therefore want to send a message to an instance, we need first have a reference to this instance.

3 First simulations

Let's stop theorizing and show a simulation of some actions in the *BlueJ* development environment (see [1]) that we use in the introductory courses of object oriented programming in Java language.

We start demonstrating everything with a project simulating a simple world of geometric shapes. Its advantage is that here we can demonstrate everything really illustratively. It helps the easy understanding of basic principles.

3.1 *The development environment* BlueJ

As I've said we use for simulations of the program behavior the integrated development environment (IDE) *BlueJ* (see figure 1). This allows us to run and test many program actions without occupying ourselves with studying the code. *BlueJ* uses for the primary view of a program its class diagram, where each class is shown as a rectangle divided horizontally into two parts:

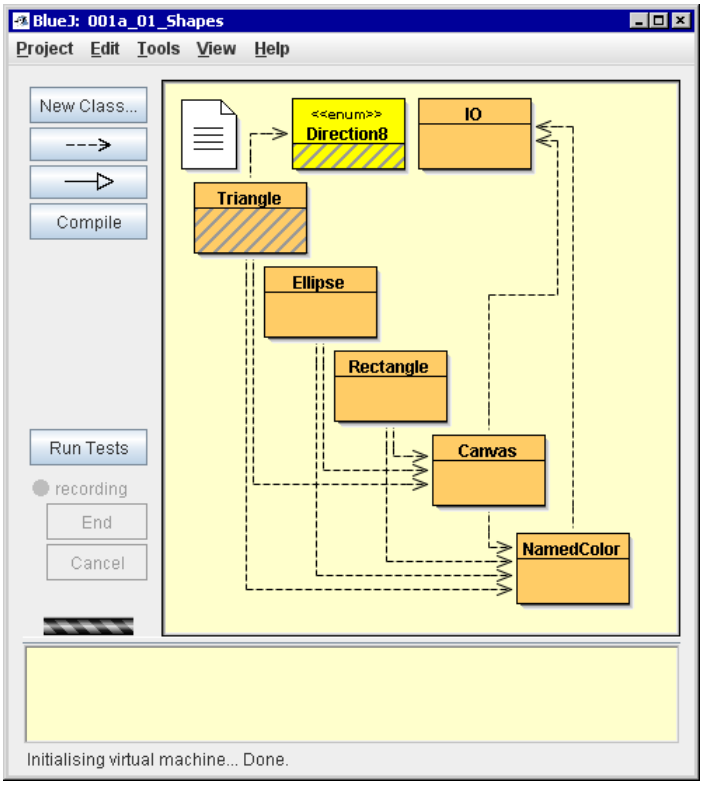

Figure 1: Simple world of geometric shapes

3.2 Introductory project

As we already said, each rectangle represents a class. There are seven classes in the project shown in fig. 1.

Some classes are connected to other classes by arrows indicating that the class, where the arrow starts, is dependent on the class, to which the arrow points. Such an arrow appears e.g. in the case, when instances of the dependent ("pointing") class send messages to instances of the "pointed to" class. It's because when I need for fulfilling my tasks the help of another object and therefore send a message to it, I became dependent on it, because I am not able to fulfill my task without its help.

3.3 Sending messages

We said that the OO programs implement the actions in the simulated world by sending messages between objects. *BlueJ* shows, in the object's context menu, the list of all the messages the object understands and we can therefore send them to it.

For now the only objects we can send messages to, are classes. Every object's context menu is divided in two parts (see fig. 2 – in this figure both parts are again divided; we'll explain it soon):

- ☞ The black commands in the upper part represent messages, which we send to particular object by entering this command.
- ☞ The dark red commands in the lower part represent the messages, which we send to IDE.

We start with sending simple messages, which don't need any parameters. And because the most interesting feature of most classes is their ability to create their instance, we start by asking for it.

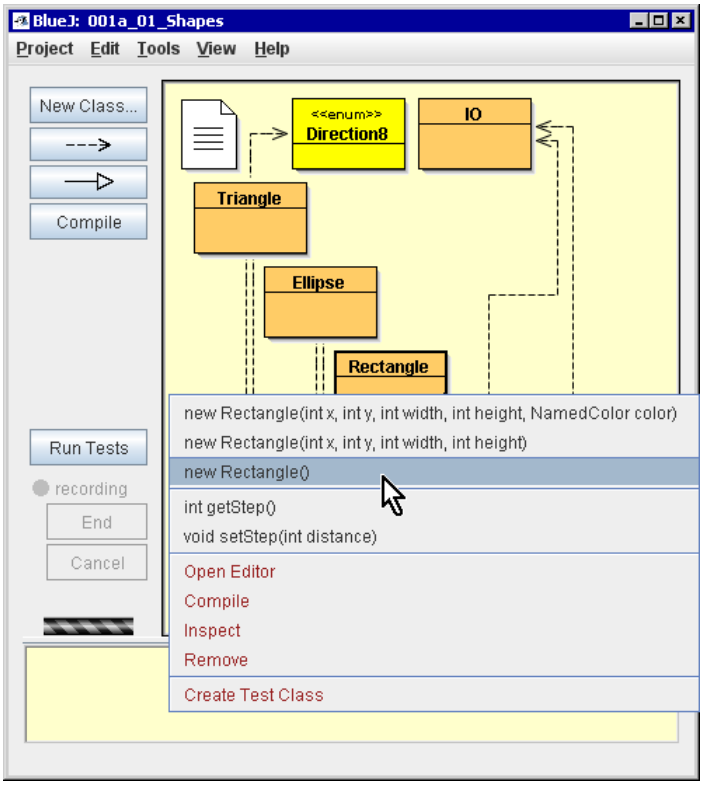

Figure 2: Context menu of the class Rectangle

3.4 Creating of an instance

We send to the class `Rectangle` the parameterless message asking for creation of a new instance, which means the message:

```
new Rectangle()
```

BlueJ expects that we will want to work with the created instance in the future and therefore we should save the reference to it. *BlueJ* therefore first asks us for the name we want to assign to the saved reference (and offers a name derived from the class name). Only then it sends the entered message to the addressed class (or we can say 'call its method') – the class `Rectangle`.

BlueJ saves the obtained reference into memory and creates in the object bench a rounded red rectangle representing the created object and having two lines of text: the entered name is in the first line and the name of class, of which instance the rectangle is, is in the second line.

We try the same operation once again – however this time we send our request to the class `Ellipse`. *BlueJ* asks us again first for the name of the asked object and only then it sends our message to the class `Ellipse`.

The same will happen when we send the parameterless message asking for the creation of a new triangle. Finally we have a shape consisting of a rectangle, an ellipse and a triangle painted on the canvas. Below in the object bench we have three references that we can use for our communication with the created objects (see fig. **3**).

3.5 Sending messages to instances

Now we can verify that we can send messages to instances in the same way as we did with classes. Also in the instances context menu we find messages associated to the instance and below them messages associated to *BlueJ*.

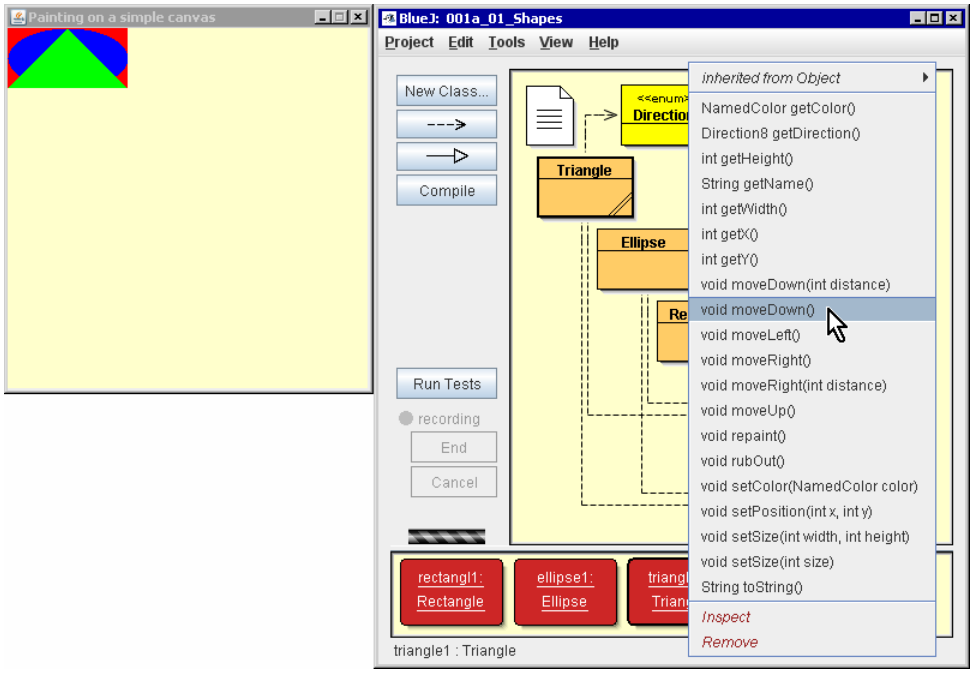

Figure 3: Sending a message to an instance

In the figure 3 a situation is shown where we want to send to the triangle the message to move down. To send a message to an object (similar to sending messages to a class) we should first open the context menu of the addressed object (in case of instances it is more accurate to talk not about objects but about references) and enter the requested command. In the shown case the triangle then really rubs itself out from the original position on the canvas and paints itself in the new position moved a little lower.

4 Test classes

After these first experiments it is time we should teach the user how to enter and save a command sequence so that they will not have to enter the whole sequence again next time. When verifying the functionality of the system under development we frequently repeat entering of testing sequences: first by their design and subsequently by verifying that the functionality hasn't changed during modifications of another part of the program.

However, here a quandary appears. The notation of the test steps is actually a program, but we promised the users that we will not teach them to program. We therefore have to teach them how to enter the requested message sequences in a way, which will be acceptable to them.

Fortunately one such way has been known for a long time and it is used e.g. when recording macros. The user only shows the requested sequence of commands and the computer then creates the appropriate program by itself. Luckily this possibility is built into BlueJ IDE where it is used in creating test classes.

4.1 Unit testing

One of most used ways of testing is the **unit testing** in which we test the behavior of particular parts of the program that we, in this context, call units. Units can be singular methods, classes or whole groups of classes.

For thorough testing we need to have prepared the tested objects together with the objects with which the tested objects cooperate and objects fulfilling auxiliary functions. We call this set of objects the **test fixture**.

We mostly make several tests with the same test fixture. Before each test we first create the appropriate test fixture then we test it with the particular test. Finally we clean up what is needed for running the next test.

4.2 *Unit testing in* BlueJ

When we have a program that we want to test, we should first create the test classes. *BlueJ* offers a useful function for creating test fixtures. It continuously watches (and records) what messages we send and to whom we send them. Therefore we can at anytime ask it to create a method which will remember everything we have done from the last reset of the virtual machine. There is a special command in test classes' context menus for this purpose – *Object Bench to Test Fixture* (see fig. **4**).

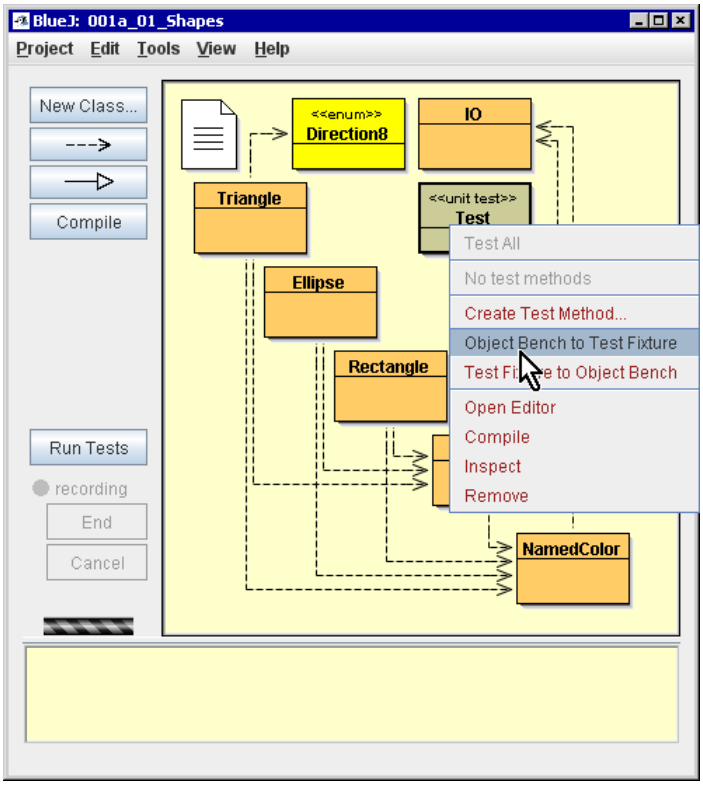

Figure 4: Creating of a test fixture from the record of sent messages

In the context menu there is the command *Test Fixture to Object Bench* under it. After entering this command the class creates a test fixture and places the references to all created objects into the object bench. So we can simply check that everything is created according to our needs.

However more important is the command *Create Test Method*. We enter it when we want to create a new test. Before entering this command it is important to clear the object bench – the best way is to reset of the virtual machine. After entering this command *BlueJ* first asks us for the name of the created method. Then it creates a test fixture and starts to record all the sent messages together with their addressee. After pressing the key *End* in the left pane *BlueJ* stops recording, creates a new test method with the given name in such a way that after calling this method the recorded sequence of messages will be sent to their addressee.

5 Messages returning values

As we said, if the object declares that it returns an object (value of an object type) as the reaction to a message, it in fact returns only a reference to the "returned" object. *BlueJ*

emphasizes this in the dialog box opened after sending a message asking for an object – it doesn't show any concrete value, but only an arrow representing the returned reference (see figure 5). We can try it by asking a graphical object for its color, which means we send it the message `getColor()`.

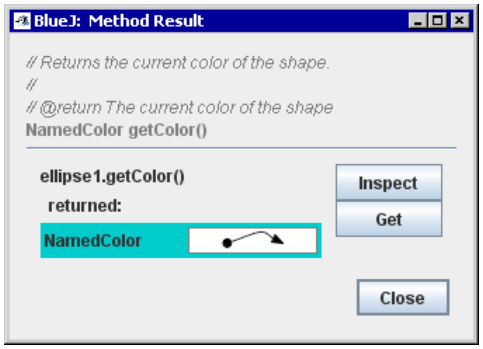

Figure 5: Dialog box with the reference to the "returned" object

At first sight it seems that we haven't got any information, because the dialog box tells us about the returned object only that we have a reference to it. However it is enough. By pressing the button *Get* we ask *BlueJ* to save the got reference to the object bench. *BlueJ* first asks for a name of a variable for this reference and then it saves the reference under this name in the object bench. Now we can send various messages to the reference and so verify that we have got the object (more exactly a reference to the object) we asked for.

5.1 Messages with more parameters

Messages may require more parameters. In such a case there is sometimes a problem not to mix up parameters and to ensure that the addressed object and its called method assign to each argument the right meaning. *BlueJ* tries to help the user with this problem. It divides the dialog box asking for entering parameters (and possibly the name of the reference to the returned object) in two parts (see figure 6).

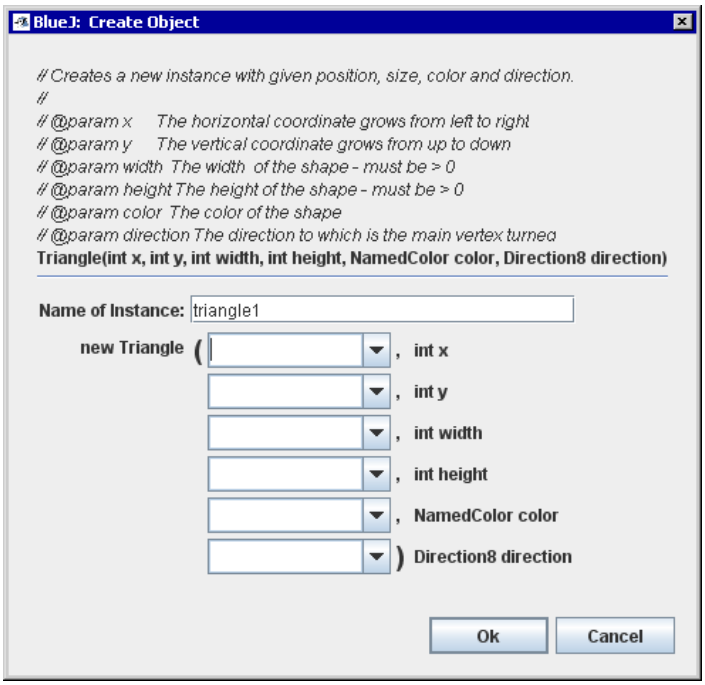

Figure 6: Dialog box allowing entering values of more parameters

- ☞ The upper part shows documentation of the called method and below it the declaration (in bold) of this method containing the type of the returned value, method name and types and names of particular parameters.
- ☞ The lower part contains a possible input field for the name of the variable where we save the obtained reference to the returned object (if the method returns any) followed by the set of input fields for entering particular parameters. On the right of each field there the type of appropriate parameter is repeated followed by its name to make for the user the entering of these values easier.

Input fields for argument values are implemented as combo-boxes whose lists contain recently entered values of the given type. It further simplifies entering of values by the user.

6 Interface

Let's for a while summarize what we already know and try to deduce what more we need. In the section *Messages* on page 3 we said that all instances of the given class are equipped with the same set of methods and that they therefore can react to the same set of messages. In the last section we said, that a message is uniquely identified by its name, number of parameters and types of particular parameters. In other words: the sender should pass in each parameter a value of type given in the message declaration. Similarly if the message returns a value, it should always return a value of the declared type.

The claims of messages to accept parameters of only the values of the declared type ties down our hands a little, because we have to define a new method for each type of value we want to use in the given parameter in order that the object will be able to react to a message with this particular type of parameter. So if we want to define a new class whose instances work with graphical objects, we have to define a separate method for each of our graphical shapes.

Speaking more specifically we give an example. In our project all three types of graphical shapes can move themselves to the specified place. However they move only by jumps. When we arrange for a new class, whose instances are able to move our graphical shapes smoothly (we name it e.g. `Mover`), it will have to have a separate method for smoothly moving each of our shapes: one for rectangles, second for ellipses and third for triangles. If we add to our project a new graphical shape (e.g. star), we have to define a new method in the class `Mover`. Nevertheless you surely feel that all these methods will be extremely similar; they will differ only in the type of parameter representing the object that will be smoothly moved across the canvas.

Let's revise what we have said about data types. We've said that the knowledge of the obtained value's type gives the program the information what the obtained object can do, what messages can be sent to it and what we can ask with these messages. But all our graphical shapes can do almost the same actions: inform about their position, size and color, repaint and rub out themselves, change their color, move and change their size. (Triangles in addition can inform about the direction which their vertex is turned to and set a new one, but for now we will ignore this small difference.) From a certain point of view we may therefore take all our graphical shape types as special cases of a more general type, whose objects can react to all the above mentioned messages.

In this context programmers distinguish two characteristics of a data type:

- ☞ **Interface** of a data type specifies what the object (instance) of the given data type can do, which messages it understands and how it reacts to them. It doesn't solve anything, it only promises what services the object is able to offer.
- ☞ **Implementation** of a data type is responsible for instances of this type behaving exactly as the interface promises.

6.1 Interface as a data type

Java introduces a special kind of data type – the `interface`. Interfaces have (unlike classes) no implementation. Theoretically they therefore cannot have any instances, because it is necessary to do something to create an instance, but interfaces don't have any implementation and therefore they cannot do anything. Fortunately Java allows classes to register for implementation of interfaces. They should then ensure that their instances can correctly react to all messages declared in the given interface and as a "present" they may pass their instances off as instances of the given interface.

Objects passing themselves off as instances of a particular interface may be passed in parameters to the messages, which require parameters of the type defined by the given interface. The interface promises that its instances will be able react to some messages and any class implementing this interface then defines how its instances will react to these messages.

Here a possible solution to our problem shapes up. Let's define an `interface` declaring the common abilities of parameters that the class bringing the additional functionality needs. When we take e.g. the above mentioned class `Mover`, it is enough that the moved object is able to move to the desired position. Mover then moves it a little, waits a while and moves it again just as a film does. The result of this sequence of actions is a feeling of a smooth move.

If we don't limit ourselves to a concrete enhancement of the functionality, we could define an `interface` in our project that declares all the methods defined by all our graphical shapes and then register all the shape classes to implement this `interface`. We name this `interface IShape`, because it defines a common interface (the staring I) of all graphical shapes. Then it is enough for the class `Mover` to define only one method moving the obtained object to the desired position – method moving an instance of `IShape` obtained as parameter. Such a method can move any of our shapes as well as an instance of a new class implementing `IShape`, because all these objects can pass themselves off as an instance of `IShape`.

BlueJ allows a simple declaration of implementation of an interface by a class. When you know that the class defines all the methods being declared by a given interface, it is enough to press the button with the arrow with the triangle head in the left pane, move the mouse pointer to the implementing class, press the mouse button and draw the arrow to the implemented `interface`. Because this declaration changes the class definition, *BlueJ* takes it as uncompiled and hatches it. At the next compilation the compiler checks, if the class really fulfills its promise and implements all the methods declared in the implemented interface. As soon the class is compiled you can create its instances and use them in all situations where the instances of the implemented interface are required.

When we afterwards add a new class implementing this common interface into the project, all its instances can immediately act as parameters of methods requiring instances of this interface. We needn't add or modify any definition, because everything that we need for incorporating the new class in the project is already prepared. In such a way we can prepare the project for many future enhancements required by the customer including some not expected.

7 Inside instances

During the explanation of messages returning values you may have wondered how the objects know the values we asking for from the messages we send. The asked objects can compute several of these values or ask other objects for them. However many of them they remember in variables called **attributes** (or **fields** or **member variables**).

A class determines which attributes will have its instances, so all instances of a given class have the same set of attributes. Instances cannot determine which attributes they have. They determine only the values saved in their particular parameters.

BlueJ allows us to take a look inside the object and find out, which values each object has saved in its attributes. The object inspector serves this purpose. We run it by entering the command *inspect* in the object's context menu. *BlueJ* then opens an object inspector's window (see figure 7) where it shows values of primitive types, enum types and text strings and allows us to look inside the attributes, which are instances of an object type.

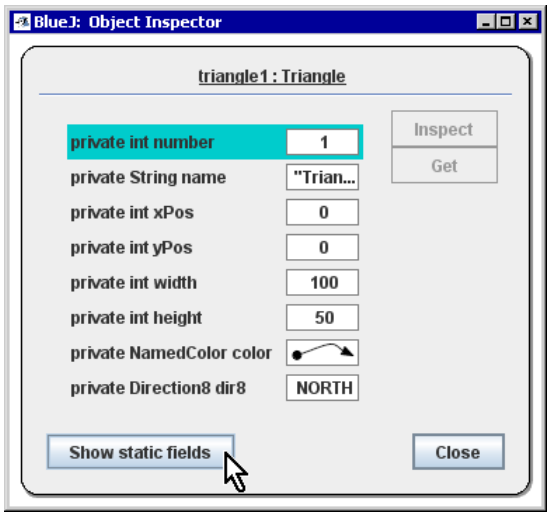

Figure 7: Object inspector window

8 More practical examples

The above shown crash course introduces users to fundamental terms and operations in object oriented programming. This crash course explained everything using the examples working in the project with graphical shapes. Experience shows that users quickly understand such "graphical" examples because in the graphical world everything is very illustrative and clear and therefore the users easily understand the explained subjects.

Now the introduced users should switch from the project with graphical shapes to a simplified version of the project, whose development they will cooperate on. Here they revise everything on objects from the problem domain that they are familiar with. They realize some relations, dependencies and connections which eliminate potential problems in the future and simultaneously they are introduced to some specific features and traits of programs solving problems of that problem domain.

9 Summary

This paper showed how we can incorporate users into a development team in such a way, to obtain much more valuable feedback from them. It demonstrated how to teach users to be acquainted with entrails of the program without the need to know how to program. It simultaneously offered a short demonstration of such a crash course that uses the educational IDE *BlueJ* that we use in introductory courses of object oriented programming. At the same time it explained that this tool can be used also in further analysis of the developed program and simulation of its behavior and it showed the use of this IDE for instance level modeling and testing.

References:

[1] *BlueJ home page*: http://www.bluej.org

[2] BARNES, D., KÖLLING, M. *Objects First With Java: A Practical Introduction Using BlueJ (2nd edition)*. Prentice Hall, 2004. ISBN 0-131-24933-9.

[3] KÖLLING, M. ROSENBERG, J.: Guidelines for Teaching Object Orientation with Java, *Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001)*, Canterbury, 2001.

[4] PECINOVSKÝ Rudolf: *Object Oriented Thinking in Java.* http://edu.netbeans.org/contrib/OOInJava/contents.html

[5] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java – kompletní učebnice pro začátečníky, 2. aktualizované a rozšířené vydání*, Grada, 2008. ISBN 978-80-247-2653-3.

[6] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: Let's Modify the Objects First Approach into Design Patterns First, *Proceedings of the Eleventh Annual Conference on Innovation and Technology in Computer Science Education,* University of Bologna 2006.

Author:

Rudolf, Pecinovský, Ing. CS.

ICS a.s.
Department of Education
Hvězdova 1689/2a
140 00 Prague 4

Faculty of Informatics and Statistics
University of Economics, Prague
Winston Churchill Sq. 4
130 67 Prague 3