greenfoot:
Combining Object Visualisation with Interaction

Poul Henriksen
Mærsk McKinney Møller Institute
University of Southern Denmark

polle@mip.sdu.dk

Michael Kölling
Mærsk McKinney Møller Institute
University of Southern Denmark

mik@mip.sdu.dk

ABSTRACT

The introduction of programming education with object-oriented languages slowly migrates down the curriculum and is now often introduced at the high school level. This migration requires teaching tools that are adequate for the intended target audience.

In this paper, we present a new tool for teaching object-oriented programming aimed at students at or below college level, with special emphasis of supporting school age learners. This tool was designed by analysing and combining the most beneficial aspects of several existing tools. It aims at combining the simplicity and visual appeal of Karel the Robot with much of the flexibility and interaction of BlueJ, while at the same time opening up possibilities of new applications.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education*

D.1.5 [**Programming Techniques**]: Object-Oriented Programming.

General Terms

Design, Human Factors, Languages

Keywords

Pedagogy, Objects-First, Micro worlds, Java, Visualisation, Experimentation.

1. INTRODUCTION

Introductory programming teaching has changed quite considerably since the widespread introduction of object orientation into first semester programming courses.

The acceptance of an 'objects-early' teaching approach, together with the inclusion of views that originate from the software engineering discipline (such as using 'larger' projects, using library classes, code maintenance) have led to a more complicated infrastructure that beginning students have to deal with. This frequently includes multiple source files, teacher supplied classes or frameworks, class libraries and environment setup to deal with compilation dependencies.

As a result, a number of tools have become popular that help students to deal with this increase in complexity of the infrastructure. These tools range from custom class libraries for specific tasks over thematic frameworks to full programming environments.

While a number of useful tools have been produced, there is considerable room for improvement. Specifically, the introduction of object orientation below the university level, e.g. in high schools, is a very recent development that might benefit from more targeted tool support.

The 'objects-early' movement has argued for a long time that it is important to teach good object-oriented practices from the start, to avoid having to correct or unlearn bad practices later. While this has led to a wide acceptance of teaching about objects early in first semester college courses, the target has since shifted: for many students, the introductory college course is not the first contact with programming anymore. Frequently, students get exposed to programming at the school level. If we are serious about teaching objects early, this is what we have to look at. We argue in this paper that goals and requirements for a software support tool for the school level differ from tools aimed at universities.

In this paper we discuss the design and development of a new tool named 'greenfoot' for introducing object-oriented programming to beginners. One explicit goal is the suitability for school age students.

By analysing benefits of several different approaches, and combining their respective strengths, we believe we can construct a system that offers a new quality of learning experience and has benefits beyond any individual system available today.

The greenfoot design was originally inspired by considering the combination of aspects of two popular teaching environments, Karel the Robot [12] and BlueJ [10]. One of the major strengths of Karel the Robot is the excellent visualisation of objects, their state and behaviour. It lacks, however, a means for direct interaction with objects.

BlueJ's tools provide strong support for direct object interaction, but the object visualisation lacks in detail and appeal compared to Karel.

Karel exists as a Java framework [4] and BlueJ is a Java environment. It is possible to execute Karel within BlueJ to get benefits of both. When doing this, however, confusing things start to happen. Since BlueJ offers a built-in object visualisation, and Karel visualises objects as well (in a different manner), a schizophrenic view of the world emerges. Many objects are represented twice (once by BlueJ and once by the Karel framework), and different aspects of visualisation or interaction are spread to different views of the same object. This has the potential to severely confuse beginners. Greenfoot combines the strengths of functionality without the resulting problems of representation.

We also show that the resulting system has qualities beyond the mere combination of Karel and BlueJ, but allows work with a more general class of interesting projects, including effectively a large number of simulation style problems.

2. RELATED TOOLS

Several existing teaching systems have influenced the design of greenfoot. The most important ones are briefly discussed here.

2.1 Karel the Robot

The idea of Karel the Robot was first published in 1981 by Richard E. Pattis [12], using a Pascal-like programming language. Since then it has been adapted to use various different programming languages [3][4], and several implementations are available for Java [2][4][6].

Karel is a conceptual framework that uses a robot that can move around in an environment in which 'beepers' and walls can be placed. The robot can then be programmed to perform a variety of tasks, such as collecting beepers and avoiding obstacles.

Programming and execution is usually done in a standard editor and execution environment. A running Karel program displays a graphical representation of Karel's world and a simple control panel to start or pause execution. Objects created by students – robots, beepers and walls – are represented graphically, and programmed behaviour can easily be observed. While the control panel offers some basic control of the execution, the objects themselves do not offer any interaction facilities.

2.2 Jeroo

One of the latest variants of Karel is Jeroo [13], which uses its own Java-like programming language. Jeroo adds several interesting features: it provides code highlighting of the source code statement being executed while running the program as well as limited capabilities of inspection. Like other Karel systems, Jeroo does not provide direct interaction with its objects.

2.3 BlueJ

BlueJ is a programming environment specifically designed for education. BlueJ encourages students to define classes and their relationships with an UML-like notation. Once classes have been compiled, students can interactively instantiate objects. These objects get a simple representation on an object bench. It is possible to inspect these objects, and execute their methods.

One strength of BlueJ is the clear separation of the concepts of classes and objects, and the possibility to interact with and inspect these.

The visual representation of objects in BlueJ provides only the name and class of the object – it does not display any visual clues about the object's state or behaviour. If the object creates its own visual representation (such as a robot in the Karel world), this is not reflected in the representation of the object on the object bench. Thus, BlueJ provides direct interaction, but does not provide direct visualisation of object behaviour or state.

2.4 Turtle Graphics

Turtle graphics is one of the oldest libraries used to introduce computing concepts to beginners. Originally developed for physical "turtle" robots, it came to fame via implementations in Logo [11] in the early 70s, and was soon available for use with several different languages.

The turtle graphics library includes the concept of a "turtle" that can move across a two-dimensional plane. While doing this, it moves a pen with it, which can be positioned on or off the ground, and thus may leave a trace of the turtle's movements.

Programming the turtle to draw different patterns can be used to introduce general computing concepts, such as iteration and recursion. Object-oriented concepts can also be mapped easily onto the turtle world [7].

2.5 Alice

Alice is an object-first approach to teaching introductory computer science courses [8]. It is based on 3D animations which can be controlled by students. Animation scenes are assembled from objects programmed by the students using a built-in programming language. To support the programming process, the environment provides an editor that allows students to drag and drop statements and objects.

When the object behaviour has been created, the objects can be inserted into a 3D world. Object properties can be inspected and changed before the animation is started. After the animation has started it is not possible to inspect objects and interact with them.

2.6 The Marine Biology Case Study

The Marine Biology Case Study is a small Java framework that provides a two-dimensional world structured as a grid of positions, and fish that populate this world [14]. Students can implement new fish behaviour, or new type of fish.

This case study is similar in character to Karel the Robot. It is being used mainly for the United States Advanced Placement Computer Science (APCS) course at the American high school level. The APCS course is coordinated centrally nation-wide in the US, and thus this case study has a large user base.

As with Karel frameworks, the Marine Biology Case Study provides immediate visual feedback of (some) object behaviour and state changes, but no direct object interaction.

2.7 Summary

We have briefly presented a number of well known tools to aid computing education. One of the strengths of BlueJ is the inspection and interaction with objects, but it lacks support for good object visualisation. Other systems, such as Karel or turtle graphics, provide good visualisation of object behaviour, but lack in interaction with objects. Our new tool will attempt to merge functionality from all these systems into a tool that has the best from each world.

The result should be functionality that allows BlueJ-like interaction on rich visualisations, with a flexibility to create diverse applications including turtle graphics style examples, as well as simulations like the Marine Biology Case Study and other kinds of simulations.

3. DESIGN CONSIDERATIONS

In this section we briefly discuss some of the background arguments and forces that had a strong influence on the design of the greenfoot system.

3.1 Closing Kolb's circle

Kolb's learning circle [9] presents a model often used to reason about the process of learning, including the learning of programming concepts. Figure 1 illustrates the four phases of the circle.

In early programming education, especially in objects-early approaches, it can be difficult to create activities in all four quadrants of the circle.
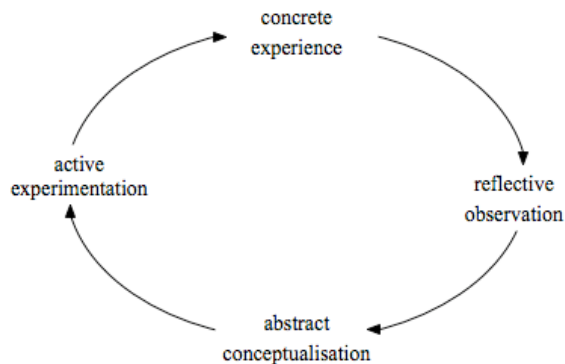

Figure 1: Kolb's learning circle

A lecture-style introduction to the concepts tends to focus on the area of abstract conceptualization. Creating active experiments and concrete experience (which then can lead to reflective observation) in a dedicated objects-early approach is often made difficult by technical obstacles. Obscure syntax and detail problems in the programming environment often force the concrete experience to be at the level of statements rather than higher conceptual abstractions.

Role playing is one technique that has been used successfully in the past to overcome parts of this problem in the first stages of a course [5]. In the design of a software teaching tool, we should aim at overcoming these problems in software as well.

Any tool to support an objects-early approach for beginners must attempt to support the practical stages of Kolb's circle (experimentation, experience, observation) explicitly at the level of the fundamental concept: objects.

In other words: students must be able to manipulate, experiment with, and observe *objects*, not merely lines of source code.

3.2 Creating interest

One aspect in which the target audience at a school level is significantly different from students at college or university level is the level of commitment to the subject.

Students at school level have not made a conscious decision to engage in the study of programming. They may frequently have no interest in the subject, or even hold prejudices against it.

Thus, the aim of a support tool for the school level is not only to illustrate the important concepts to students, it must also generate interest in the subject in the first place. The activities students engage in through use of the tool should be engaging and relevant to the students.

For both of these characteristics – engaging and relevant – there is no simple recipe defining what this means for any particular student.

There are, however, some general observations: a system that is interactive, visual, allows experimentation, generates curiosity, without requiring substantial prior theoretical study, is more likely to create student engagement.

Whether a system is perceived as relevant for any particular student depends highly on the student's cultural and personal background – we do not believe there to be a single solution to this goal.

For our system design, this means that we need to strive for a level of flexibility in scenarios presented to students that allows teachers to address students in locally or personally relevant ways.

3.3 Supporting teachers

Another aspect that distinguishes the teaching situation at schools from that at universities is the level of preparation that can be expected from teachers.

Computing teachers at school level often have significantly less training in the field of computing, less time and support for professional training to keep up with latest developments and less time to prepare teaching material.

As a result, it is beneficial to view teachers as a second target group to be supported by our teaching tool. Apart from supporting students in their learning, greenfoot should be designed to also support teachers in their teaching.

This can be done in various ways: we can arrange the presentation of important concepts in the tool in ways that encourage discussion of selected topics early. We can also design the tool so that sharing of scenarios and exercises becomes easy. We will discuss this in more detail below.

There is, however, a clear tension between flexibility (as discussed in the previous section to allow adaptation to students' backgrounds) and teacher support.

Supporting teachers tends to mean to provide a rigid framework in the support tool, so that teachers have less work to do themselves, and better guidance in their teaching activities. Allowing flexibility can directly contradict this.

One of the challenges of the greenfoot design will be to find a solution that allows these two goals to coexist.

4. GOALS

Before describing the design of our new tool, we summarise the design goals that guided the development of the concrete system.

The overall goal could be summarised as "suitable for teaching object orientation at the school level". After the discussion in the previous sections, however, we can now formulate this more precisely. We should also note that the focus on school level does not exclude use of this tool at early college level.

Our goals are:

- experimentation and visual feedback
- flexible scenarios
- clean illustration of object-oriented concepts

- easy development of scenarios and exercises
- support migration to other environments

Experimentation and visual feedback. We intend our system to be highly visual and interactive. Users should be able to experiment with instantiations of concepts directly via the user interface, and acquire an understanding of important concepts through direct visual feedback. This is hoped to close the circle of activities in Kolb's model, as discussed above, and also to contribute greatly to the challenge of engaging students with no prior interest in programming. Striking a balance between simplicity and richness of tools is important. If the program is not simple to use, the users could loose interest and use of the tool would be counter productive.

Flexible scenarios. To engage students interest, the system needs to be able to support examples relevant to their age, gender, personal and cultural background, and other individual factors. For the system design, this means that greenfoot must support a wide variety of different tasks and scenarios. Creating flexibility in scenarios also allows to vary the complexity and thus the difficulty level of the material to be learned.

Clean illustration of object-oriented concepts. The primary focus is to develop students' understanding of concepts used in object-oriented programming. Through the use of greenfoot students should get familiar with fundamental concepts of object-oriented programming, such as objects, classes, method invocation and imperative programming concepts.

Easy development of scenarios and exercises. Part of the larger goal of providing good support for teachers is that it should be easy to develop scenarios and exercises. Traditional systems, such as Karel the Robot or the Marine Biology Case Study, are of a scope that makes it impractical for most teachers to develop similar systems with alternative stories. Greenfoot should attempt to make scenario and exercise development easy enough so that many teachers can develop their own versions.

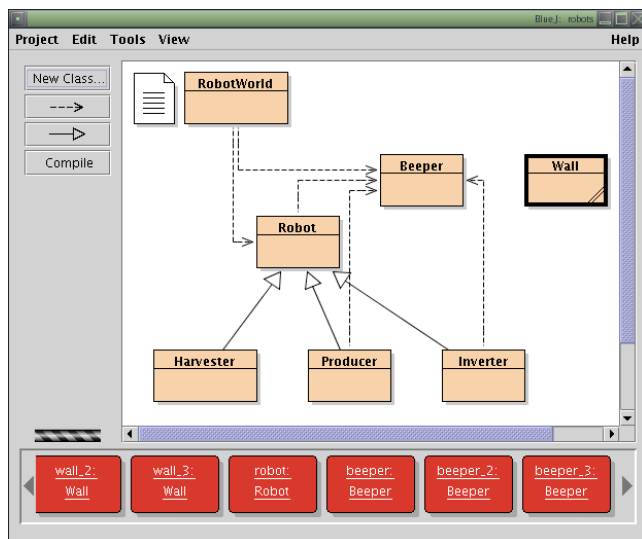This means that the user level scenario should be separated from the general framework implementation.

Figure 2: Robot, Wall and Beeper objects, as shown in BlueJ

Support migration to other environments. The greenfoot system should be designed so that concepts and skills learned can be transferred easily to other environments, such as BlueJ, which may be used as the next development environment.

5. THE GREENFOOT SYSTEM

The greenfoot system is a framework and environment to create interactive, simulation-like applications in a two-dimensional plane.

One way to view the system is as an extension of the BlueJ object bench. In BlueJ, the state of the object is observable only by opening a separate inspector window, which displays the object's field values. The main visualisation of the object – a red rectangle with rounded corners – does not convey any state information, neither via its appearance nor its position (Figure 2).

Greenfoot extends the idea of the object bench to an object world. In this world, all objects have a graphical appearance and a position in the world. Direct interaction with these objects is still possible, as in the original BlueJ system, but object behaviour can now be observed directly by observing changes in the position and appearance of individual objects.

The object world itself (visible as the background area behind greenfoot objects) also becomes an interactive, programmable object, thus being integrated into the application framework.
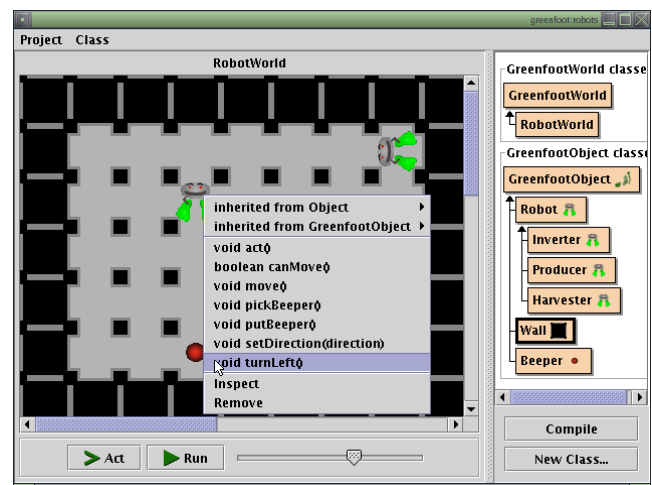

Figure 3: The greenfoot main window

5.1 The Visual System Interface

The largest part of greenfoot's user interface is reserved for the display of the world, shown in the centre of the screen (Figure 3). It holds the greenfoot objects (two greenfoot robots, a beeper and some walls in this example). To the right of the world is a class display. Here, all classes involved in the current application are shown along with buttons to compile and create new classes. The classes are divided into *Greenfoot-World Classes* and *Greenfoot-Object Classes*. Each of these groups is discussed below.

The lower part of the window holds execution controls to run, stop or single-step the simulation and a slider to control the execution speed.

5.2   Greenfoot Scenario Structure

All classes whose instances should be visible in the greenfoot world extend the predefined superclass *GreenfootObject*. The environment also provides a predefined class *GreenfootWorld*, which implements the world itself.

The world provides a grid of cells, which can hold greenfoot objects. Each greenfoot object can specify its own individual appearance using an icon or a drawing method. Greenfoot objects have a location in the world and a rotation that is applied to the icon. The appearance can span one or more cells.

The world itself provides methods, among others, to change the resolution (essentially setting the size of each cell in pixels), to change size of the world (number of cells), to set a background image and to draw on its background. Using these methods, worlds that differ greatly in visual appearance can be created as part of creating a scenario.

All objects in a greenfoot world are automatically animated and interactive. They can have behaviour that is exhibited when the simulation is run using the *Run* button, and they can be used for direct interaction through associated popup menus when the simulation is paused. The simulation animation and direct object interaction features are built into the greenfoot environment.

An object's popup menu contains a list of methods that can be invoked on the object as well as an option to inspect the full object state (Figure 3).

5.3   The Greenfoot IDE

The greenfoot system is an integrated environment: apart from the main interface described above, it contains an editor, a compiler and a debugger. Thus, it is a self-contained system that provides all necessary tools to develop, examine and execute a complete application.

The underlying runtime and compiler uses standard Java. Greenfoot classes are standard Java classes.

The greenfoot implementation is based on the BlueJ system, and many of the BlueJ tools – the editor, the debugger, *Javadoc* generation – are available in greenfoot in a very similar form to BlueJ.
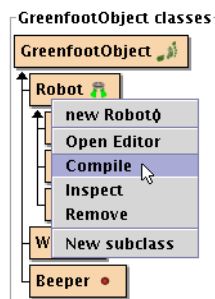

Figure 4: The popup menu of a greenfoot class

5.4   The Class Browser

The environment also supplies a view of the classes that participate in the simulation on the right side of the main window.

These classes can be edited, compiled and instantiated. These actions can be accessed from the popup menu of the class (Figure 4).

Creating new classes can be done either by selecting *New subclass* from the class' popup menu, or by clicking the *New Class* button below the class icons.

The class-browser is divided into two sections, discussed below.

GreenfootObject classes are the classes that are to be visualised in the world. Their superclass – *GreenfootObject* – will always be shown in the class browser. The GreenfootObject class cannot be modified.

Subclasses of GreenfootObject will typically have an individual icon. This icon is shown in the representation of the class next to the class name. Greenfoot objects that do not specify an appearance have a default look defined in their superclass.

GreenfootWorld classes are classes that represent worlds. Different worlds may exist in a single project (holding, for example, different initial populations of walls and beepers). The superclass of these – *GreenfootWorld* – will always be shown in the class browser.

The GreenfootWorld classes have popup menus exactly like the ones described for GreenfootObject classes. When a constructor is selected for a subclass of GreenfootWorld, the new world object will automatically replace the existing world in the main view of the greenfoot user interface.

6.   SAMPLE APPLICATIONS

The way we envisage greenfoot being used is for teachers to create custom scenarios, which students then investigate and extend.

Custom scenarios can resemble Karel the Robot, the Marine Biology Case Study or Turtle Graphics, or they can represent other simulation scenarios such as traffic or lift simulations.

Student activities include instantiating objects, running a simulation, interactively calling single methods, reading and modifying the code, compiling and creating new simulation object subclasses.

We will now describe some possible scenarios and discuss the activities expected of scenario writers, teachers and students in turn, in order to give an impression of the level of difficulty involved at each level.

First, we discuss what a Karel the Robot implementation would look like in greenfoot. This serves to illustrate some of the most important characteristics of this framework. The following two examples (*Shapes* and *Turtle Graphics*) are briefly presented to illustrate alternative scenario styles.

6.1   The Karel-The-Robot Scenario

First, we discuss in more detail a Karel-like scenario, which readers have already seen in the general system introduction above.

6.1.1   Scenario Writer Activities

Scenario writers are typically teachers, but since scenarios can be shared between teachers, not every teacher needs to be a scenario writer.

The greenfoot framework provides most of the functional aspects of the simulation and lets teachers focus on creating an interesting scenario.

When a new project is created, the class browser contains only the GreenfootWorld and the GreenfootObject classes. These are the base classes for all others created by the teacher.

First, we want to create a world for the robots. We create a subclass of GreenfootWorld, which is automatically initialised with a source skeleton, in which we only have to modify the constructor.

To create a world of size 20x20 cells with a cell size of 50x50 pixels and a tiled background image, the *RobotWorld* class looks like this:

```
public class RobotWorld extends GreenfootWorld
{
    public RobotWorld() {
        super(20,20,50,50);
        setBackgroundColor(Color.BLACK);
        setBackgroundImage("road.gif");
        setTiledBackground(true);
    }
}
```

All methods called in this constructor are inherited from GreenfootWorld.

The next task for the scenario writer is to create the core classes for the project: the robot, the wall and the beeper. The last two classes are easy to create because they do not have any behaviour. The only thing that needs to be changed are the objects' icons. The *Beeper* class then looks like this:

```
public class Beeper extends GreenfootObject
{
    public Beeper()
    {
        setImage("beeper.gif");
    }
}
```

Since a skeleton for this class is provided when it is created, the scenario writer needs to edit only a single line of code.

The class for the robot is a bit more interesting because it should have behaviour to control the robot in various ways: moving, turning, picking up and putting down beepers. The moving and turning of the robot is easy to do using methods from its superclass. For moving the robot, the `setLocation(int x, int y)` method can be used together with the `getX()` and `getY()` methods. Turning the robot can be accomplished using the `setRotation(int degrees)` method. This rotation will be automatically reflected in the visualisation of the robot by rotating the icon.

The robot should be able to collect beepers. For this we use a collection which we call *beeperBag*. To pick up a beeper at the robot's current location we create a method `pickBeeper()`:[1]

---

[1] We are using Java 1.5 syntax in this example. Syntax from previous Java versions can, of course, also be used.

```
public void pickBeeper()
{
    GreenfootWorld myWorld = getWorld();
    Set objectsHere =
        myWorld.getObjectsAtCell(getX(), getY());

    for(Beeper beeper : objectsHere) {
        myWorld.removeObject(beeper);
        beeperBag.add(beeper);
    }
}
```

This method illustrates the use of several of the methods available from the GreenfootWorld and GreenfootObject classes. The rest of the robot methods are rather simple and will not be shown in detail. The full set of methods implemented in the Robot class is:

- `void move()`: Move the robot one cell forward in its current direction.
- `void setDirection(int direction)`: Set the direction of the robot to one of the static values EAST, WEST, NORTH, or SOUTH.
- `void turnLeft()`: Turn the robot 90 degrees counterclockwise.
- `boolean canMove()`: Determine whether the robot can move forward (it cannot move if there is a wall in front of the robot).
- `void putBeeper()`: Put down a beeper from the robot's beeperBag if there are any.
- `void pickBeeper()`: Pick up a beeper.

The Robot class so far only contains methods for explicit invocation and is not very difficult to develop. To create behaviour that lets this robot act as part of a continuous simulation, the `act()` method can be implemented. All objects in the world are part of the simulation loop, and the act method of these objects will be called in each time step of the simulation. The default (inherited) act method is empty.

An example of such a robot could be one that moves forward and collects all beepers on the way. Such a harvester robot can be created, for example, as a subclass of Robot. The full implementation reads:

```
public class Harvester extends Robot
{
    public Harvester()
    {
    }

    public void act()
    {
        pickBeeper();
        move();
    }
}
```

Several robots could be created in this way that solve various tasks.

When using this scenario, students would typically be given an initial world, which contains an instance of a robot and possibly walls and beepers. This world is created by the constructor of the RobotWorld class. For instance, to add a beeper in the top left cell as part of the world creation, the following lines should be added to the world constructor:

```
Beeper beeper = new Beeper();
beeper.setLocation(0,0);
addObject(beeper);
```

6.1.2 Student Activities

Students can perform the same activities as in other Karel implementations with better support, and they can perform additional activities that enhance the students understanding of the dynamics of the program.

The traditional activities – executing and observing a robot; modifying a robot; creating and running new robots – are better supported since the editor, compiler and runtime display are all in one place and tightly integrated. There is no need to deal directly with file system structures, command line interfaces or class paths.

Additional activities possible in greenfoot include interactive placement of one or more robots (or other objects), interactive execution of any public method of the robot to test parts of its behaviour, inspection of the robot's fields, easy creation of test worlds by placing rows of walls interactively and single stepping in the debugger while observing source code execution and robot behaviour at the same time.

Observing behaviour. When students use greenfoot for the first time, they will typically interact with a scenario provided by the teacher. Students simply open the project in order to get started. On opening the project, the environment will automatically instantiate a new RobotWorld and show it. This initial world can include instances of objects, so the student can start experimenting and observing.

If the RobotWorld has, for instance, a harvester robot and a number of beepers, the student could start by pressing the *Run* button to observe how the robot is able to move and pick up beepers. (The *Run* button causes the simulation to start – that is, the `act` methods of all objects are repeatedly executed.)

Pressing the *Act* button calls each object's `act` method exactly once. It can be used to single step through the behaviour for a more fine-grained observation.
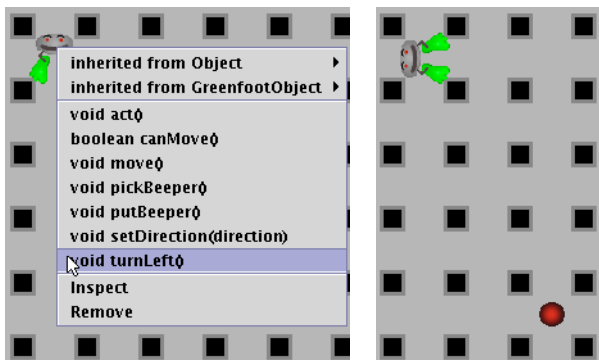

Figure 5: Interactive method invocation (left) with immediate visual feedback (right)

Direct interaction with individual objects. Another possibility is to directly interact with the objects in the world. For instance, it is possible to invoke methods on the objects and immediately observe the resulting change in state. Figure 5 shows the selection of the `turnLeft()` method from the popup menu on a Robot object (left) and robot after the method has been invoked (right).

Instantiating objects. Instantiation is directly supported by the greenfoot user interface. To instantiate a robot, students select a constructor from the class's popup menu (Figure 6, left). The mouse cursor then shows the image of the new object, which can be placed in the world (Figure 6, right).
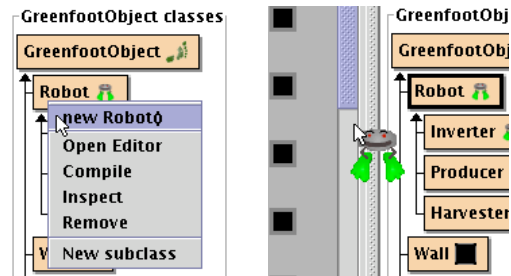

Figure 6: Interactive instantiation of objects

Students can easily instantiate several objects, and through experimentation and interaction observe common behaviour and differences in state. Object identity and the relation between classes and objects are emphasised.

Inspection. The visualisation of objects in the world does not show the complete state of the object. For example, it may not be possible to see how many beepers the robot carries. Objects may be explicitly inspected (by selecting the *Inspect* option from their popup menu) to display the complete set of values for the object's fields. Figure 7 shows an inspection of a Robot object.
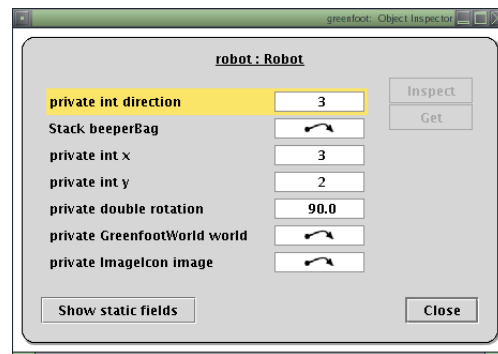

Figure 7: Inspection of a Robot object

Modifying behaviour. So far we have discussed how students can experiment with and observe objects without looking at the source code. The next step may be to modify some of the teacher supplied classes in order to see how methods are defined programmatically. For this the integrated editor can be invoked by double clicking a class.

Editing, compiling and error reporting are integrated in the environment, enabling a quick turn-around between code editing and execution. Students can also easily create new subclasses of existing classes.
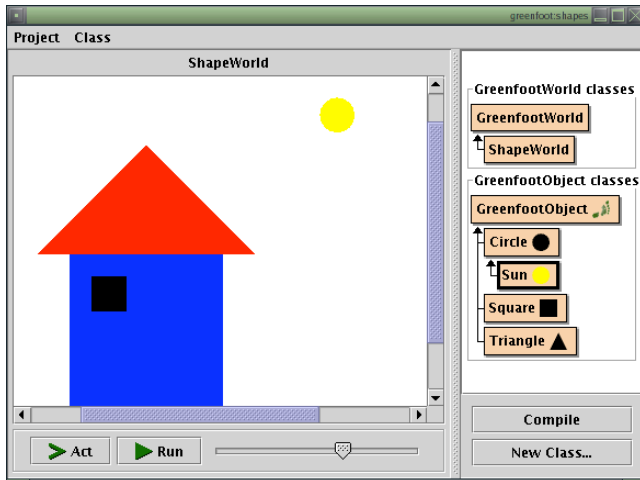
Figure 8: The 'shapes' example in greenfoot

6.2   The Shapes Scenario

The shapes example illustrates a different style of scenario to the grid based robot worlds. It is based on the shapes example for BlueJ, described in [1].

In the shapes scenario we have three types of shapes: circles, rectangles and triangles. These basic objects can be used to create various drawings by instantiating objects and changing their size, colour, rotation and location (Figure 8).

This example illustrates how greenfoot can be used for applications that are not obviously grid-based. This is achieved by creating a world with cells that span only a single pixel on screen, and simulation objects with icons that span multiple cells.

Students can interact directly with the shapes in the drawing to change their state. Shapes can then be programmed to include animated behaviour. The sun in this project, for instance, has a `sunset()` method that causes it to slowly move down.

6.3   The Turtle Graphics Scenario

One of the classic examples of a teaching world is the Turtle Graphics scenario. A greenfoot version of this is shown in Figure 9.

As in the shapes scenario the turtles do not seem to be in an obvious grid because the resolution of the cells is a single pixel.

The turtles can paint on the world through a standard interface supplied to greenfoot objects by the world class. Figure 9 shows three turtles painting various shapes in different colours.

It is easily possible to alternate between programmed behaviour (the execution of the `act()` method through the simulation framework) and direct interaction with the turtles. The turtle drawing the circle, for instance, has been stopped several times to manually raise and lower the pen and change its colours. Turtles could also be picked up and dropped elsewhere to continue their actions at another position on screen.
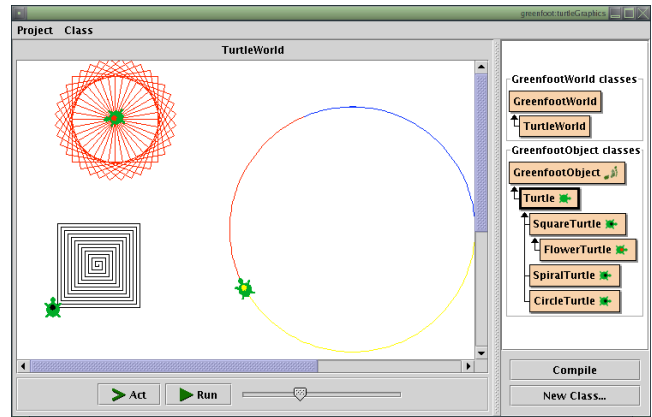

Figure 9: Turtle Graphics in greenfoot

6.4   Other scenarios

A wide range of other applications can be fitted into the greenfoot framework. While best suited to applications that produce two-dimensional graphic output, other uses are not excluded. Since drawing capabilities on the world include the drawing of text, some objects could display a behaviour that displays textual information on the screen. While this is not the main goal for greenfoot, it extends its capabilities.

A scenario like the Marine Biology Case Study can easily be implemented. This example is structurally similar to Karel the Robot, and does not require further detailed discussion here. One difference in use patterns is that there are often many fish involved in such a simulation, whereas Karel often only uses a few robots. To add a larger number of fish, a constructor or a method of the world can be used.

The graphical simulation output generated by greenfoot does not have to be a birds-eye view of a two dimensional area. A lift simulation is an example where the output might be a schematic animated drawing of a building with lifts and people.

It is easy to imagine many other simulation scenarios that are straight forward to implement. These include emergency evacuation of buildings, traffic simulations, supermarket checkout queues, predator/prey simulations and many more. Greenfoot may even be used to provide an easy-to-use output mechanism to more advanced exercises such as, for example, the dining philosophers problem.

7.   DISCUSSION AND CONCLUSION

The examples discussed attempted to illustrate the functionality and flexibility of greenfoot. The greenfoot system provides enhanced support for two user groups: teachers and students.

7.1   Support for student activities

Students are presented with richly visual scenarios that have proven popular in other systems, combined with interaction and easy experimentation tools that are intended to increase their engagement and understanding.

As in Karel-like systems, state changes and object behaviour are immediately observable through visual changes of greenfoot objects on screen.

Execution of object methods in 'simulation mode' is supported, which lets students sit back and observe object behaviour over a larger number of execution steps (after having implemented this behaviour), including interaction of objects in the greenfoot world and other emergent behaviour.

In addition to this, direct interaction with individual methods of individual objects is also supported, which lets student perform ad-hoc experiments with object behaviour before coding the behaviour in Java source code.

This functionality is available in addition to more standard tools for an educational IDE: an easy to use integrated source code editor, integrated compilation and a source level debugger.

7.2   Support for teachers
In section 3, we have discussed the tension between flexibility and framework support. Different approaches are at different ends of this scale.

On the one hand, teachers could write the complete framework themselves, thus achieving maximum flexibility in creating their preferred scenarios. In practice, this is impractical, because the workload of writing and maintaining such a framework is too high for the average teacher.

On the other hand, a ready-made framework could be used (such as Karel, Turtle Graphics or the Marine Biology Case Study). This avoids a lot of the work for the teacher, but has the disadvantage that the scenario is not flexible. If the framework gives you robots and beepers, then that's what students deal with: robots and beepers. Whether they like it or not.

The second is the situation commonly found today: micro world frameworks or libraries are written and maintained by a single person or a small group, including the design of scenarios. Exercises can then be created and shared between teachers. As a result of this, only a very small number of these micro world scenarios are available today.

Figure 10: Roles of people involved in creation and use of micro worlds (greenfoot vs. traditional)

Greenfoot decouples the framework creation from the scenario writing (Figure 10). Once the greenfoot framework is available, the difficulty and time requirements of writing a scenario become manageable for a much larger number of teachers. Projects such as robot worlds or lift simulations are easy to create, since all execution logic is provided by the environment. In addition, well tested and useful tools such as an editor, integrated compilation, a debugger and documentation generation are provided in a simple and easy to use interface.

Not every teacher needs to write scenarios (these can still be shared), but we expect that many teachers can, and will, create their own scenarios for greenfoot.

Thus, greenfoot is not a micro world, but a meta framework for micro worlds.

This decoupling of framework functionality from the user level scenario allows us to provide good tool support for the teaching of object orientation while allowing flexibility for individual adaptation at the same time.

7.3   New possibilities
Apart from making current curricula more engaging or more interactive, we believe that the use of greenfoot can open new possibilities for course design that are not practical today.

One of those possibilities is a direct side effect of the decoupling of the scenarios from the framework: using multiple worlds in a single course.

Currently, each micro world carries a significant overhead in its use. Each has different technical requirements, different installation procedures, different user interfaces that need to be learned, and different interaction models. As a result, there is usually not enough time in a single course to use more than one world scenario.

Since greenfoot has a consistent interface and interaction model across different world scenarios, the overhead of learning to use greenfoot exists only once, and using additional worlds is almost 'free'. This opens up the possibility to use a sequence of simulation scenarios (maybe of increasing complexity or from different application areas) in the same course. A course could start with turtles, move to Karel-like robots, and end by implementing a lift simulation for a high rise building.

7.4   Status
An implementation of a greenfoot prototype has been completed, and experimentation with this prototype with the goal of functional refinement is currently underway. A beta release of greenfoot will be available for free download shortly from *www.greenfoot.org*. A complete system, also to be distributed freely, is expected later in the year.

8.   REFERENCES

[1] Barnes, D. J. & Kölling, M., Objects First With Java – A Practical Introduction Using BlueJ, ISBN 0-13-044929-6, Pearson Education, 2003.

[2] Becker, B. W., Teaching CS1 with Karel the Robot in Java. In Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education. Vol. 33. (Charlotte, North Carolina, 2001)

[3] Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. Karel++ – A Gentle Introduction to the Art of Object-Oriented Programming. John Wiley & Sons, 1997
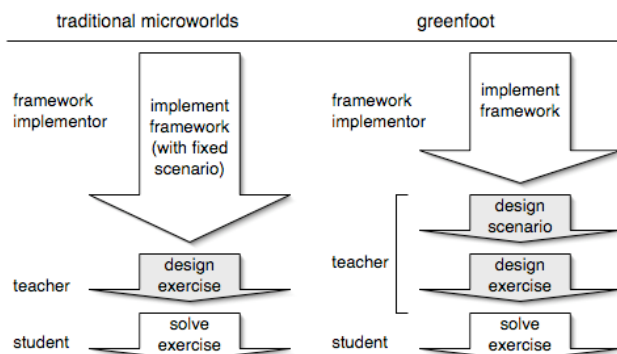
[4] Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. Karel J. Robot – A Gentle Introduction to the Art of Object-Oriented Programming in Java. Unpublished manuscript, available [18 March 2004] from: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

[5] Andrianoff, S. K. & Levine D. B., Role Playing in an Object-Oriented World, Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, Cincinnati, Kentucky, 121-125, 2002.

[6] Buck, D., Stucki, D.J., JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education. Vol. 33. (Charlotte, North Carolina, 2001)

[7] Caspersen, M. E., Christensen, H. B., Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS1, Proceedings of the Fourth Australasian Conference on Computing Education, 34-40, 2000

[8] Cooper S., Dann, W. & Pausch R. Teaching Objects-First In Introductory Computer Science. In Proceedings of the 34th SIGCSE symposium (Reno, Nevada, February 2003).

[9] Kolb, D. A., Experiential Learning. Experiences as the source of Learning and Development. Prentice-Hall, ISBN 0-13-295261, 1983.

[10] Kölling, M., Quig, B., Patterson, A. & Rosenberg, J., The BlueJ system and its pedagogy. In Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, 13 (4), December 2003

[11] Lukas, G., Uses of the LOGO programming language in undergraduate instruction. In Proc. of the ACM annual conference, Volume 2. (Boston, Massachusetts, 1972)

[12] Pattis, R., Roberts, J., & Stehlik, M. Karel the robot: a gentle introduction to the art of programming, 2nd Edition. John Wiley & Sons, 1994.

[13] Sanders, D., Dorn, B. Introduction to OO: Jeroo: a tool for introducing object-oriented programming. In 34th SIGCSE Proceedings (Reno, Nevada, February 2003)

[14] The College Board (Advanced Placement Program), Marine Biology Case Study. Available [September 10, 2003] from: http://www.collegeboard.com/