VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

A Web-Based Programming Learning Environment

Anna Maria Luxton

Supervisors: Robert Biddle and James Noble

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Learning to program is hard. Education in Computer Science has suffered a long history of low levels of retention particularly amongst females, as well as generally high failure rates. This report presents JavanOwl, a web-based programming tool we have built in order to address these issues. We propose that through the use of JavanOwl, novice programmers will be able to gain a sense of familiarity and confidence with basic programming during the early stages of their education in Computer Science. Evidence in the form of actual experiences and usability testing shows us that JavanOwl is successful in achieving its aims.

Acknowledgments

Firstly, I would like to thank my supervisors, Robert Biddle and James Noble, for constantly providing helpful and constructive criticism. Their joint sense of humour was both relaxing and encouraging. Secondly, thanks to all the other 'hermits' of Memphis for *always being there* in every sense of the phrase. In particular, I wish to acknowledge Jeromé Dolman, Donald Gordon and Francis Saul for their incredible amount of knowledge and kindness. I am eternally thankful for your contributions to this research project. Last but not least, I wish to thank my four kind evaluators for having been very helpful and patient in completing their evaluations of JavanOwl.

Contents

1 Introduction **1**

2 Background **3**

2.1 Education — The Current Situation . . . 3
2.2 Prior Familiarity . . . 3
2.3 Online Learning Approach . . . 4
2.4 Visualisations Approach . . . 4
2.5 Related Work . . . 5
2.5.1 BlueJ . . . 5
2.5.2 LearningWorks . . . 6
2.5.3 Jeroo . . . 7
2.5.4 Jeliot . . . 7
2.5.5 `www.publicstaticvoidmain.com` . . . 8
2.5.6 ELP . . . 8
2.6 Summary . . . 9

3 Design **10**

3.1 Web Technology . . . 10
3.1.1 Java Applets . . . 11
3.1.2 JavaScript . . . 12
3.1.3 JSP and JavaBeans . . . 12
3.2 Visualisation Technology . . . 13
3.2.1 Event-Driven Approach . . . 14
3.2.2 Aspect-Oriented Programming Approach . . . 14
3.2.3 AspectJ . . . 15
3.2.4 Image Technology . . . 16
3.3 Conclusion . . . 16

4 JavanOwl **17**

4.1 Goals . . . 17
4.2 JavanOwl . . . 17
4.3 JavanOwl Usage: A Demonstrative Scenario . . . 19
4.4 Why Java? . . . 20
4.5 Educational Practices Embedded in JavanOwl . . . 20
4.6 A Breakdown of The JavanOwl Application . . . 20
4.6.1 Before entering the system . . . 21
4.6.2 Writing and Saving a Program . . . 22
4.6.3 Running a Program . . . 24
4.6.4 Errors in user code . . . 25
4.6.5 Writing, Saving and Using a Class . . . 26

4.6.6 Studying Data Structures . . . 27
4.6.7 Managing Your Files . . . 30
4.6.8 User Forum . . . 30
4.7 Program Code Visualisations . . . 30
4.8 JavanOwl Library and API . . . 32
4.8.1 Classes . . . 32
4.8.2 JavaDocs and Related Documentation . . . 32

5 Implementation **33**
5.1 System Architecture . . . 33
5.2 Compile and Run Technology . . . 33
5.2.1 Security . . . 34
5.2.2 Code Samples . . . 35
5.3 Server Side Storage . . . 41
5.4 AOP and AspectJ for Program Code Visualisation . . . 41
5.4.1 GIF Generation . . . 42

6 Discussion **45**
6.1 Web advantages . . . 45
6.2 JSP advantages . . . 46
6.3 AspectJ advantages . . . 46
6.4 Visualisation using AspectJ . . . 46
6.5 JavanOwl visualisations . . . 47
6.6 Scalability . . . 47

7 Experience and Usability Studies **49**
7.1 Science Experience . . . 49
7.2 Science Experience Results . . . 50
7.2.1 Observations . . . 50
7.2.2 Evaluation Survey Results . . . 51
7.3 Heuristic Evaluation . . . 52
7.3.1 Experiment . . . 54
7.3.2 Results and Evaluation Summary . . . 56

8 Conclusions **59**
8.1 Contributions . . . 59
8.2 Comparison With Related Work . . . 60
8.3 Future Work . . . 60
8.3.1 JavanOwl API . . . 60
8.3.2 Traces and Visualisation Library . . . 61
8.3.3 Support for Visualistaions of User-Written Classes . . . 61
8.3.4 Interactivity in User Programs . . . 61

A JavanOwl++ **62**
A.1 Overview of Functionality . . . 62
A.1.1 Object Creation . . . 62
A.1.2 Object Manipulation . . . 62
A.2 How JavanOwl++ Uses JavanOwl Technology . . . 64

Figures

2.1 BlueJ - The Interactive Java Environment [1]. 6
2.2 LearningWorks - An example LearningBook page [2]. 7
2.3 Jeroo - The User Interface [3]. 8
2.4 ELP - Example ELP Exercise [4]. 9

4.1 JavanOwl in Use 18
4.2 JavanOwl Homepage 21
4.3 User enters code into the web form and saves it with a filename. 22
4.4 Example of code that uses library objects, written in JavanOwl 23
4.5 Example of code that outputs text, written in JavanOwl 23
4.6 JavanOwl system after having run a program that outputs program code visualisations 24
4.7 JavanOwl system after having run a textual program 25
4.8 Running user code that contains a syntactic error 26
4.9 Example class written in JavanOwl 27
4.10 JavanOwl system after having run a data structure program 28
4.11 Example of code that uses the Stack class, written in JavanOwl 29
4.12 User attempting to delete one of their programs. 30
4.13 JavanOwl animation of a user-written program that uses the Stack class. . . . 31

5.1 JavanOwl system architecture. 33
5.2 Transformation of user code that uses Person and House classes into JSP . . . 36
5.3 Transformation of user code that uses Stack class into JSP 37
5.4 Transformation of user code that outputs text into JSP 38
5.5 Transformation of user code that uses a user-defined class into JSP 40
5.6 An example pointcut defined for JavanOwl, written in AspectJ 41
5.7 An advice defined on the changePersonPC pointcut 42
5.8 Example of code that uses library objects, written in JavanOwl 43
5.9 JavanOwl animation of a user program that uses the JavanOwl API 44

7.1 General computer Usage 51
7.2 Writing code using JavanOwl 52
7.3 Jakob Nielsen's Ten Usability Heuristics 53
7.4 Severity Ratings 54
7.5 Scenario used in Heuristic Evaluation of JavanOwl 55
7.6 Heuristic Evaluation of JavanOwl Table of Results 58

A.1 Object Creation in JavanOwl++ 63
A.2 Object Manipulation in JavanOwl++ 65

Chapter 1

Introduction

A large proportion of students who enter introductory Computer Science courses have never had any experience in programming. For a student entering a CS1 course without any background in Computer Science, the course material can be very hard and overwhelming. This has been identified as one of the main causes of the very low level of retention and an equally high level of failure in CS1 courses worldwide.

The primary goal of this project is to support the early stages of learning to program, giving beginners the ability to experience a sense of familiarity and satisfaction with programming before or early on during their course. Tools specifically formulated to help students in the early stages of learning to program do exist, but they do not always target the core problems and are often not easily accessible. We aim to create a ubiquitous and simple programming environment that emphasizes fundamental programming principles. This will be accomplished by means of an attractive online service that would allow students anywhere to practice programming, facilitating the somewhat arduous process of learning to program.

A secondary goal of this project is to use program code visualisations for educational purposes. Visualisations have been used to examine the complex inner workings of program code since the birth of computers. Therefrom, both novice programmers and experienced software developers have employed different forms of visualisations of program code to better acquaint themselves with the intricacies of the code. One difficulty in this area is the actual collection of the information that is to be visually displayed. Often this information is spread across different classes and objects involved in the code. This type of problem is more commonly known as a 'cross-cutting concern'. Aspect-oriented programming is a paradigm which has been developed to help monitor information affected by cross-cutting concerns. The use of aspect-oriented programming as a program monitoring technique to aid in the creation of program code visualisations in this project has therefore been examined.

This report presents JavanOwl, a ubiquitous web-based programming environment designed to support novice programmers to learn how to program. JavanOwl makes use of visualisations to teach Java to novice programmers. Essentially, JavanOwl is a system that provides the type of service that "Hotmail" provides for email, but for Java programs. The intended audience for JavanOwl are learners before CS1, or learners having difficulty with CS1: people who will benefit from support to learn the basics of programming. The main technologies we explored in order to create JavanOwl are JavaServer Pages (JSP) [5], JavaBeans, JDBC, an SQL database, Aspect-Oriented Programming (AOP) [6] and HTML, all used in conjunction with a JavaServer Pages "Tomcat" web server.

The structure of this report is as follows: in the next chapter, a review of the background to this work is presented. Chapter 3 explores the different technologies currently available to support the key design elements of a new educational programming tool. This is followed by Chapter 4, which presents and explores JavanOwl; the system created as a result of the research conducted. Chapter 5 gives implementation details of how JavanOwl was built, while Chapter 6 evaluates various aspects of the technologies used in building the system. Chapter 7 gives a descant of some actual experience in using the system with learners, together with the results of usability testing. Finally, Chapter 8 provides a summary, presents the contributions to knowledge that this project has made and discusses the opportunities for future work on JavanOwl. Included in Appendix A is a description of JavanOwl++, an educational programming tool that was built using JavanOwl technology but serves a different purpose.

Chapter 2

Background

2.1 Education — The Current Situation

For a beginner, learning to program is hard. Because of this, the effectiveness of education in Computer Science has been under careful observation for many years. International studies of programming performance have produced compounding evidence that students' level of programming skills are not commensurate with their instructors' expectations [7], that overall retention rates are low [8], and that overall failure rates are high compared to other disciplines (as high as 30%) [9]. The reasons for these alarming facts are various: there are many difficulties encountered by beginners in Computer Science — for example, the syntax of a programming language, the inner workings of the computer behind the scenes [10], and the mathematics behind these concepts, all framed by the abstract concepts of the programming paradigm, such as object-orientation (OO) [11, 12]. This high myriad of complex information that students in introductory Computer Science courses are expected to absorb is exceedingly high. Within the first few weeks of a CS1 course, many students feel intimidated and disatisfied with their progress, and subsequently quit the course.

2.2 Prior Familiarity

Typically, students in introductory Computer Science courses can be classified into 3 different categories: novice programmers, experienced programmers and average programmers [13]. Naturally, this leads courses to be targeted at the middle ground — the 'average' group, which therefore disadvantages the disproportionately large group of novice programmers (by moving too fast) as well as the relatively small group of experienced programmers (by moving too slow). For those students who unfortunately find themselves in the novice group, having no prior experience and little confidence with computers, these courses can be quite overwhelming.

Research has shown that the most useful predictor of success in introductory Computer Science courses is prior familiarity with programming concepts, and that a high level of success is difficult to achieve without this prior knowledge [14]. Unfortunately, many high schools do not offer Computer Science as a subject. According to Rodger and Walker [15], most high school girls do not even know what Computer Science is. This lack of prior familiarity appears to affect women in particular, as they tend to have less exposure and less confidence with computers than men do upon entering Computer Science courses. This is supported by studies conducted at Victoria University that also showed that women withdraw or fail at higher rates than men, and are probably more anxious, less confident and have a poorer attitude towards computers in comparison to men as a result of genderstereotyping that has occurred prior to their arrival at University [16]. For social reasons, men appear to engage in more computer related activities with their friends, play computer games and explore the Internet, all of which help them gain a higher level of confidence with computers before their arrival at University. The existing relative imbalance of males and females in Computer Science reciprocates this fact [17].

Overall, this means that a large proportion of students entering introductory Computer Science courses at tertiary level, in particular women, fall into an educationally disadvantaged 'novice' group. Despite the efforts of educators in the field, students who lack this unspoken but perhaps essential 'pre-requisite' of some prior experience with programming are less likely to succeed in introductory Computer Science courses. Attempts to implement retention strategies, varying in methodology from large changes such as altering the textbook and course content, providing more help and reducing the male-orientation to small aesthetic changes such as lab atmosphere, have not addressed these earlier problems [18].

2.3 Online Learning Approach

Because of the large amount of information that students are expected to internalize during introductory Computer Science courses, the pace of these courses tends to be very fast. Concepts and skills are covered briefly simply in order to cover all the required material in time. Novice programmers may fall behind very quickly, constantly trying to catch up with topics they did not fully understand the first time. Some courses arrange tutorials to specifically address difficult course-related material. However, these are scheduled at specific times, so if a student misses a tutorial, they have missed the only 'outside of class' help they could have hoped for on a specific topic. The result is that students are not given sufficient or adequate support for these courses. Consequently, students don't learn as much nor as well as they potentially could, given more help. Students must be given the ability to access, outside of lecture and tutorial time, vital scaffolding and support regarding a particular course which allows them to practice and reinforce the concepts they are being taught in class. The very nature of online teaching objects allows for this type of support to be provided. In terms of novice programmers entering a Computer Science course, a supportive and well managed system of self-paced learning is crucial for their success in the course.

Another problem faced by education in Computer Science is the large need for resources. Students require access to textbooks, computers and software in order to experience writing code as well as compiling, debugging, and running programs. Even if a student did develop an interest in computer programming, they would still need access to these resources which may be expensive and unavailable to them. This puts a large demand on (especially primary or secondary) educational institutions' computing infrastructure. Online learning tools address these issues by helping reduce the need for so many expensive resources: textbooks are no longer as crucial to the learning process, fewer computers are needed, hardware, platform and software requirements are lessened and teacher-student time becomes less important. Providing access to an online system would help reduce the demand on institutions' computing infrastructure as students could work collaboratively in online communities with the necessary resources and materials provided automatically by the system.

2.4 Visualisations Approach

Visualisations are widely used as general teaching aids in many different disciplines to help students understand the concepts being taught [19]. It is common in Computer Science for visual depictions to be used to help communicate the complex inner workings of program code [12]. According to Blaine A. Price, Ronald M. Baecker and Ian S. Small [20], the usage of visual representations in understanding computer programs is by no means new. In 1947, Goldstein and von Neumann exposed the usefulness of flowcharts. In 1959, Haibt advanced this concept by developing a system that could draw flowcharts automatically from Fortran or assembly language programs. In 1963, Knuth expanded further on this by developing a system which integrated documentation with the source code and also automatically generated flowcharts. Since then, there have been major advancements in the use and applicability of software visualisations. Improvements in technology have enabled more diverse, more detailed, and more immediate visualisations of program code to be actualised and used with increasing ease.

Visualisations help students form the link between the theoretical concepts and more concrete ideas, as well as helping form the link between what the program output is and what actually happened behind the scenes. Writing a line of code and immediately seeing a change in the image that is generated provides an effect that helps motivate the link between syntactic commands and their semantic effect. Tanimoto describes these immediate visualisations of programs at runtime using the term 'Liveness' [21] of which there are four levels. These levels differ in the immediacy of the visual change in the visualised image of the program. The quicker the corresponding change is viewable, the higher the level of 'Liveness'. A system that incorporates level 4 'Liveness' visualisations is a system where any change in the program is reflected in real-time in the visualisation. Such immediacy in visibility of cause and effect is important in education as it highlights precisely what the effect of a certain change was, leaving little room for confusion. Obviously, in terms of education, the use of this type of visualisation is ideal — anything that helps ease the steepness of the learning curve and lessen the confusion felt by students, by providing an alternative way of looking at a complex concept, should be taken advantage of as much as possible.

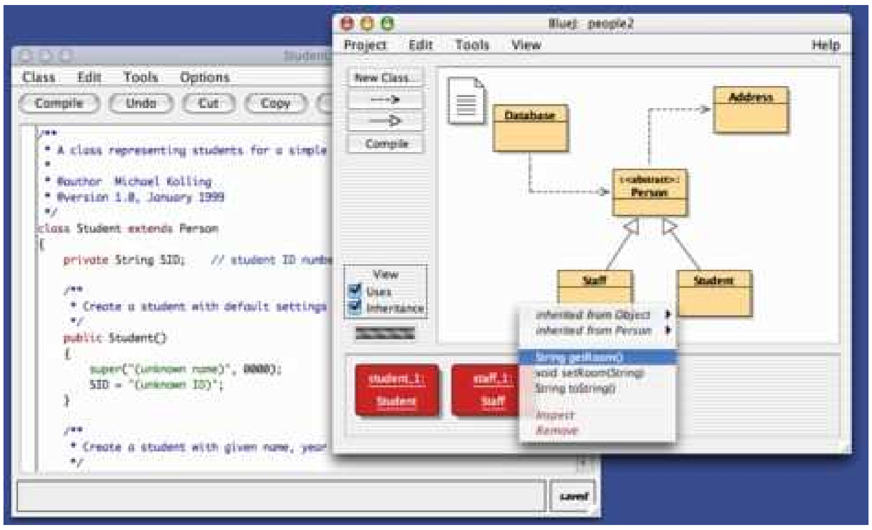
Unfortunately, there is a problem involved in providing comprehensive visualisations: it takes considerable time to individually hand draw and explain effective diagrams of code, especially if a hand version of level 4 'Liveness' is to be employed. There simply isn't enough time for the teachers and tutors of these introductory Computer Science courses to be drawing visualisations of all the subject matter they cover if they are to cover all the required material during the specified duration of the course.

2.5 Related Work

There are a number of existing educational programming tools that incorporate program visualisations and/or function online. Below is a description of six such tools, together with any major advantages and disadvantages that have been identified in the design and implementation of their frameworks.

2.5.1 BlueJ

BlueJ is an educational programming tool that is designed to teach object-orientation to beginners. The BlueJ environment was developed as part of a university research project about teaching object-orientation to beginners. It is being developed and maintained by a joint research group at Deakin University, Melbourne, Australia, the Mærsk Institute at the University of Southern Denmark, and the University of Kent in Canterbury, UK. It uses the Unified Modelling Language (UML) to visualise code. Code can either be written by hand or through the use of class diagrams as shown in Figure 2.1. Through an interactive interface, users can create objects and run methods on them, all by clicking on diagrams and through pop-up menus. It also includes a simple but effective debugger. Unfortunately, although free, BlueJ does not run over the Internet so users must download and install this system, as well as the Java 2 Platform, Standard Edition (J2SE). Users are therefore constrained to working on the same workstation and/or copying their files to disk and installing BlueJ on every machine they plan to work on.



```
/**
 * A class representing students for a simple
 *
 * @author  Michael Kolling
 * @version 1.0, January 1999
 */
class Student extends Person
{
    private String SID;    // student ID numbe

    /**
     * Create a student with default settings
     */
    public Student()
    {
        super("(unknown name)", 0000);
        SID = "(unknown ID)";
    }

    /**
     * Create a student with given name, year
     */
```

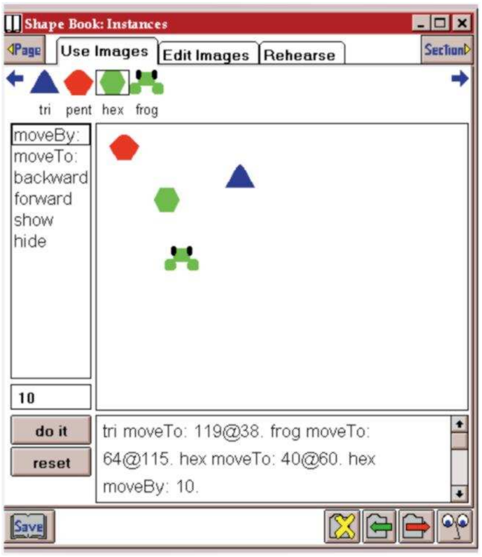
Figure 2.1: BlueJ - The Interactive Java Environment [1].

2.5.2 LearningWorks

LearningWorks is a learning environment that aims to teach ideas about computing as well as software systems architectures [2]. The LearningWorks project was started by Adele Goldberg and others in 1994 in response to an underlying problem with respect to building large maintainable software systems amongst corporate users of object technology. The basic conclusion was that these users needed to be targeted before reaching the corporate level; while students, they needed to have more experience with system building concepts and practices. This needed to be provided without increasing the need for teacher-student interaction [22].

LearningWorks was created as a solution to this problem. It is based on the Smalltalk programming language and is a highly organized self-contained teaching unit, structured as four frameworks: a LearningBook presentation and interaction framework that is meant to support the basic user model, a programming framework that enables users to write code using a library of reusable software components, a framework to allow the creation of LearningBooks and a communications framework to allow students to work collaboratively.

Users learn through the use of LearningBooks. Each LearningBook consists of pages that contain activities or applications that students interact with in order to learn more about a certain topic. Some pages may contain programming tools for students to explore existing object definitions or write new ones, such as the one shown in Figure 2.2. Since LearningBook sections and pages are self-contained components, they can be reused by other LearningBook authors. LearningWorks uses visualisation to show object interaction within the system, as well as more detailed views of the implementation of objects. It does not, however, run over the Internet. It must be downloaded and installed on the client machine, constraining the user to one computer unless they copy their files to other machines.



```
tri moveTo: 119@38. frog moveTo:
64@115. hex moveTo: 40@60. hex
moveBy: 10.
```

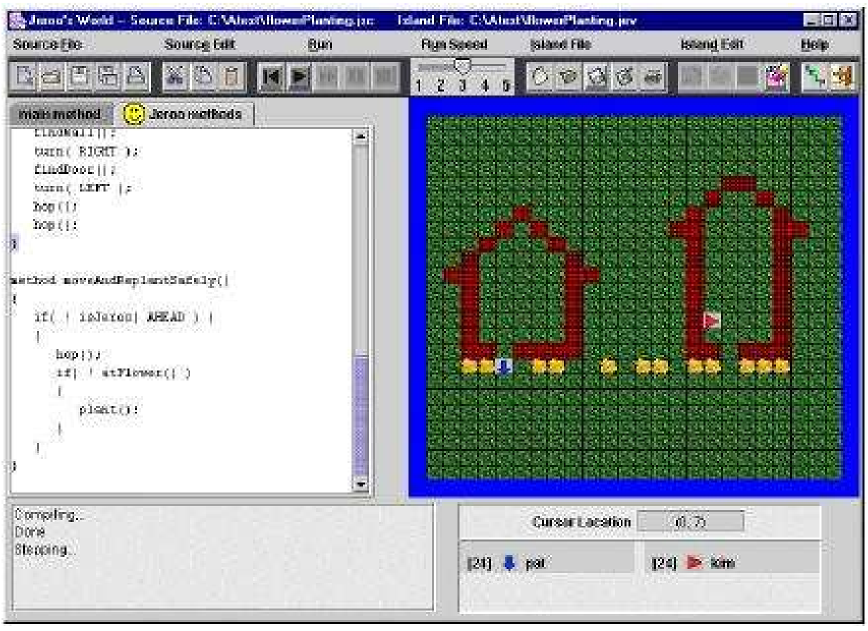
Figure 2.2: LearningWorks - An example LearningBook page [2].

2.5.3 Jeroo

Jeroo is an integrated development environment inspired by Karel the Robot [23] and its descendants. The Jeroo programming system has a simple syntax that provides a smooth transition to either Java or C++ [24]. Its design is straightforward and the system covers only a small scope of material aiming to help novice programmers maintain their confidence in the early stages of introductory Computer Science courses. Jeroo helps students learn about the instantiation and use of objects, how to design and write methods and about common control structures. The interface consists of a single-screen development environment that is structured so that students enter or edit code in a left hand panel and witness the changes in a visual animated representation of their 'virtual' world in the right hand panel of the screen (see Figure 2.3). One disadvantage of Jeroo is that it must be installed and run on the client machine. This imposes the requirement that the user must continually work on one workstation unless they copy their files to another computer that has also Jeroo installed on it.

2.5.4 Jeliot

One online system for teaching Java is Jeliot. It enables students to write source code and get back an animation of their code, generated automatically from the code itself. Jeliot is structured as a server application that works together with client Java applets so that it can be run over the Internet [25]. The Jeliot system allows users to relate the changes in the state of the program animation to the lines of code it is running. Being online is a major advantage, as it allows collaborative learning as well as easy access from many computers. A disadvantage of the design is that it relies on the browser's capabilities. Many applets pop-up during use of this system, which can confuse the user. Also, there can be problems with browser compatibility: a browser that is not Java-enabled will not be able to run the system properly.



```
   findWall();
   turn( RIGHT );
   findDoor();
   turn( LEFT );
   hop();
   hop();
}

method moveAndReplantSafely()
{
   if( ! isJeroo( AHEAD ) )
   {
      hop();
      if( ! atFlower() )
      {
         plant();
      }
   }
}
```

```
Compiling...
Done
Stepping...
```

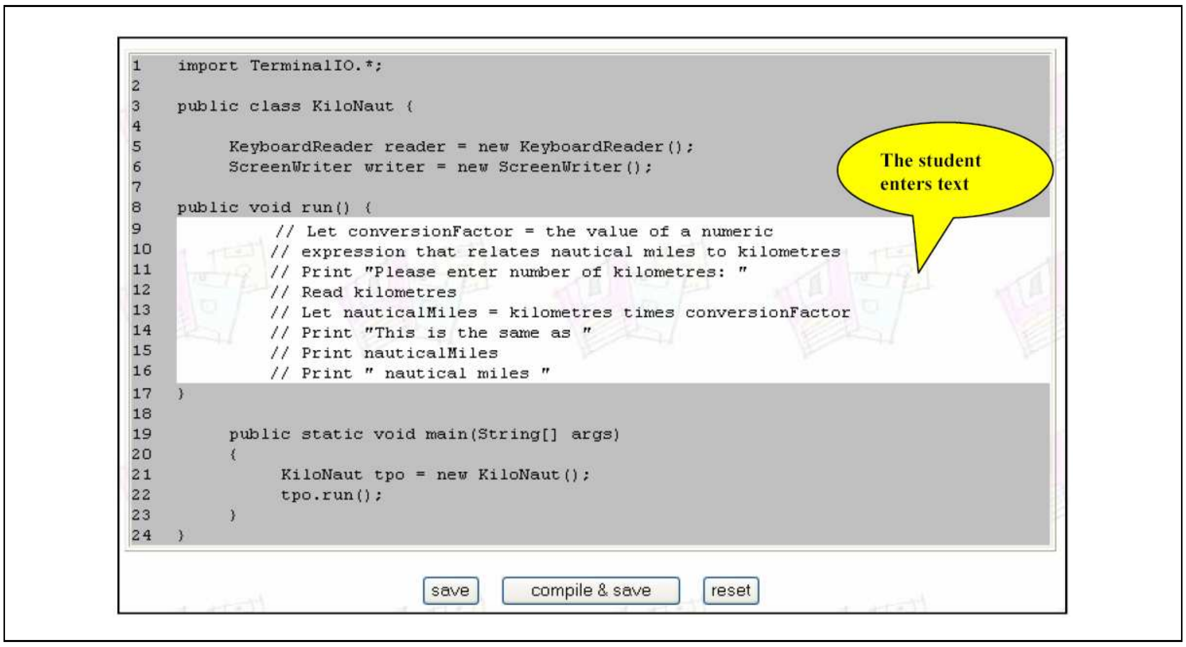
Figure 2.3: Jeroo - The User Interface [3].

2.5.5 `www.publicstaticvoidmain.com`

Another online Java system is `www.publicstaticvoidmain.com`. Through a teacher account group, students are allowed to set up their own accounts, log into the system and write and compile Java programs all online. `www.publicstaticvoidmain.com` is structured as a server application and works by means of client Java applets. One of the main advantages of this system is that the user does not need to have a Java Runtime system installed on their machine to compile and run any code. All code processing is done on the server side. One of the weaknesses of this system is that it is commercial and you must be part of a group that is lead by a teacher to get an account. Another disadvantage is that it does not incorporate any form of visualisation to help explain the code. As was noted with the other online systems explored, `www.publicstaticvoidmain.com` is not supported by all web browsers. This is due to its reliability on Java applets — in order to use `www.publicstaticvoidmain.com`, the web browser must be Java-enabled so as to be able to run the Java applets on the client side.

2.5.6 ELP

Lastly, a recently created online programming system for teaching Java is the Environment for Learning to Program (ELP) [4]. ELP is a highly structured system that provides an interactive web-based environment for teaching programming to first year Information Technology students, developed at Queensland University of Technology. It is strongly oriented to university course delivery and assessment. Learners enter code via a web site as shown in Figure 2.4, and the code is compiled on the server-side. However, the compiled code is then transmitted to the client machine as a JAR file, where the user must have a Java Runtime system to execute the code. This, and the restriction to one particular web browser, makes ELP less accessible than we would like.



```
import TerminalIO.*;

public class KiloNaut {

     KeyboardReader reader = new KeyboardReader();
     ScreenWriter writer = new ScreenWriter();

public void run() {
          // Let conversionFactor = the value of a numeric
          // expression that relates nautical miles to kilometres
          // Print "Please enter number of kilometres: "
          // Read kilometres
          // Let nauticalMiles = kilometres times conversionFactor
          // Print "This is the same as "
          // Print nauticalMiles
          // Print " nautical miles "
}

     public static void main(String[] args)
     {
          KiloNaut tpo = new KiloNaut();
          tpo.run();
     }
}
```

Figure 2.4: ELP - Example ELP Exercise [4].

2.6 Summary

This chapter presented an overview of some of the difficulties found in teaching Computer Science in CS1 courses. We discussed both the difficulties found within the context of teaching the complex material that Computer Science consists of, as well as problems present before the students even arrive at University. We examined online learning in terms of its applicability in education of Computer Science. We also discussed the use of visualisations to facilitate the understanding of some complicated OOP and general programming concepts, as well as the obstacles found in providing these visualisations in traditional teaching environments. All of this is then illustrated through a critical exposition of six existing learning objects that are designed to teach programming through the use of code visualisation and/or ubiquitous access provided by online functionality. The main advantages found in these systems due to critical design and implementation decisions will be further explored in the following chapter.

Chapter 3

Design

In Chapter 2, we discussed some of the approaches that can and have previously been employed in the design of educational programming tools. These approaches form the key design elements in planning the architecture of a new educational programming environment. This chapter will examine relevant approaches to these design elements and the technologies available to leverage them. Below are detailed these approaches; the key design elements in building a new educational programming tool:

- Web-based educational programming tools.
  Section 2.3 identified the benefits of using Internet applications as opposed to standalone applications as the basis for educational tools. In this chapter, we will explore the different web technologies available to support online learning tools, finding in particular those that provide the best advantage:disadvantage ratio whilst keeping the client side lightweight.

- Visualisations in educational programming tools.
  Section 2.4 explained the application and advantages of using visualisations in teaching. Here we will discuss the benefits and weaknesses of different visualisation technologies employed to support this approach. This requires a survey of different program monitoring techniques as well as a look at different image technologies that can be used to deploy effective images on the web.

3.1 Web Technology

Traditionally, the Internet is geared towards supporting static HTML that contains plain text and static images. More value can be added to the Internet 'experience' in various ways. One way is to create dynamic HTML that allows users to enter and retrieve relevant information. This is done through web programming using languages such as Java Applets, JavaScript, JavaServer Pages [5], or other scripting languages such as ASP, PHP or Flash animations. Use of such technologies lead to sophisticated and dynamic websites that can be much more interesting and interactive than standard static HTML.

Running a dynamic website does, however, involve a fair amount of information processing which can happen either on the client side and/or the server side. Processing information on the client side often creates problems with compatibility and technical requirements on the client machine. Since an educational programming tool, if it is to be interactive, may need to process a lot of information, then it may be convenient to process as much of this information as possible on the server side. This will enable users on any client computer, old or new, with as little processing power as is needed to support Internet connectivity and run a browser to be able to use the tool. Having a lightweight client side avoids many of the problems regarding browser compatibility since much of the information is processed on the server, bypassing the problems caused by different browsers processing the same information slightly differently.

What follows is a discussion of three of the main technologies that are commonly employed to add value to websites. The advantages and disadvantages of using each of these technologies are also considered.

3.1.1 Java Applets

Java applets are essentially Java programs, except they run through web browsers. Since Sun designed Java to be an operating system independent language, Java programs can be written and compiled on one machine and then be run on many other machines without needing to be altered. This fact makes programs written in Java conducive to being distributed over the Internet. In 1995, Sun released HotJava, a Java-enabled browser that allowed Java programs to be run over the web. Since then, most (but not all) other browser vendors have added Java support to their browsers so as to make them Java-enabled. What makes browsers Java-enabled is the fact that they include a Java Virtual Machine (JVM). Web programmers can write Java programs, compile them as they would normally, and then include the bytecode, as they would images, in their HTML pages using the `<APPLET>` tags. The bytecode is downloaded from the server side and executed on the client's machine, using the JVM in the browser. This is possible since Java applications are platform and machine independent, so the program can be written and compiled on the server machine and the resulting bytecode can be downloaded and executed on any other machine.

Java applets are very useful for adding a dynamic, interactive and multimedia edge to a web page. They are dynamic in the sense that they are active programs essentially 'running' on the client's machine, through their browser. By being dynamic, they can simulate changing parameters — such as motion of a pendulum. They are interactive because they are Java programs, running at real time on the client's machine, so they user can interact with it (if it has been programmed to be interactive). This allows programs that rely on the user changing parameters in order create some form of understanding to be written. Applets can also make extensive use of multimedia — graphics, animation and sound can all be included in an applet. All of this makes Java applets an good choice for adding value to websites.

Unfortunately, there are disadvantages of using Java applets. Their main weakness is that not all web browsers are Java-enabled. Some newer versions of browsers have Java support built in to them, but others require Java plug-ins to be downloaded and installed. Therefore, web-based systems that are built using Java applets are not completely platform and machine independent. This reduces the overall group of potential users. Also, newer versions of Java cause inconsistencies in Java applets. Applets written in an newer version of Java, such as Java 1.2, may not work in a browser that is designed to support Java 1.0. Changes in HTML Data Type Definitions (DTDs) also cause compatibility problems between browsers. Another disadvantage of embedding Java applets into websites is that although in general the size of applets is small, they do have to be completely downloaded onto the client's machine before they begin to run. This increases the amount of information that users will be required to download when visiting a page that includes Java applets.

Overall, Java applets can add a tremendous amount of value to websites. They add an ample amount of interactivity to websites, through dynamic simulations and different types of multimedia. Unfortunately, due to browser compatibility problems as well as added download size, Java applets can cause unwelcomed inconsistency to websites. Some of the common applications of Java applets, such as dynamic simulations, can be made equally as effectively through the use of animated GIFs, a much simpler technology that will not cause any browser compatibility problems.

3.1.2 JavaScript

When invented in 1995, the JavaScript language created by Netscape was better known as LiveScript. The name was later changed to JavaScript in a joint effort by Netscape and Sun, apparently due to the growing popularity of Java [26]. Although some of the syntax of JavaScript is quite similar to that of Java, there is little in common between the two languages.

Originally, JavaScript was included in the Netscape Navigator (NN) 2.0 browser via an interpreter that read and executed the JavaScript that was included in the HTML pages it rendered. Since then, the language has become more and more popular and is now supported by most popular browsers although different browsers implement JavaScript in slightly different ways. Originally, JavaScript was mainly used for client side form validation — verifying and checking user input (such as date format) in web forms. Although this validation can be done on the server side, the advantages of doing it on the client side is to avoid sending invalid data to the server which may need to be sent back with an error, thus reducing the amount of bandwidth used and server processing power. JavaScript is also used to implement many sophisticated dynamic web interfaces.

Although effects implemented in JavaScript are much faster to download than some other front-end technologies like Flash and Java applets, they still add to the amount of information (even if only by a small amount) that the user will have to download when they visit a website. Users do not need to download a plugin before they can view JavaScript, as they do with Flash for example. They are required, however, to have a browser that supports JavaScript installed on their machine. This causes browser compatibility problems. Most modern browsers do support JavaScript, but different browsers implement and thus process and display JavaScript slightly differently, so the results can be rather unexpected and inconsistent depending on which browser the client chooses to use. Code that works on Internet Explorer 4 might not work at all on Netscape 4, and differences may even be found between different versions of the same browser. Some older browsers do not support JavaScript at all. Browsers from different vendors allow scripting languages to access different HTML elements and features of the browser, which can cause inconsistencies from the client's perspective when any script is included in websites. Another problem is related to the different versions of JavaScript that have been released over the years. Although most JavaScript is more or less still the same, some subtle but noticeable differences can be found between different versions of JavaScript which can cause further inconsistencies.

Overall, the use of JavaScript can add significant value to web sites as it adds an elaborate level of dynamic interaction, but all the compatibility issues involved in doing so can cause serious inconsistency problems.

3.1.3 JSP and JavaBeans

JavaServer Pages (JSP) is a Java technology that is aimed at allowing web programmers to dynamically generate HTML. JSP files are stored on web servers, and are essentially HTML pages with Java code embedded in them between specially defined JSP tags. When a user makes a request to a JSP page on the server side, the corresponding JSP file is compiled into a Java file by a JSP web server such as "Tomcat". This Java file is better known as a *servlet*. Servlets are also kept on the server. Their function is to dynamically generate normal HTML content, which can then be displayed to the user as normal static HTML. The effect of this is to create dynamic and personalized webpages [27] depending on the user's actions, which can be displayed to the user as a normal HTML page. Since all the processing involved in creating these dynamic HTML pages is kept completely server side, it is invisible to the user requesting the HTML page.

JSP is simple to write — it is normal Java code written in between JSP tags. In order to keep a degree of separation between the Java code and the HTML in a web page, the Java code can be written into JavaBeans. JavaBeans are reusable components written in Java which form a natural extension to JSP [28]. Like JSP, they are also stored, compiled and run on the server side. The code they contain is normal Java code that can be used to store state as well as manipulate user inputted data. JavaBeans can also use other Java classes, making them very powerful. The advantage of using JavaBeans is that only small amounts of JSP need to be written into HTML pages. These snippets of JSP can then call larger and more complex reusable methods written in JavaBeans to perform the required manipulation of data. This separation of processing code and HTML makes dynamic websites easier and quicker to create.

Since JSP is stored and processed completely on the server side, any request that a user makes to a web page that contains JSP will be sent back to the server machine, processed and sent back to the client. This may be slower than the JavaScript way of processing the information directly on the client machine. Request speeds depend on the speed of the Internet connection and processing power of the server, as opposed to relying heavily on the client machine's processing power. This means that clients can run online systems built with JSP on any computer with a browser and Internet connectivity, regardless of their CPU speed. This server side processing also avoids browser compatibility problems. Since the only information being sent between the client and server machines are normal HTTP requests and responses, the user only needs to have a browser, without any special plug-ins, to display the requested page. Since JSP is based on Java, which is a platform independent language, the server can also be run on any machine, regardless of which operating system is being run. Other server side scripting languages such as ASP, on the other hand, locks the developer into running their server on specific platforms, such as WindowsNT/2000.

One disadvantage of using JSP is that it does restrict the graphic design of sites as it does not produce as visually gratifying results as using Flash or JavaScript would. However, it's ability to store state, connect to databases and manipulate data with defined reusable components in normal Java code, all on the server side and without causing browser compatibility problems, makes JSP a very powerful tool for web-programming.

3.2 Visualisation Technology

In education, it is important to be able to explain code in more than one way. Graphically visualising code is one attempt at doing this, producing traces of program code is another. But in order to collect information about a program so that it can then be used to better explain the code's behaviour, we need some way of capturing the required information at critical or interesting points during the execution of the program. This is commonly known as 'program monitoring'. The task of program monitoring has been done in various ways in the past. One approach commonly taken is known as the *event-driven approach* [29]. Another approach to this task is to use *Aspect-Oriented Programming*. The advantages and disadvantages of using these two different approaches are discussed below.

3.2.1 Event-Driven Approach

The crux of this approach is the concept of annotating the target program. What this means is that in order to capture information about the program, the programmer must first decide which events are of enough importance that they deserve to be monitored, and then alter or insert code before, after or into those events that will output statements containing the information being sought out. This output can take any form — it can be purely textual or visual or even a mixture, depending on the nature of the program being monitored and the purpose of the monitoring. The programmer simply needs to include code that will create the type of output they wish to display. Commonly, simple calls to `System.out.println(x)` are made, outputting some interesting information `x` in a textual manner.

An advantage of this technique of program monitoring is that it is simple and easy to collect and display any sort of information needed. A rather large disadvantage, however, is that it is an intrusive technique. It requires altering the target program. Not only may this be inconvenient, but it may also not be allowed in certain cases where the user is not permitted to change the target code. Directly modifying the target program may also obscure the semantic meaning of the program and may introduce bugs that did not exist beforehand. Also, this technique implies that there is no degree of separation between the output statements and the program code. So if the output format is modified, all the output statements must be changed everywhere in the target program. This can be a tedious task to perform if there are many events which have been marked as being worthwhile monitoring. One way to avoid this problem is to define visualisation 'events' outside of the program code and inserting 'event markers' in the target program, rather than inserting output statements directly. This way, when an event marker is reached, it makes a call to a method which runs some form of output code, outside of the target program's scope. This is similar to keeping a 'log' class with log entries made everytime an interesting event occurs. This technique introduces a degree of separation — if a modification is made to the format of the output, it only needs to be made in one place. This reduces the need to keep making alterations in the target program. But this method is still intrusive as the target program will still need to be modified to include the event markers.

3.2.2 Aspect-Oriented Programming Approach

AOP is a relatively new paradigm designed to address programming problems that cannot be cleanly encapsulated as separate components. Some systems are difficult to build using the traditional style of modularity [30]. Examples of this include debugging, error-handling or system logging across entire systems that utilize several classes. These types of tasks can be described as "cross-cutting concerns" [29] or *aspects* of a system. Aspects of systems such as these cut across one another as well as the final executable code, that is, other programming concerns. This type of problematic scenario has lead to the development of a new programming paradigm, appropriately named 'Aspect-Oriented Programming' (AOP). AOP determines that any programming concern that cuts across other programming concerns, (i.e. cannot be cleanly encapsulated into a separate component) should be declared as an *aspect*. Using AOP, programmers have the power to break down large problems not in an object-oriented manner, but rather using *aspectual decomposition*. Cross-cutting problems such as debugging or system logging which require large mechanisms for gathering information can be facilitated by using AOP.

In AOP, rather than writing classes, programmers write *aspects*. Aspects are written in an AOP language (such as AspectJ). Aspects define which classes to monitor as well as what information they should capture. In order to specify what information to capture, aspect developers must use *join points*. Join points tell the aspect to do 'something' (such as collect program information and manipulate it) at a specific point in the target program execution. The original AOP definition of join points is: "... those elements of the component language semantics that the aspect programs coordinate with" and "...clear, but perhaps implicit, elements of the component program semantics" [29]. Using join points, the developers catch the information they require about any classes in the target program, and manipulate it accordingly. This is how AOP addresses the problem of cross-cutting concerns.

Program code visualisation is a cross-cutting concern [31]. This is due to the nature of the information that is required to create the visualisations. Depending on the type of visualisation, information from various classes in the target program may need to be collected. Since AOP is a good way of collecting information about a running program, particularly the type of information we would normally be interested in visualising, it follows that it may be a good way of collecting information for program code visualisations. One main advantage of using AOP to collect information for program visualisation is it's ability to monitor programs without being intrusive. This means that unlike the event-driven approach described above, neither the target program nor the source of the classes being used need to be altered in any way — no extra code needs to be added to them at all. This is a big advantage as the program monitoring is done in the background, without any alterations needing to be made or visible to the user. Intrusiveness is a common problem amongst other methods of program monitoring [29].

3.2.3 AspectJ

AspectJ is an AOP extension to the Java programming language [6]. Using AspectJ, which is essentially Java code, 'aspects' are written. AspectJ can be used to monitor many different kinds of 'interesting' landmark events, including method calls, method executions, object instantiations, constructor executions, and field references. In AspectJ, these events are known as the 'join points of Java' [6]. AspectJ allows the programmer to monitor these join points quite simply with the use of *pointcuts*. The way in which these events occur and what sort of information the programmer expects to get back determine how they should be captured. This is done with the use of *designators*. There are many different designators. The *execution* designator is used when information about *when a particular method body executes* is needed, whereas the *call* designator is used when information about *when a method is called* is required. Method or constructor names, together with wildcards and logical operators, are used to tell these designators which methods to monitor. Once some pointcuts have been defined, the programmer must write *advices* for them as pointcuts don't actually do anything other than pick out join points. Advices are Java code that gets executed at the join points that have been picked out by a pointcut. It is in these advices that the programmer can write code to manipulate and format the captured information in whatever way they wish to output it.

One of the main uses of AspectJ is to collect, format and display trace information about a running program for debugging purposes. The reason for this is that by using aspects, the programmer can essentially 'tap' into the state of the running program and collect any data they think may be useful. Java code allows the programmer to manipulate and format this data into textual or graphical output using the Graphics2D Java API for example, depending on the nature of the debugging data being displayed or the target audience.

In order to trigger the pointcuts when calls are made to methods in the classes, all the files defining the classes, programs and aspects must be compiled together. This is done using the AspectJ compiler (AJC). AJC *weaves* the aspect code into the classes, so that the resulting bytecode contains the relevant aspect code. This way, when a program (which has also been compiled using AJC) that utilizes these classes is run, calls to methods in the classes cause not only the normal effect of the classes, but also the aspect code to be triggered.

3.2.4 Image Technology

The use of images in websites is widespread. They add colour, catch peoples attention and often convey a lot of information in a relatively small amount of space. Normally, images on websites are static. They are stored in a file somewhere on the server side and the user is shown the same file everytime they visit the website. Significant value can be added to websites if images are dynamically generated depending on user actions.

Images can be regenerated depending on user actions quite simply: when a user submits data, the server performs the appropriate processing using Java code together with a GIF generator to create the image 'on-the-fly'. Users can then be shown different and more appropriate images that reflect their actions. This is the core idea behind program code visualisations — the visual depictions are designed to highlight and reflect relevant changes in the state of the user's program code. Two main technologies available to do deliver dynamically created images are discussed below.

Write Generated Image to File

Once an image has been dynamically re-created using Java and a GIF generator, it can be written to a file on the server machine. The website can then refresh, displaying the new image. This method can potentially cause problems if the image file being overwritten is already open elsewhere. Some browsers insist on caching images and so this method can also fail if the browser simply redisplays old cached files.

Use a Java Servlet to Serve Image

Another way to deliver dynamically re-created images on the web is by using a Java servlet. The image is still dynamically generated depending on user actions using Java code and a GIF generator, but this time rather than being written to a file, it is written directly to the HTTP response using a servlet. This avoids having to create files on the server and can avoid browser caching problems as well.

3.3 Conclusion

This chapter has examined the different technologies available to support the approaches that address the key design elements of building a new educational programming tool. The technology we have found to best support lightweight client side online systems is JavaServer Pages. We found that aspect oriented programming through AspectJ would be the best technology to support program code visualisations. To deliver these visualisations, we chose to use a Java servlet to avoid browser caching problems. Together, using these technologies we have built a web-based programming environment that makes extensive use of program code visualisations. This system is described in detail in the following chapter.

Chapter 4

JavanOwl

4.1 Goals

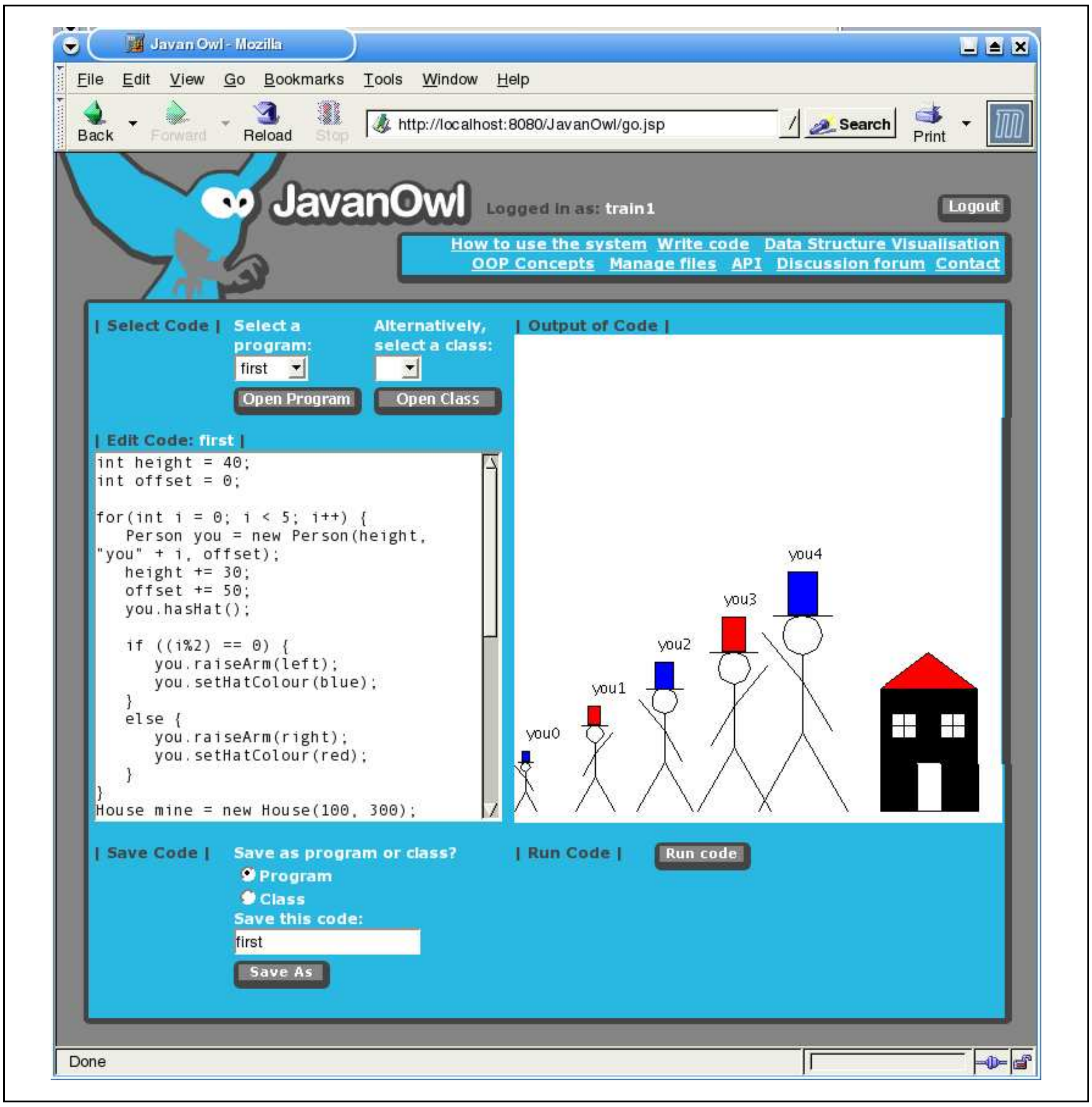
Many problems associated with education in Computer Science we have identified. Of these, the following have been identified as some solutions that may help the overall current situation:

- Provide an accessible opportunity for learners to familiarize themselves with programming before the start of a CS1 course.
- Provide a safe and clear environment in which CS1 students can practice and experience first-hand some of the concepts and programming techniques they are being taught at their course.
- Reduce the need for learners to buy or install new software, hardware or textbooks.
- Eventually add to the combined effort of reducing the large gap between the number of females and males in Computer Science.

4.2 JavanOwl

JavanOwl is a programming environment designed to help beginners learn how to program. It is web-based and provides a cheaper and ubiquitous alternative to traditional applications. JavanOwl works on all web browsers dating back to early versions, and does not require any applets or plugins. The JavanOwl system aims to be a collaborative learning environment, based on a cognitive and constructivist pedagogical strategy, where immediate feedback is provided, and learning from errors is encouraged. Learners will be able to use it from any computer, no matter how old and cheap, with minimal Internet connectivity. This reduces the need for resources such as expensive computers, software, textbooks and teacher time.

JavanOwl allows users to create, edit, save, and retrieve programs, and to run both programs written by users, and example programs provided in the system. Usage is simple and straightforward, with a split-screen web page showing both the code and the output. The user enters their code through a simple form and then runs it. If the code is error-free, the browser will be display the result in the output pane. If the code has errors, the output pane will display a simple specific error message. Users can then edit their code and re-run it. The system interaction is thus based on a principle of immediate feedback with visibility of cause and effect.



```
int height = 40;
int offset = 0;

for(int i = 0; i < 5; i++) {
   Person you = new Person(height,
"you" + i, offset);
   height += 30;
   offset += 50;
   you.hasHat();

   if ((i%2) == 0) {
      you.raiseArm(left);
      you.setHatColour(blue);
   }
   else {
      you.raiseArm(right);
      you.setHatColour(red);
   }
}
House mine = new House(100, 300);
```

Figure 4.1: JavanOwl in Use

Users can write simple stand-alone Java code entered in a single form. This is powerful enough to allow learners to work with library objects and their methods, and to use sequences of statements, iteration and selection, as shown generating the output in Figure 4.1. Users can also write programs that output textual output. The system is powerful enough to allow users to write their own classes and use them in programs they write. This gears the learning towards an OO-first approach. JavanOwl includes both textual and graphical output, which is supported by dynamic image generation over the web. The dynamic image is an important idea in our approach, and has been developed in our other web technology work [32]. Library objects with graphical output constitute our primary approach to run-time visualisation, as shown in the figures. This approach is both simple and motivational. Aspect Oriented Programming has been used to provide the run-time visualisations. This approach may be extended to provide more or less detailed visualisations of showing different views of the program code, such as traces.

4.3 JavanOwl Usage: A Demonstrative Scenario

If used as a teaching aid, JavanOwl will give lecturers and tutors of introductory Computer Science courses more time to focus on the more abstract side of Computer Science, allowing them to leave some of the more practical programming exercises for the students to practice independently. By providing a clear environment, this system can be used as a learning playground by anybody who wishes to practice and reinforce what they learn in class.

Alternatively, if introduced to students at earlier levels, such as secondary school, JavanOwl will provide an opportunity for learners to familiarize themselves with basic programming concepts before the start of a Computer Science course, giving them a higher chance of succeeding in their course. Picture the following scenario:

Student X attends a local public high school. She wishes to learn how to program in Java, but is told by her teacher that programming is not taught at their school. Student X is advised to:

- buy herself a relevant textbook, download or buy all the relevant software and install it on a PC either at home or at school, and begin working on it herself. Student X finds that textbooks are very expensive and that she cannot possibly afford one. The PC she has at home is not 'hers', so she cannot install software on it, and she is afraid of doing something wrong on the ones at school. She feels lost and confused. Without some teacher aid, she doesn't know where to begin the rather long and arduous task of learning how to program. Student X decides to wait until she can attend an introductory Computer Science course at a tertiary education institution.

  Eventually, Student X enrols in a CS1 course. Finally she is going to learn how to program. Within the first couple of weeks of the course, she is expected to learn a large myriad of information regarding object oriented programming, data structures, algorithms and other programming concepts. She has already been asked to write her own Java program as part of the first assignment. She is overwhelmed and very lost. Student X feels that no matter how hard she tries, everyone else in the course seems to already know more than her. At last, Student X decides that there is no point in continuing to try, she simply cannot absorb this much information. Her self-confidence with computers decreases drastically and she arrives at the inevitable conclusion that she is no good with computers. Student X decides that she was not meant to be a programmer and quits the course.

- try using JavanOwl. Student X accesses the JavanOwl application online from the school computer during her lunch break, begins by looking at some sample code in the Help section. Student X realises she is able to access JavanOwl from any computer, and so spends some time at Internet cafes playing with code. Soon, student X finds the courage to try writing some code of her own, experimenting with different programming constructs and techniques. She uses the visualisation tool to help her grasp how the code works and if she has any questions, she posts a message to the web-forum and waits for a reply from a tutor or a fellow student. Student X writes some fun programs using the JavanOwl library and passes her program's URL onto friends for them to try running. Student X has now successfully gained the experience with programming that will increase her chances of succeeding at an introductory Computer Science course.

This is a vision of how JavanOwl will help education in Computer Science.

4.4 Why Java?

JavanOwl introduces Java at a basic level, helping portray some of the important Object-Oriented paradigms that students must learn. The motivation behind choosing Java as the language to teach is threefold:

- Many universities around the world use Java as the first language they teach as part of their introductory Computer Science courses. JavanOwl would therefore directly benefit students entering these courses.
- Java is a real language, it exists in the real business world, so it will be useful for students of all ages, providing practical motivation.
- Java is used for web programming, so the web context of JavanOwl is based on a real and useful application delivery domain.

4.5 Educational Practices Embedded in JavanOwl

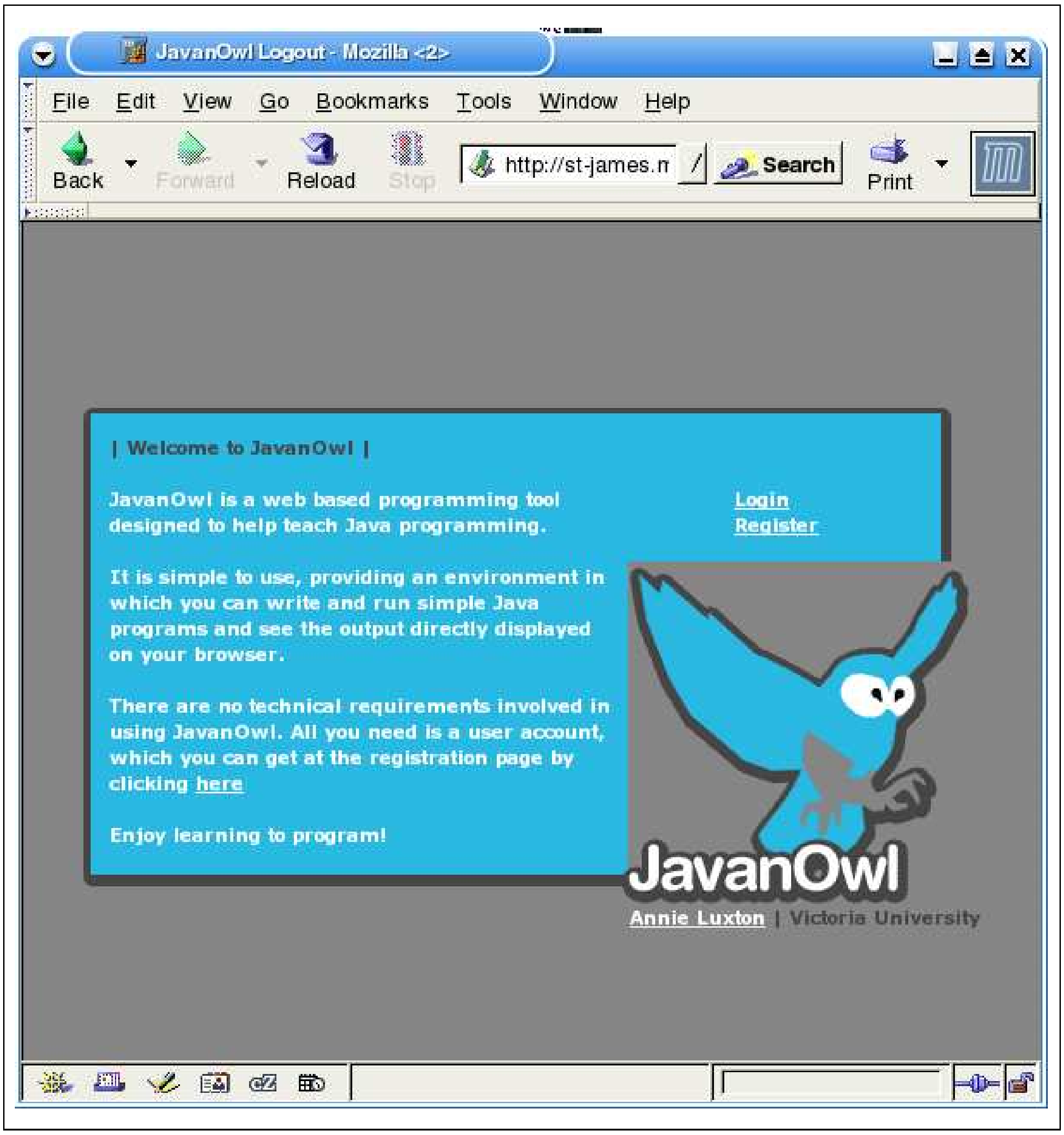
JavanOwl supports both scaffolding and collaborative learning, practices that have become increasingly more popular following the move towards a more constructivist, generative approach regarding education [33]. Scaffolding plays a very important role in the success of computer-based learning environments as they motivate the learner, reduce task complexity, provide structure and reduce learner frustration [34]. JavanOwl takes this approach by the dramatic simplification of the programming environment, immediate feedback on the same page, and the use of pedagogical library objects with graphical output. JavanOwl also includes its own library, with tutorials and documention using conventional web technology. The system also supports collaboration, and provides a web discussion forum built-in so as to provide users with a support system that will allow a collaborative community of peer support and file sharing to arise.

4.6 A Breakdown of The JavanOwl Application

JavanOwl is designed as a complete web-based system. This means that within the JavanOwl application, there are various different sections that allow users to perform different functions. This section gives an in depth description of each of the different actions that users can perform through JavanOwl.

4.6.1 Before entering the system

The home page is a simple title page for JavanOwl that briefly explains what it is for and where to find more information. From here, users can register to use the system or log into the system using their username and password. Figure 4.2 shows the JavanOwl homepage.


JavanOwl Logout - Mozilla <2>
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://st-james.n
Search
Print

| Welcome to JavanOwl |

JavanOwl is a web based programming tool designed to help teach Java programming.

Login
Register

It is simple to use, providing an environment in which you can write and run simple Java programs and see the output directly displayed on your browser.

There are no technical requirements involved in using JavanOwl. All you need is a user account, which you can get at the registration page by clicking here

Enjoy learning to program!

JavanOwl

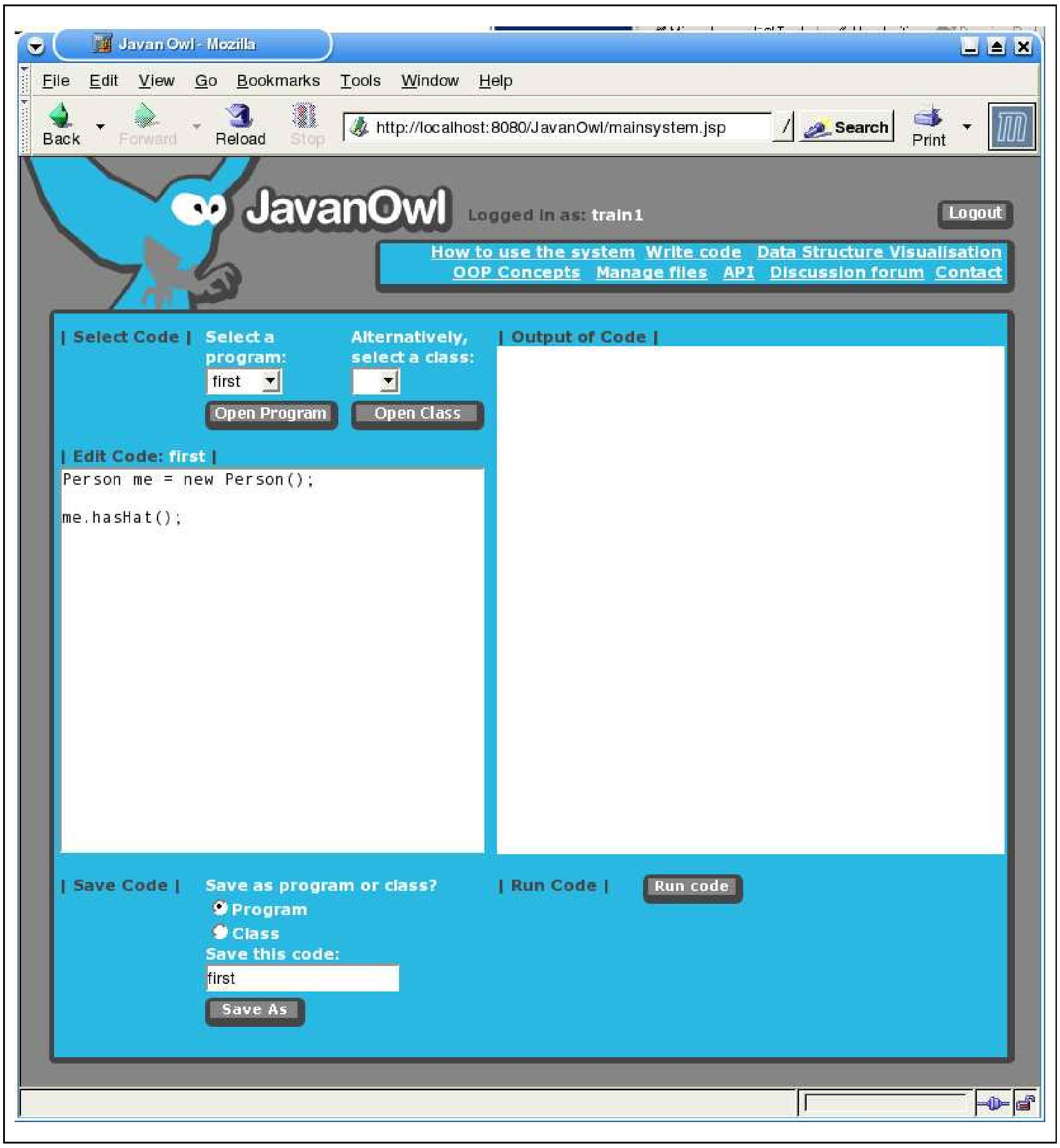
Annie Luxton | Victoria University

Figure 4.2: JavanOwl Homepage

Users are required to register with the system in order to be able to use it. This is necessary because JavanOwl allows users to create and save program files. Whenever users log in to the system, all the files they have previously created and saved are available for them to keep working on. JavanOwl provides an online storage and running facility for simple Java programs that the users write.

4.6.2 Writing and Saving a Program

JavanOwl allows users to create, edit, save, and retrieve Java programs. The interface for this is a simple and straightforward single-screen development area. The development area is split in two — a panel on the left hand side that contains a web form into which users can write code and a panel on the right hand side that displays the output of running their code (as shown in Figure 4.3). Users enter their program code into the web form and save it as a 'program'. By writing programs and saving them with different names, users can build up a repository of Java programs.



Javan Owl - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/JavanOwl/mainsystem.jsp
Search
Print
JavanOwl
Logged in as: train1
Logout
How to use the system Write code Data Structure Visualisation
OOP Concepts Manage files API Discussion forum Contact
| Select Code |
Select a program:
first
Alternatively, select a class:
Open Program
Open Class
| Output of Code |
| Edit Code: first |
| Save Code |
Save as program or class?
Program
Class
Save this code:
first
Save As
| Run Code |
Run code

```
Person me = new Person();

me.hasHat();
```

Figure 4.3: User enters code into the web form and saves it with a filename.

JavanOwl allows users to write simple Java code, that is, sequences of statements such as object creation and method calls, iteration and selection. A library of classes has been written and is provided with the system. This allows users to write programs that create objects, and then be able to run the available methods on the objects created such as the code shown in Figure 4.4. The JavanOwl API is given in the normal JavaDoc format through a link in the system. They can view the API to find which classes and methods are available for them to play with.

```
Person bob = new Person();
bob.hasHat();
bob.setHatColour(green);

House bobsHouse = new House();
bobsHouse.hasChimney();
bob.setChimneyColour(blue);
```

Figure 4.4: Example of code that uses library objects, written in JavanOwl

Aside from using the JavanOwl API, users can write programs that output plain text. Again, these programs may contain simple Java code such as sequences of statements, iteration and selection. Figure 4.5 shows an example of the type of program that users can write that outputs text only. For security reasons, users are not permitted to execute code that accesses any of the system resources. This is explained in more detail in Section 5.2.1.

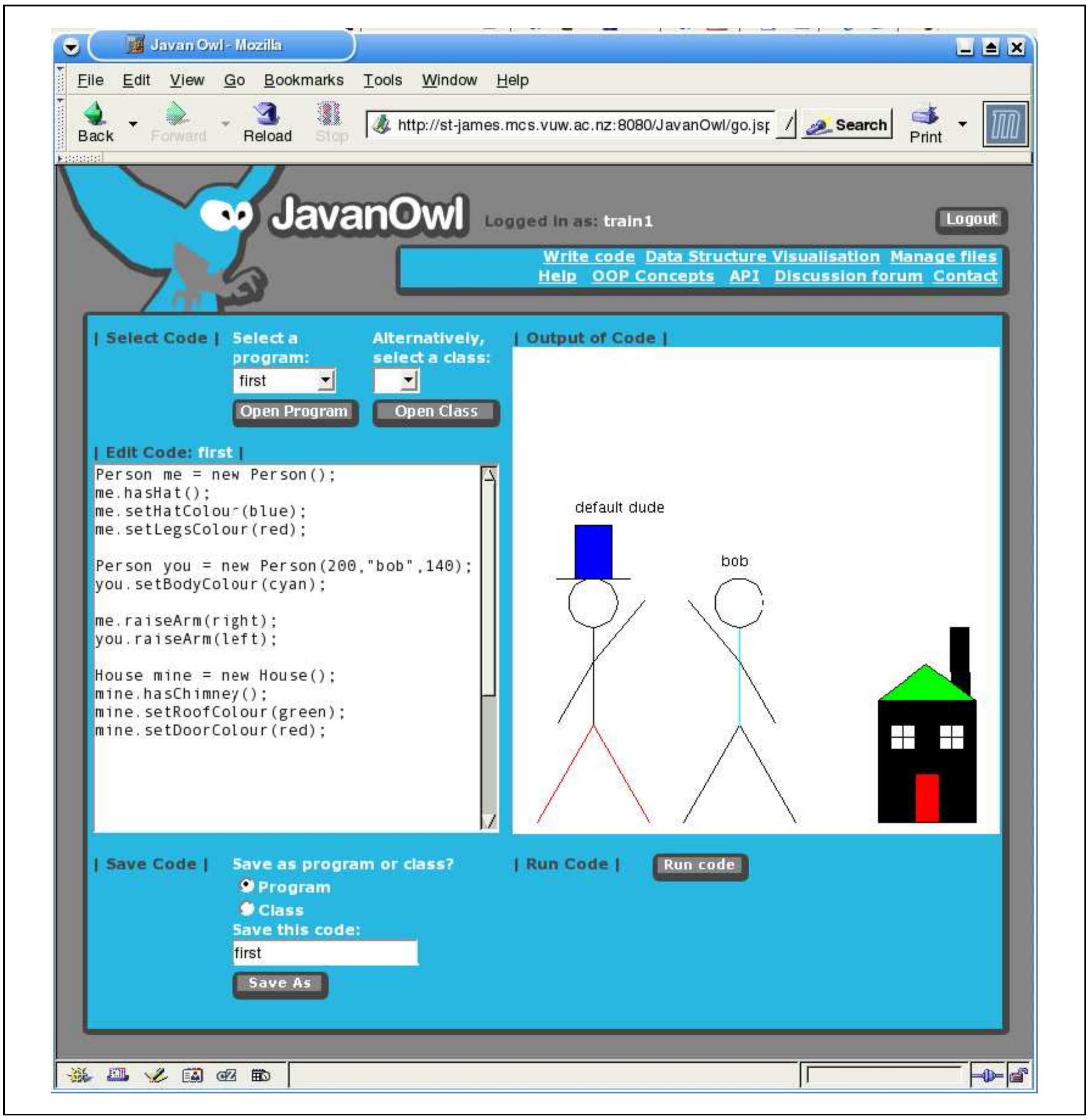
```
output.println("Hello World!");
output.println("");

for (int i = 1; i < 10; i++) {
   if ((i%2) == 0) {
      output.println(i + " is an even number");
   }

   else {
      output.println(i + " is an odd number");
   }
}
```

Figure 4.5: Example of code that outputs text, written in JavanOwl

Users can also write and use their own classes that output text. This is covered in more depth in Section 4.6.5.

4.6.3 Running a Program

Once the user has written or opened an existing program, they can run it. Provided the code is error-free, the browser will display the result in the output pane. In this way, system interaction is based on a principle of immediate feedback with visibility of cause and effect. The type of output shown in the output pane depends on the type of the program they are running. If the user code uses any of the JavanOwl API classes that outputs visualisations, then an animated visualisation of the program code will appear in the output pane. If the user code outputs text only, then the output text will be shown in the output pane. Figure 4.6 shows the state of the JavanOwl system after a program that utilizes the JavanOwl API and produces a visualisation of the program code has been run. Figure 4.7 shows the state of JavanOwl after a textual program has been run.



Javan Owl - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
Search
Print
JavanOwl
Logout
Write code Data Structure Visualisation Manage files
Help OOP Concepts API Discussion forum Contact
| Select Code |
Alternatively, select a class:
Open Program
Open Class
| Output of Code |
default dude
bob
| Save Code |
Save as program or class?
Program
Class
| Run Code |
Run code
Save As

http://st-james.mcs.vuw.ac.nz:8080/JavanOwl/go.jsp

Logged in as: train1

Select a program: first

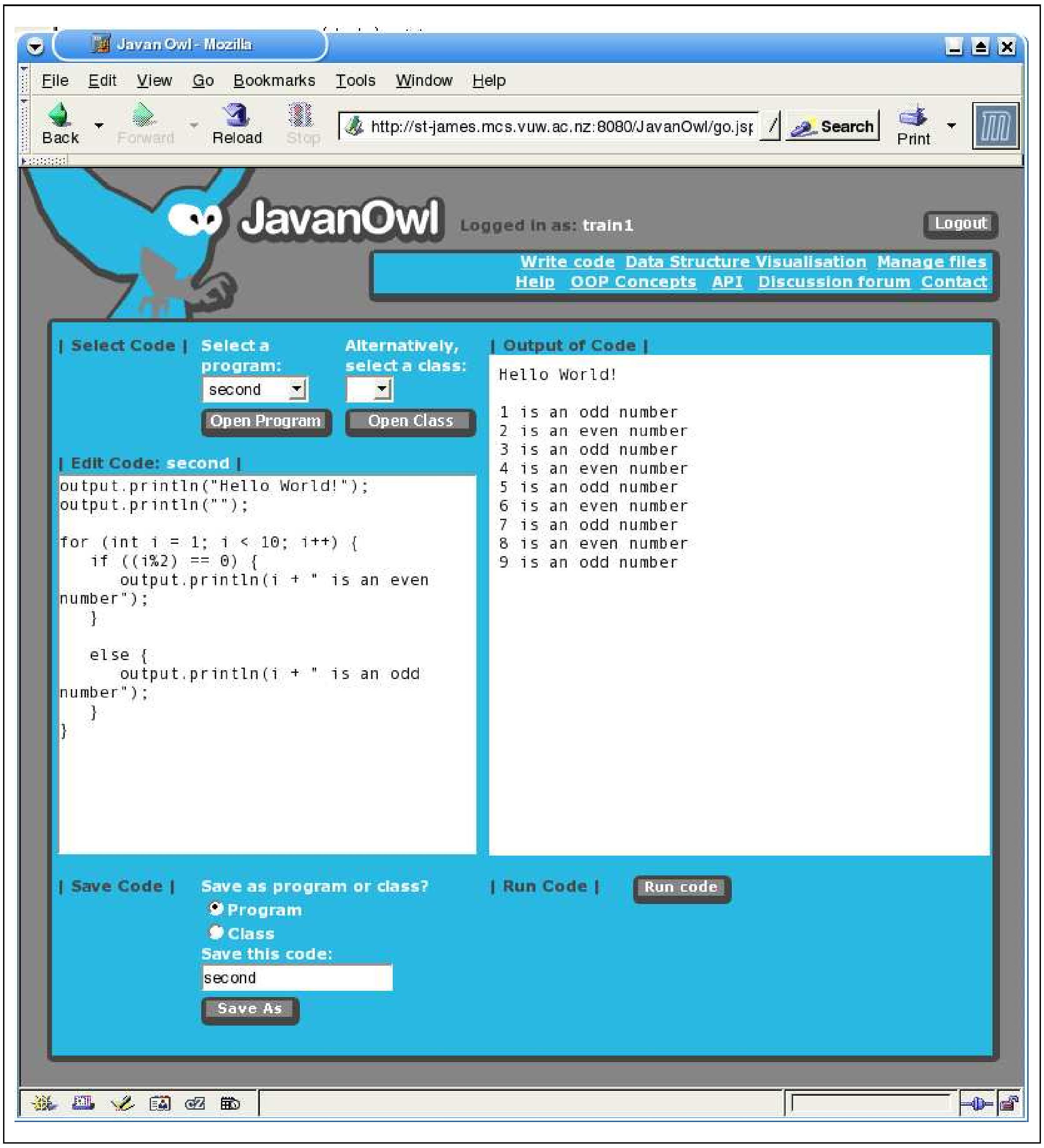
| Edit Code: first |

```
Person me = new Person();
me.hasHat();
me.setHatColour(blue);
me.setLegsColour(red);

Person you = new Person(200,"bob",140);
you.setBodyColour(cyan);

me.raiseArm(right);
you.raiseArm(left);

House mine = new House();
mine.hasChimney();
mine.setRoofColour(green);
mine.setDoorColour(red);
```

Save as program or class? [✓ Program] Class

Save this code: first

Figure 4.6: JavanOwl system after having run a program that outputs program code visualisations



```
output.println("Hello World!");
output.println("");

for (int i = 1; i < 10; i++) {
   if ((i%2) == 0) {
      output.println(i + " is an even
number");
   }

   else {
      output.println(i + " is an odd
number");
   }
}
```

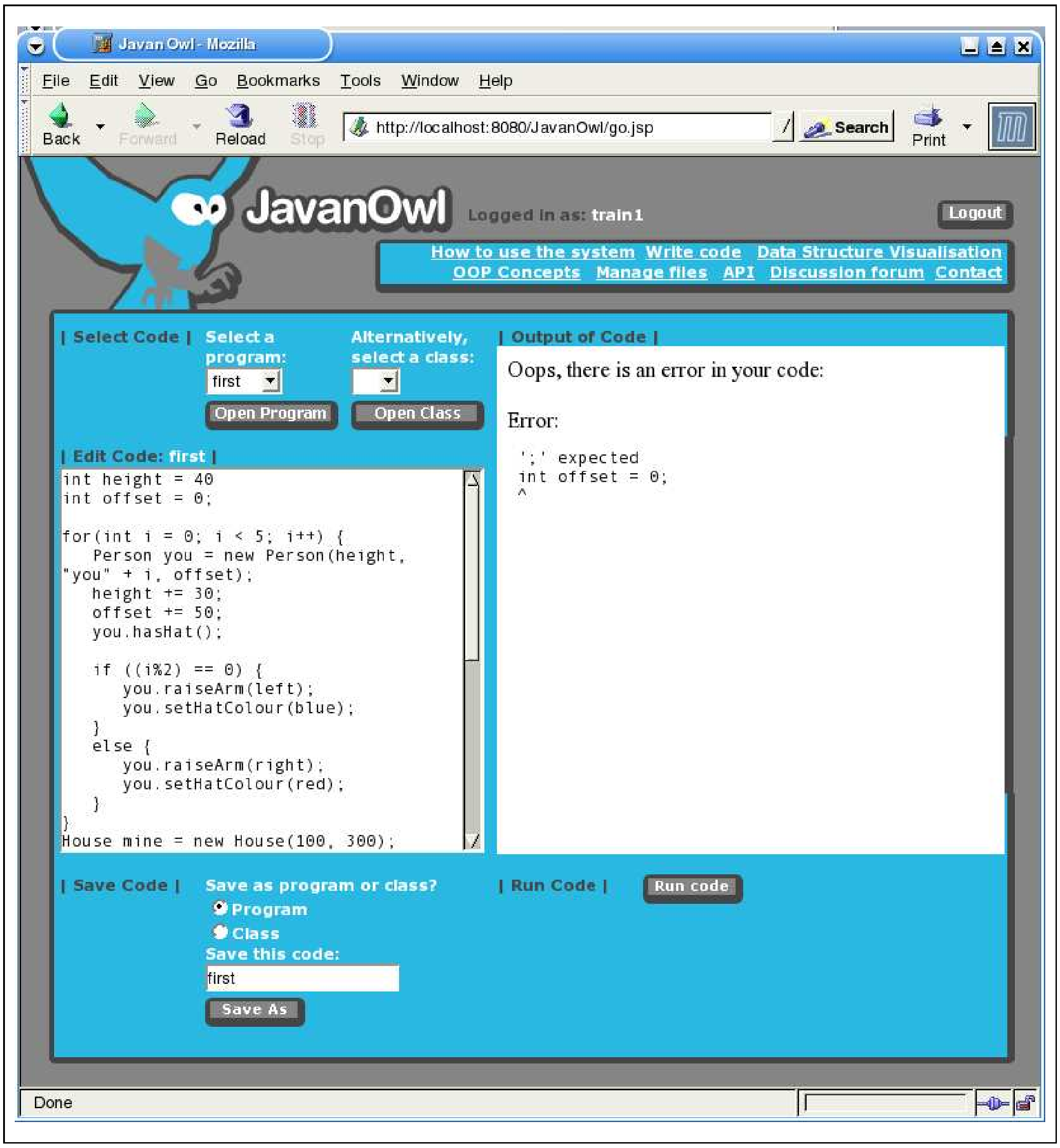
```
Hello World!

1 is an odd number
2 is an even number
3 is an odd number
4 is an even number
5 is an odd number
6 is an even number
7 is an odd number
8 is an even number
9 is an odd number
```

Figure 4.7: JavanOwl system after having run a textual program

4.6.4 Errors in user code

JavanOwl will catch some errors in user code. In particular, syntactic errors that are normally picked up by a compiler are easily dealt with. If a user attempts to run code that has multiple syntactic errors in it, JavanOwl will find the very first error (parsing top-down) and the output pane will display a simple and specific message regarding the error as is shown in Figure 4.8. Runtime exceptions are also caught by JavanOwl and produce a similar error message. This form of debugging is simple but effective.

Infinite loops in code that use visualisations of JavanOwl APIs can also be caught. In the case of infinite loops, an upper limit has been set on the number of animation frames that can be created in generating the visualisations of program code. If the limit is reached, then the visualisation fails and an error message is given to the user advising them of their mistake. In both cases, once the users have been shown their error, they can find it in their code, edit it and re-run it.


Javan Owl - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/JavanOwl/go.jsp
Search Print
JavanOwl
Logged in as: train1
Logout
How to use the system Write code Data Structure Visualisation
OOP Concepts Manage files API Discussion forum Contact
| Select Code | Select a program: first Alternatively, select a class:
Open Program Open Class
| Edit Code: first |
| Output of Code |
| Save Code | Save as program or class? Program Class Save this code: first Save As
| Run Code | Run code
Done

```
int height = 40
int offset = 0;

for(int i = 0; i < 5; i++) {
   Person you = new Person(height,
"you" + i, offset);
   height += 30;
   offset += 50;
   you.hasHat();

   if ((i%2) == 0) {
      you.raiseArm(left);
      you.setHatColour(blue);
   }
   else {
      you.raiseArm(right);
      you.setHatColour(red);
   }
}
House mine = new House(100, 300);
```

Oops, there is an error in your code:

Error:

```
 ';' expected
 int offset = 0;
 ^
```

Figure 4.8: Running user code that contains a syntactic error

4.6.5 Writing, Saving and Using a Class

JavanOwl also allows users to create, edit, save, retrieve and use their own Java classes in the programs they write and run. Within their classes, users are able to write both void methods and methods that return a value. The structure of classes written for JavanOwl is the same as the structure of normal Java classes. Methods are defined as they normally are in Java classes. Although the filenames given to their classes is immaterial since they are simply used to store the classes in their account, the actual class names in the code are

```
public class Hello {

   public void printHelloOnce() {
      output.println("Hello!");
   }

   public void printHelloManyTimes(int count) {
      for (int i = 0; i < count; i++){
         output.print("Hello ");
         output.print(i);
         output.print(" times");
         output.println("");
      }
   }

   public String returnsHello() {
      return "hello";
   }
}
```

Figure 4.9: Example class written in JavanOwl

important. The class name will need to be used to instantiate objects of that class, so no two classes should have the same name. This is the same behaviour that is encountered in normal Java programming. Figure 4.9 shows a class that could have been written and used in JavanOwl.

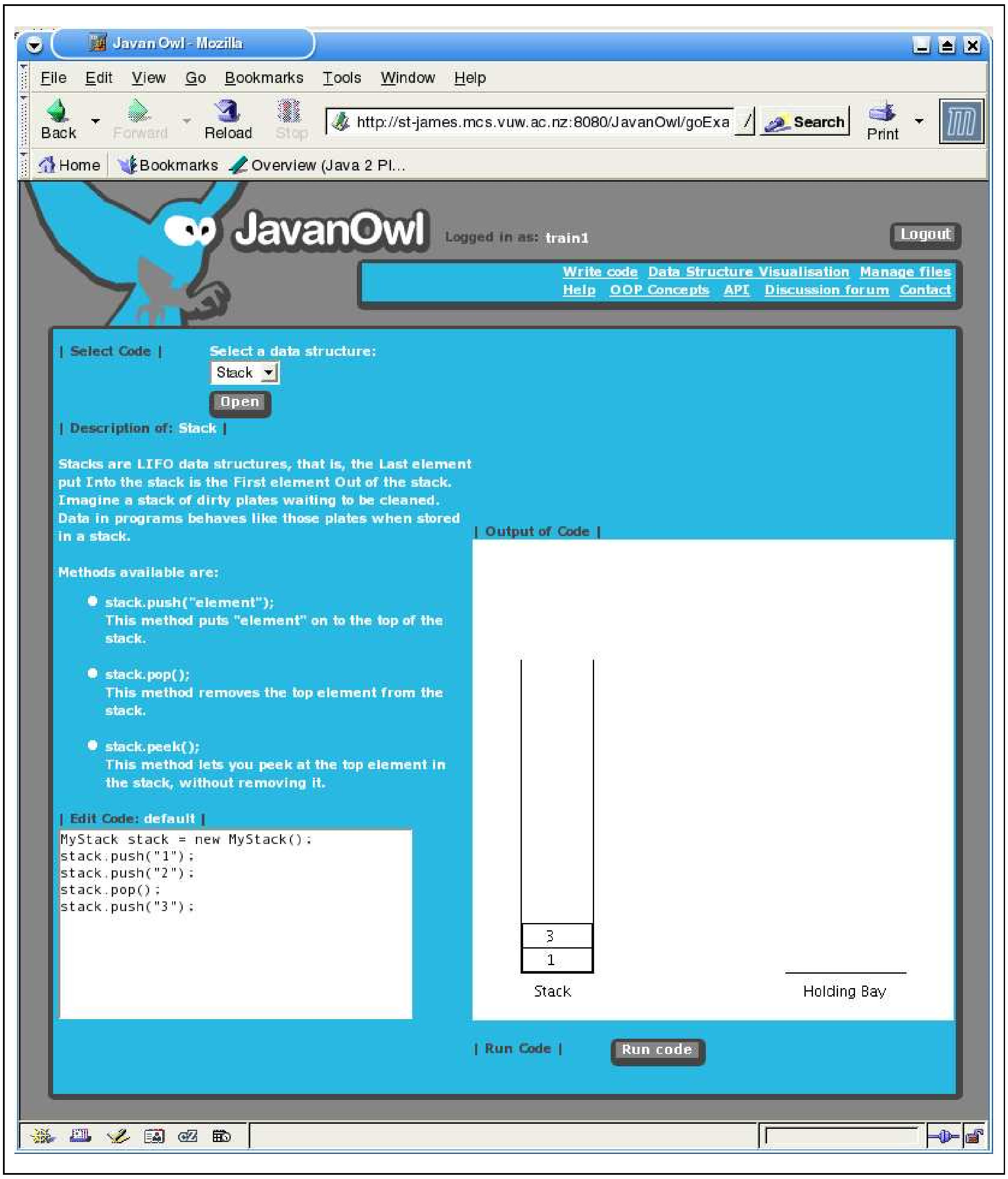
The same split-screen development area that is used to write and run programs is used to write classes. Users enter their class code into the same web form that they write program code in. When they save their code, it must be saved as a 'class'. The class is then saved into their account. Users can write programs that use classes they have written by instantiating an object of that class in their code. In order to run a program that does this, users must first select the classes they have used in their program, essentially *importing* the required classes into their program code. A list of all classes the user has written is displayed next to the 'Run Code' button so that they are able to select all the necessary classes before running their code. Then, provided both the program and the class are error-free and that they have selected all the necessary classes to import, the program will run and the output will be shown in the output pane.

The effect of allowing users to write their own classes is to gear the learning towards an OO-first approach. It is, however, kept optional — users can choose to not write any classes; they are never forced to write their own classes. This functionality was included as a proof-of-concept as well as to cater for those who feel comfortable enough with programming and wish to extend the system by writing and using their own classes.

4.6.6 Studying Data Structures

JavanOwl provides a section for users to study the data structures that are commonly used in programming, such as Stacks, Queues and Arrays. Theoretically, data structures can be hard for students to understand. Often the best way to explain how data structures work is by drawing pictures of the state of the structure before and after calls to methods that store or remove elements from it, thus visually depicting the effect of these methods. This section of JavanOwl provides a place where students can explore the behaviour of a particular data structure. Users can write and run code that uses a data structure, the result of which will be an animated visualisations of their code.

The layout of this section of JavanOwl is very similar to that of the section where users write and run normal programs. The interface is simple: a single-screen development area that is split into two columns. A panel on the left hand side displays the description of a data structure that the user has selected from a drop down list. It also contains a web form into which users can write code. The panel on the right hand side displays the output of running their code. Figure 4.10 shows an example of what is displayed to the user.


Javan Owl - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://st-james.mcs.vuw.ac.nz:8080/JavanOwl/goExa
Search
Print
Home Bookmarks Overview (Java 2 Pl...
JavanOwl
Logout
Write code Data Structure Visualisation Manage files
Help OOP Concepts API Discussion forum Contact
| Select Code |
Select a data structure:
Stack
Open
| Output of Code |
3
1
Stack
Holding Bay
| Run Code |
Run code

Logged in as: train1

| Description of: Stack |

Stacks are LIFO data structures, that is, the Last element put Into the stack is the First element Out of the stack. Imagine a stack of dirty plates waiting to be cleaned. Data in programs behaves like those plates when stored in a stack.

Methods available are:

- stack.push("element");
  This method puts "element" on to the top of the stack.
- stack.pop();
  This method removes the top element from the stack.
- stack.peek();
  This method lets you peek at the top element in the stack, without removing it.

| Edit Code: default |

```
MyStack stack = new MyStack();
stack.push("1");
stack.push("2");
stack.pop();
stack.push("3");
```

Figure 4.10: JavanOwl system after having run a data structure program

Users select a data structure they wish to study from a drop down list. This displays a description of the selected data structure as well as a list of methods that are applicable to that data structure. A small skeleton program also appears in the code text box, which the user can run. The user can also edit the code provided and run it. The output of running their program, provided it is error-free, is an animated visualisation of the program. This provides the user with a graphical representation of their program. The animations have been created to clearly reflect the effect caused by a method call. For example, Figure 4.13 was generated from the code shown in Figure 4.11.

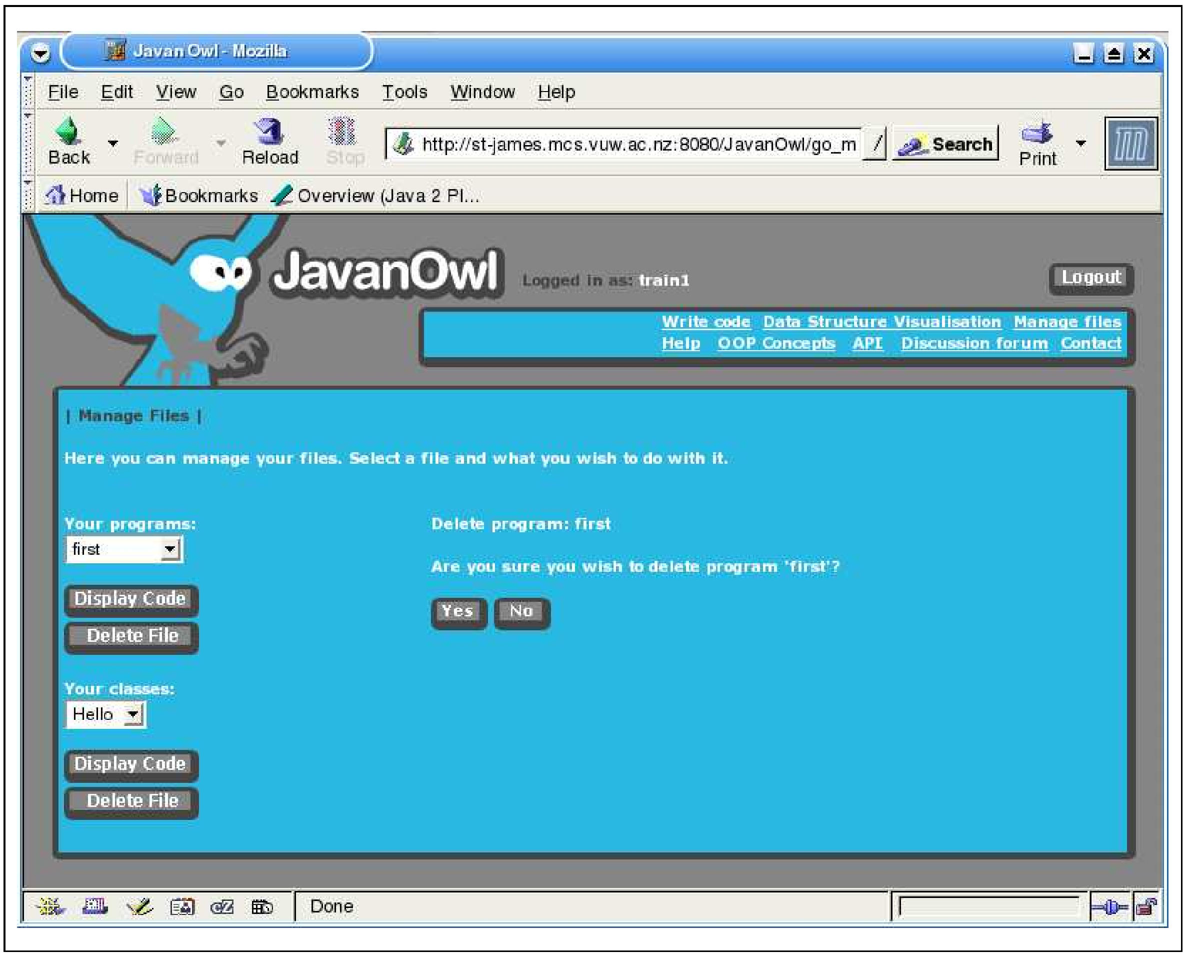
```
MyStack stack = new MyStack();
stack.push("1");
stack.push("2");
stack.pop();
stack.push("3");
```

Figure 4.11: Example of code that uses the Stack class, written in JavanOwl

4.6.7 Managing Your Files

The user is given the ability to manage their files. By means of a separate section in the system, users can view the code and delete programs and classes they have created. This section was added for convenience as well as providing the user with a sense of freedom within the system.

Displaying code simply displays the code on the screen. Deleting code completely deletes the selected program or class from the user's account. The user is given a chance to exit the procedure of deleting a program. Figure 4.12 shows a user attempting to delete a program.



Javan Owl - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://st-james.mcs.vuw.ac.nz:8080/JavanOwl/go_m
Search
Print
Home Bookmarks Overview (Java 2 Pl...
JavanOwl
Logged in as: train1
Logout
Write code Data Structure Visualisation Manage files
Help OOP Concepts API Discussion forum Contact
Display Code
Delete File
Display Code
Delete File
Yes
No
Done

| Manage Files |

Here you can manage your files. Select a file and what you wish to do with it.

Your programs: first

Delete program: first

Are you sure you wish to delete program 'first'?

Your classes: Hello

Figure 4.12: User attempting to delete one of their programs.

4.6.8 User Forum

The system supports collaboration by providing a built-in web discussion forum so as to give users a support system that will allow a collaborative community of peer support and file sharing to arise.

4.7 Program Code Visualisations

One main concern was to provide users with immediate feedback — allowing them to get instant satisfaction from the code they write. One approach to this was to compile and run user code 'on-the-fly'. Another approach we took was to provide program code visualisations also generated 'on-the-fly' that help explain the effects of the code that the user is writing. Together, these two approaches have created a system where the interaction is based on a principle of immediate feedback with visibility of cause and effect. When users submit some code, they immediately receive feedback in the form of a graphical representation of that code. This graphical feedback emphasizes the actions the user chose to run in their program, highlighting the effects of different programming structures.

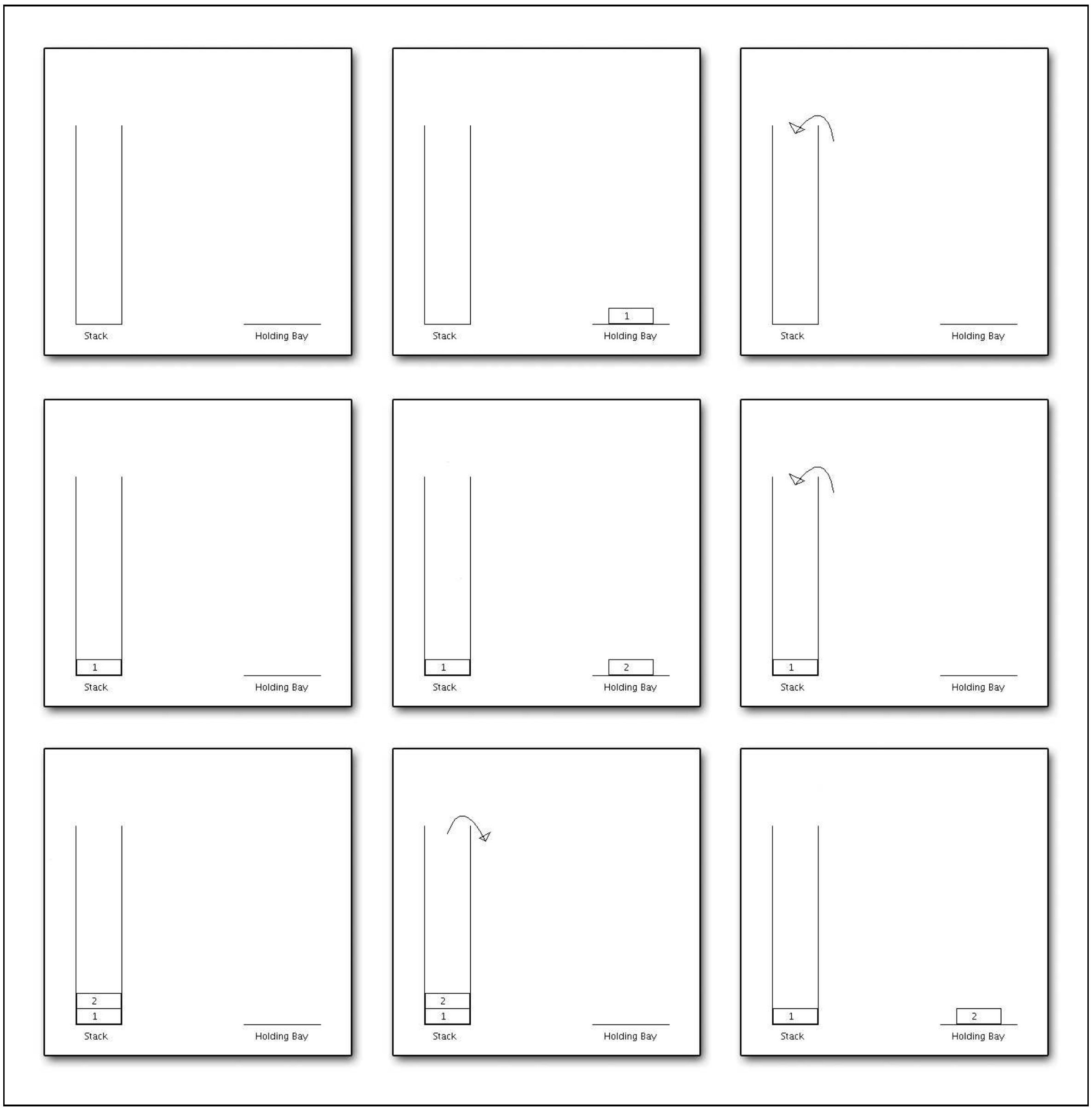


Figure 4.13: JavanOwl animation of a user-written program that uses the Stack class.

JavanOwl creates animated program code visualisations of user code. Each important event in the user's program execution, such as a method call or a constructor execution, draws a picture of the state of the program after the event has occurred. Each one of these images becomes a frame in an animated sequence of images that shows the progression of the program execution. The effect of this is to show the user, in a simple graphical manner, how the user program code develops — how each method call or constructor execution affects the state of the program. Figure 4.13 shows the sequence of frames that would make up the animated visualisation that would be generated from some Stack code. It shows in some detail the effect of different methods in the Stack class.

4.8 JavanOwl Library and API

JavanOwl includes its own library of classes. This allows learners to work with library objects and their methods, through the use of sequences of statements, iteration and selection. The result of executing programs that use the JavanOwl library are program code visualisations.

4.8.1 Classes

So far, the classes provided in the JavanOwl library are the following:

Simple classes provided for interesting exercises:

- **Person class**. This class creates a stick person object. It provides methods to add a hat to the stick person, raise its arms, lower its arms and change the colour of various parts of it's body.
- **House class**. This class creates a house object. It provides methods to add a chimney to the house and change the color of various parts of the house.

Classes provided to help explain data structures:

- **MyStack class**. This class creates a stack object. It provides methods to push, pop and peek strings in to and out of the stack object. This class is intended to be used with run-time visualisation to help learners understand stacks.

4.8.2 JavaDocs and Related Documentation

Standard JavaDocs were created and have been provided for all the classes provided within the JavanOwl library. This is intended to not only give user a reference point for available methods on classes and the respective signatures, but also to familiarize users with the normal JavaDoc format.

Chapter 5

Implementation

5.1 System Architecture

JavanOwl has been implemented using JavaServer Pages (JSP) [5], along with JavaBeans, JDBC, an SQL database, and HTML, all run on the server-side, using the JavaServer Pages "Tomcat" web server; software for which is all freely available. JavanOwl has been designed to be as portable as possible, as Tomcat runs on a wide variety of systems. The web browser clients can be very lightweight and require no plug-ins, and avoids browser compatibility problems. The architecture is therefore easy to provide on both server and
client sides.

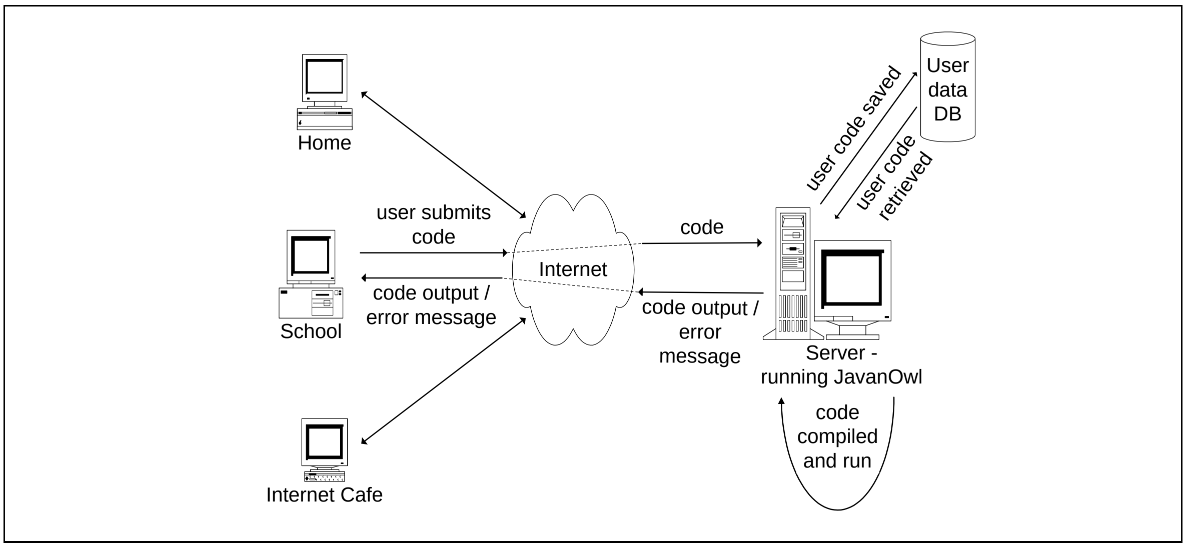


Figure 5.1: JavanOwl system architecture.

5.2 Compile and Run Technology

The system allows users to write simple Java code, including standard object creation and manipulation using the library of classes provided, sequences of basic statements, iteration and selection. The system allows users to write their own classes and use them in programs they write as well. Users enter their code via a web form, which is stored in the database. They don't have to explicitly compile their code, they simply 'run' it. When they run their code, the session bean retrieves their code from the database and writes it into a .JSP file, already wrapped with appropriate wrapper code. In order to catch compile errors, the JSP file is pre-compiled on the server side using a Java Runtime process to run a pre-defined Ant Task, and any errors that are caught are parsed and then output to the user. If the user code is error-free, 'Tomcat's Jasper Engine then converts the JSP file into a corresponding .Java file. This file is then compiled by 'javac', the Java Compiler, further converting it into its corresponding .Class file. The user's code has therefore essentially been turned into a valid Java Servlet. This servlet is executed on the server and the user's browser is then redirected to a page where the result of running the Java Servlet is displayed.

This method of compiling and running user code has been successful thus far. It has several advantages:

- It does not require large amounts of (possibly non-terminating) Java Runtime processes to be initiated — our first attempt at compiling and running Java programs for this system included creating Java Runtime processes from the session bean, and executing the appropriate commands to compile and run the code. This method did not work as well as we had hoped. Our experience showed that even with only a few concurrent users, the system would overload the server's processor with rogue threads that would eventually crash the system.

- It uses the web server's own technology to compile and run simple Java code. This means that no new applications specifically designed to compile and run Java over the web are needed.

- It allows users to write very simplified Java code — without the usual 'public static void main (String[] args)' at the beginning. This gives users a chance to get some code working very quickly (just one line is sufficient), without needing to learn all the syntax involved in writing a complete stand alone Java program.

- It is flexible enough to allow users to write their own classes and use them in their code. Since JSP allows embedded classes to be written within .JSP files, it was a simple enough extension to include user-written classes in the .JSP file written together with the wrapper code. The 'Tomcat' webserver handles the rest.

5.2.1 Security

Some consideration was given to security issues even though the topic is out of the scope of this project. Since JavanOwl allows users to write and execute code by executing their code on the server, the implications of this are quite serious. The user could in theory use JavanOwl to execute code that accesses the servers system resources or writes to the disk. Used in a malicious manner, this could cause the server to shutdown, crash or worse, lose all it's data.

However, the "Tomcat" web server that we are using to host JavanOwl has its own implementation of Java's Security Manager [35]. The developer must first define the permissions that a class loaded by Tomcat should have by editting the default `catalina.policy` policy file. The effect of this is to disallow any web applications to access the servers system resources. Once this is done, the Tomcat web server should be started with a special command `$CATALINA_HOME/bin/catalina.sh start -security` to tell it to run in secure mode — that is, apply the permissions set in the `catalina.policy` file. By setting the correct permissions, security should not be a concern in JavanOwl.

5.2.2 Code Samples

Below are some code samples of how the user's code gets transformed into valid JSP that the "Tomcat" web server can convert into Java files or servlets, together with explanations of why the respective JSP was added.

Transforming Code That Uses the JavanOwl Library

Figure 5.2 shows some user-written code that makes use of JavanOwl library objects such as Person and House objects, together with the respective JSP version of the same code. The JSP wrapper code begins by importing a few necessary classes. The JSP wrapper code also defines a few variables in order to simplify the code that users have to write as much as possible. For example, when JavanOwl transforms code that contains Person or House objects, it defines variables of type Color and gives them names corresponding to the colour they represent. This is so that JavanOwl users can just type the word 'green' when calling methods that require a colour to be specified, rather than needing to specify the object 'Color.green' which would be perhaps too advanced for novice programmers. Therefore, when the JSP file is converted into a servlet and executed, the colours which the user specified as arguments in their methods are actually variable names that refer to real Java Colour objects. At the end of the file, it creates the visualisation of the program code by calling on the Image class which was specifically written to write the visualisations out to animated GIFs.

```
Person jerome = new Person(150,"jerome",20);
jerome.hasHat();
jerome.raiseArm(left);
jerome.raiseArm(right);

jerome.setHatColour(red);
jerome.setBodyColour(green);
jerome.setHatColour(orange);

House yours = new House();
yours.setRoofColour(blue);
```

```
<html>
<%@ page import="mcs.javanowl.*"%>
<%@ page import="sciexp.*"%>
<%@ page import="java.awt.Color"%>
<%@ page import="java.util.*"%>
<%@ page import="java.io.*"%>
<%@ page import="sciexp.proxy.Person"%>
<%@ page import="sciexp.proxy.House"%>
<jsp:useBean id="Mgmt" class="mcs.javanowl.SQLmgmt" scope="session" />
<%
   Color blue = Color.blue;
   Color cyan = Color.cyan;
   Color yellow = Color.yellow;
   Color black = Color.black;
   Color white = Color.white;
   Color red = Color.red;
   Color pink = Color.pink;
   Color green = Color.green;
   Color gray = Color.gray;
   Color orange = Color.orange;
   String right = "right";
   String left = "left";

      Person jerome = new Person(150,"jerome",20);
      jerome.hasHat();
      jerome.raiseArm(left);
      jerome.raiseArm(right);

      jerome.setHatColour(red);
      jerome.setBodyColour(green);
      jerome.setHatColour(orange);

      House yours = new House();
      yours.setRoofColour(blue);

  Image image = new Image();
  Mgmt.setLastImage(image);

%>
<jsp:forward page="/output.jsp"/>
</html>
```

Figure 5.2: Transformation of user code that uses Person and House classes into JSP

Figure 5.3 shows some user-written code that makes use of the Stack objects, together with the respective JSP version of the same code. Like the JSP wrapper code used to transform user code that utilizes other JavanOwl library objects, the JSP wrapper code in this case must first import a few necessary classes at the beginning of the file. It must also create the visualisation of the program code at the end by calling on the Image class. However, no extra variables need to be defined in the JSP wrapper code to simplify user code in this case as there is nothing to simplify.

```
MyStack stack = new MyStack();
stack.push("1");
stack.push("2");
stack.pop();
stack.push("3");
stack.peek();
```

```
<html>
<%@ page import="mcs.javanowl.*"%>
<%@ page import="sciexp.*"%>
<%@ page import="java.awt.Color"%>
<%@ page import="java.util.*"%>
<%@ page import="java.io.*"%>
<%@ page import="sciexp.proxy.MyStack"%>
<jsp:useBean id="Mgmt" class="mcs.javanowl.SQLmgmt" scope="session" />
<%
      MyStack stack = new MyStack();
      stack.push("1");
      stack.push("2");
      stack.pop();
      stack.push("3");
      stack.peek();

   Image image = new Image();
   Mgmt.setLastImage(image);

%>
<jsp:forward page="/examples_output.jsp"/>
</html>
```

Figure 5.3: Transformation of user code that uses Stack class into JSP

Transforming Code That Outputs Text and Uses User-Defined Classes

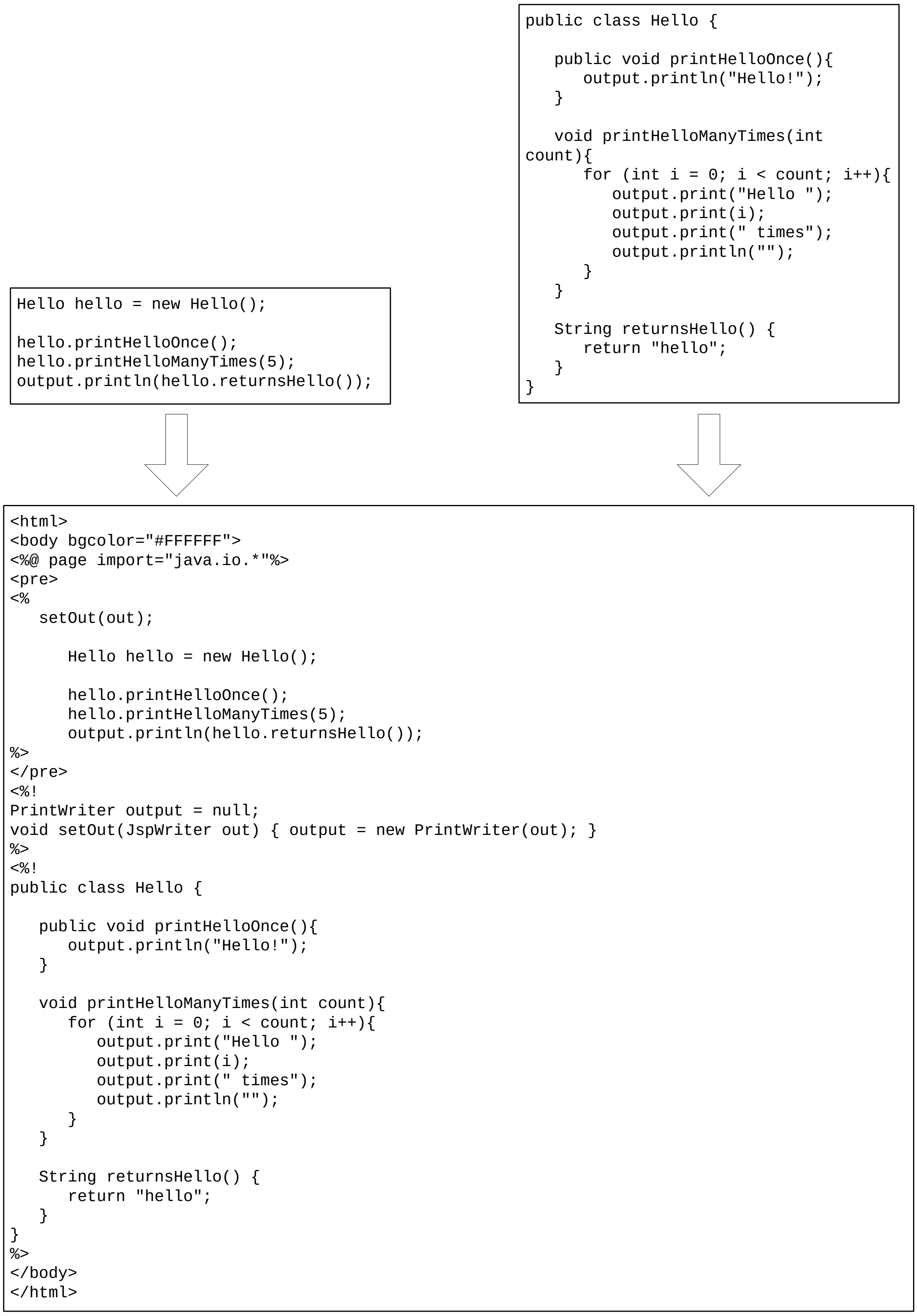
Figure 5.4 shows some user-written code that outputs text, together with the respective JSP version of the same code. Figure 5.5 on the other hand shows some user-written code that uses a user-written class, together with the respective JSP version of the same program code. In both cases, the JSP wrapper code required for this type of user code first imports necessary classes. If the user has defined any classes and is using them in their program code as in Figure 5.5, then the whole class is also written into the file as an *inner* class.

```
output.println("Hello World!");
output.println("");

for (int i = 1; i < 10; i++) {
   if ((i%2) == 0) {
      output.println(i + " is an even
number");
   }

   else {
      output.println(i + " is an odd
number");
   }
}
```

```
<html>
<body bgcolor="#FFFFFF">
<%@ page import="java.io.*"%>
<pre>
<%
   setOut(out);

      output.println("Hello World!");
      output.println("");

      for (int i = 1; i < 10; i++) {
         if ((i%2) == 0) {
            output.println(i + " is an even number");
         }

         else {
            output.println(i + " is an odd number");
         }
      }

%>
</pre>
<%!
PrintWriter output = null;
void setOut(JspWriter out) { output = new PrintWriter(out); }
%>
</body>
</html>
```

Figure 5.4: Transformation of user code that outputs text into JSP

The most interesting feature about this wrapper code is that it declares a PrintWriter field called `output` and sets it to null. It also declares a method that sets this PrintWriter field equal to a PrintWriter created using JSP's JSPWriter variable *out*, which is passed to it as a argument. Before any of the user's code gets executed, this method is called with the default JSPWriter *out* object as it's argument. The effect of this is to define the *output* variable that students can call to print messages to the output pane as a PrintWriter object rather than using the default JSPWriter object. The reason for doing this is that many of the methods defined for the JSPWriter objects throw IOExceptions. Without this JSP wrapper code, if users attempts to write code such as `out.println("hello")` to print a string the output pane using the default JSPWriter object *out*, then they would need to put it within a *try catch* pair. Similarly, without this JSP wrapper code, if a user writes a method within their own class that writes out to the output pane, then they would need to declare the method to throw an *IOException*. The PrintWriter class has been created to provide the same functionality as classes like the JSPWriter class, except without throwing IOExceptions. Therefore, by wrapping the default JSPWriter object *out* in a PrintWriter object, the result is an object that has the same functionality as *out* had, except it doesn't throw IOExceptions. This means that users can write simplified code — they do not need to worry about throwing or catching exceptions. Learning about exceptions is out of the scope of JavanOwl.



```
Hello hello = new Hello();

hello.printHelloOnce();
hello.printHelloManyTimes(5);
output.println(hello.returnsHello());
```

```
public class Hello {

   public void printHelloOnce(){
      output.println("Hello!");
   }

   void printHelloManyTimes(int
count){
      for (int i = 0; i < count; i++){
         output.print("Hello ");
         output.print(i);
         output.print(" times");
         output.println("");
      }
   }

   String returnsHello() {
      return "hello";
   }
}
```

```
<html>
<body bgcolor="#FFFFFF">
<%@ page import="java.io.*"%>
<pre>
<%
   setOut(out);

      Hello hello = new Hello();

      hello.printHelloOnce();
      hello.printHelloManyTimes(5);
      output.println(hello.returnsHello());
%>
</pre>
<%!
PrintWriter output = null;
void setOut(JspWriter out) { output = new PrintWriter(out); }
%>
<%!
public class Hello {

   public void printHelloOnce(){
      output.println("Hello!");
   }

   void printHelloManyTimes(int count){
      for (int i = 0; i < count; i++){
         output.print("Hello ");
         output.print(i);
         output.print(" times");
         output.println("");
      }
   }

   String returnsHello() {
      return "hello";
   }
}
%>
</body>
</html>
```

Figure 5.5: Transformation of user code that uses a user-defined class into JSP

5.3 Server Side Storage

In order to increase accessibility JavanOwl has been made portable and client-lightweight with as much processing as possible done on the server side. All the JSP files and JavaBeans are stored on the server. All the user's code is stored in a relational SQL database stored on the server. All requests made to JSP pages are sent off to the processor, processed, and sent back to the client as HTML.

The advantage of making the system as client-lightweight as possible is that no special plug-ins are required on the client machine. The need for plug-ins can cause problems as they are not necessarily always available for all browsers and platforms, nor is there always a standard between plug-ins for different platforms. Not using plug-ins avoids the users needing to download special plug-ins for their browsers. JavanOwl users can use any platform and any browser to use the system, increasing accessibility. On the other hand, a disadvantage of this is that the speed of the system relies heavily on the speed of the internet connection of the client's machine and the processing power of the server machine. If the server were to be held up by some other process the whole system may slow down or possibly collapse.

5.4 AOP and AspectJ for Program Code Visualisation

AOP and AspectJ were specifically designed to address cross-cutting concerns. Therefore, since program monitoring is considered to be a cross-cutting concern, AspectJ was used to monitor the objects that the users created and manipulated in their code. The information gathered this way was then used together with the Graphic2D class in Java's API and a GIF animator class to generate the visualisations of the user's program code.

One aspect was written in AspectJ for JavanOwl. This aspect was used to monitor all important join points in the JavanOwl API, that is, constructor executions and calls to methods of classes in the JavanOwl API. Therefore, pointcuts were defined to pick out each of these join points. Figure 5.6 shows one such pointcut. This pointcut, called `changePersonPC` was defined on methods that changed a Person object. With the use of wildcards and logical operators, this pointcut captures calls to all methods defined for the Person class that either begin with the word *set*, *has* or *raiseArm*. The pointcut uses the designator *call* to capture information about when the methods being monitored are called since that is all the information we needed to collect for the purposes of the visualisations to be drawn. It also captures the callee Person object.

```
pointcut changePersonPC(Person person):
    ((call(void Person.set*(..)) || call(void Person.has*(..)) ||
    call(void Person.raiseArm(..))) && target(person));
```

Figure 5.6: An example pointcut defined for JavanOwl, written in AspectJ

Since pointcuts do not actually *do* anything, advices were written to contain code that would be executed when a pointcut picked out a join point. Figure 5.7 shows an advice written for the pointcut in Figure 5.6. It is scheduled to be triggered *after* the call to the methods defined in the pointcut. This is done so that the visualisation is drawn the call to a particular method is finished, just in case the target program does not execute properly. The advice in Figure 5.7 first adds the callee Person object, data that was collected using the pointcut, to a list of objects kept by the aspect. The advice then creates a new BufferedImage — essentially a new empty frame of the animation. The advice iterates through the list of all objects that have been created or changed in the program so far, drawing each of them into the new frame using the Graphics2D library. The result of this is a BufferedImage with the current state of the program drawn in it. This BufferedImage is then added to a list of BufferedImages kept by the aspect. The list of BufferedImage will be drawn at the end of the program, creating visualisation which is shown to the user.

```
after(Person person): changePersonPC(person) {

    getSoFar().add(person);
    BufferedImage bi = new BufferedImage(400,400, BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D) bi.getGraphics();
    g2d.setColor(Color.white);
    g2d.fill(new Rectangle(0, 0, 400, 400));

    for (Iterator iter = getSoFar().iterator(); iter.hasNext();){
        Object o = (Object)iter.next();
        Class c = o.getClass();
        String s = c.getName();

        if (s.equals("sciexp.Person")){
            Person p = (Person) o;
            p.makePerson(g2d);
        }

        else {
            House h = (House) o;
            h.makeHouse(g2d);
        }
    }

    getBufferedImages().add(bi);
    g2d.dispose();
}
```

Figure 5.7: An advice defined on the changePersonPC pointcut

Pointcuts and advices such as the ones described above were written to handle all calls to methods and constructor executions in the users program code. Each advice created a BufferedImage and drew into it the current state of the program during its execution. Therefore, it follows that each method call or constructor execution caused a new BufferedImage to be created. By the end of the target program execution, a list of BufferedImages is created. This list of BufferedImages is then used to create the animations of the program code since each BufferedImage represents a momentary state of the program. The effect of this is to have one animation frame for each method call or constructor execution, resulting in a graphical visualisation of the effects of the methods and constructors called upon by the target program.

5.4.1 GIF Generation

When a user program has been run by the user and the aspect has created a corresponding list of BufferedImages that shows each individual state of the program during its execution, an animated GIF is generated. JavanOwl uses a GIF animator encoder package called `to.mumble.GIFCodec` to do this. This animated GIF encoder was written by the Mumble Internet Services group, to be used in Servlets or other server-side applications and so is quite suited to JavanOwl.

```
Person me = new Person();
me.hasHat();
me.setHatColour(blue);
me.setLegsColour(red);

Person you = new Person(200,"bob",140);
you.setBodyColour(cyan);

me.raiseArm(right);
you.raiseArm(left);

House mine = new House();
mine.hasChimney();
mine.setRoofColour(green);
mine.setDoorColour(red);
```

Figure 5.8: Example of code that uses library objects, written in JavanOwl

A Java class that handles the creation of the animated GIFs we wrote. This class receives the list of BufferedImages created by the aspect and uses the `to.mumble.GIFCodec` package to create the animated GIFs. This is done by adding each BufferedImage from the list into an AnimGifEncoder object. The result is an animated GIF, where each frame is a BufferedImage. The animation then displays a sequence of states of the users program, highlighting the effects of each method or constructor call. Figure 5.9 shows the frames of an animated GIF created by the code in Figure 5.8 which was written using JavanOwl.

Rather than creating a file on the server and writing the animated GIF to it, a servlet was written to handle this. When a user requests a page with an animated GIF on it, the animated GIF is written directly out to the response's OutputStream object. Doing this avoids having to create a file on the server which could cause problems with duplicate filenames and hard-drive space since many users may all use the system at once. Also, this avoids problems experience with image caching in earlier versions of JavanOwl. Visualisations play an important role in the JavanOwl system, so it was important that the visualisations that were being shown to the user were not being cached by the browser. That would have been very misleading and incorrect.

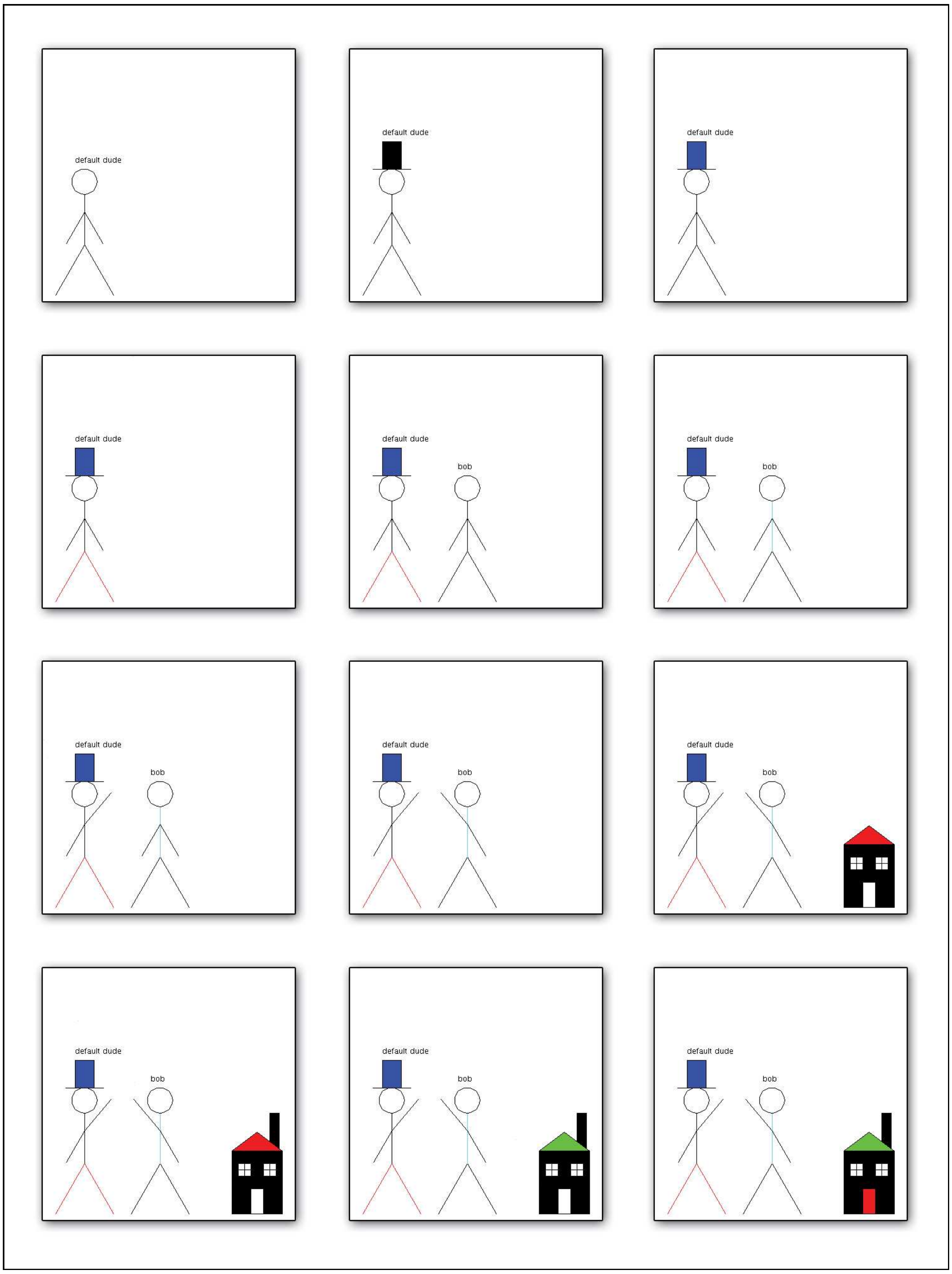


Figure 5.9: JavanOwl animation of a user program that uses the JavanOwl API

Chapter 6

Discussion

JavanOwl was created in response to all the problems identified with education in Computer Science, targeting novice programmers in introductory Computer Science coursesin in particular. It was implemented using the technologies which exhibited the largest number of benefits. In order to evaluate the effectiveness of JavanOwl as a programming tool for beginners, these technological choices must be considered. The following is a run down of some commonly asked questions regarding decisions made about it's design, as well as some answers to those questions, based on observations made.

- What are the advantages of making JavanOwl a web-based system?
- What are the advantages of using JSP as the main web programming technology for the backend of JavanOwl?
- What are the advantages of using AspectJ to monitor the programs written by the users?
- Is AspectJ expressive enough to capture all the data that may be required?
- How effective are the program code visualisations provided by the JavanOwl system?
- How well can JavanOwl scale for larger groups?

6.1 Web advantages

The aim of this project was to make the experience of learning to program for beginner programmers easier and more bearable. One way to do this was to target students both before and during introductory Computer Science courses. The easiest way to gain access to this wide a variety of people is to utilize the web to distribute the system. The Internet provides a cheap, easy and rapid method of disseminating information and applications alike. Users do not need to own a computer — they simply need access to one. Since JavanOwl is completely web-based and client side lightweight, requiring no plugins or applets, there are no technical requirements involved in using it. This also includes requirements such as CPU speed or operating system type, so any old computer with an Internet connection and a browser will be able to run JavanOwl. Users don't need to download or install any software, making the system simple to begin using. This is important as the target audience for this system, novice programmers, often don't have much confidence with computer-usage in general, so avoiding the need for any complicated downloads and installations may encourage them to use the system. Also, web usage is high amongst today's population, even for people who suffer from a low level of technical confidence. Such people still use the Internet to use systems such as online email services or simply to read websites. The hope here is that by being web-based and client side lightweight, JavanOwl will help these people make the transition from using the aforementioned services to learning to program.

The online nature of JavanOwl makes updating and maintaining it very straightforward. Changes need only be made on the server-side and not on separate client machines. Updated versions could be deployed overnight during a pre-announced time and users would see the updated version of the system the next time they log in. This type of maintenance then appears invisible to the users, creating a seemingly seamless updating process.

6.2 JSP advantages

JSP provides a way of adding a lot of extra value to web-based systems. As explained in section 3.1.3, since JSP is kept and executed solely on the server side it is one of the few existing technologies that does not cause compatibility problems. JavaScript and Java Applets can both be used to add significant value to web-based systems, but they can both potentially introduce problems to do with browser compatilibity and needing the user to download relevant plugins. JSP on the other hand has allowed the major functionality of JavanOwl to be built as well as maintaining the client side lightweight. Unlike applications like Flash, however, JSP does not produce very high quality graphics as all we can work with is Java's graphics API. We have chosen to forgo the aesthetics that could have been gained by having effulgent eye-catching graphics produced by JavaScript or Flash in order to maintain a stable, platform and plugin independent, lightweight system.

Another major advantage of using JSP to build JavanOwl was it's close relation to the language that the system actually teaches — Java. Since JSP is essentially an extension of Java for the purposes of web programming, this means that JavanOwl was created using the same technology that it is designed to teach. The consequence of this is that the implementation of certain aspects of the JavanOwl system, such as the compile and run functionality, was greatly facilitated. As described in section 5.2, Tomcat's own mechanism for executing JSP was used to compile and run user-written Java code. In a sense, JavanOwl can be thought of as an educational extension to JSP.

6.3 AspectJ advantages

We have examined a few different methods of program monitoring to collect data about interesting events in program execution. The event-driven technique provides the benefit that it is simple to use and can capture many types of interesting information about differene events. Unfortunately, this method does have the disadvantage of being intrusive and thus needing to alter the target program. In terms of JavanOwl, although this could be done, it is not a convenient solution. An API that needs to be altered everytime more information needs to be captured in order to produce different visualisations would not be suitable. AspectJ provides this adequate degree of separation between the actual program code and the code that produces the visualisations. AspectJ is also powerful enough to collect the kind of program execution information that is required for JavanOwl.

6.4 Visualisation using AspectJ

For visualisations, the programmer really needs to decide what changes in the program state are important enough to be visualised. The programmer can decide, depending on the task at hand, what the granularity of the data collected should be. AspectJ's join points are powerful enough to be used to capture information about many interesting events in program execution. For the purpose of beginner programmers, the kinds of events that should be monitored in order to obtain information that can be used to provide adequate visualisations are events such as method calls, method executions, object instantiations, constructor executions, and field references. AspectJ can easily monitor these events using join points, pointcuts, designators and advices. In order to capture the type and granularity of information needed for the visualisations that JavanOwl currently produces, the main designators used are the *execution* and *call* designators. Using pointcuts and these designators, JavanOwl collects information about when a library class method body executes or when a library class method is called. Thus, the names of all the interesting methods and constructors in the JavanOwl library classes, together with wildcards and logical operators, have been used in the execution and call designators in order to monitor them in the program execution.

Visualisation developers must write the visualisation code into advices. Advice code gets executed at the join points that have been picked out by all the pointcuts defined in the aspect. Advices should define what the aspect should do with the captured information. Since AspectJ is a Java extension to AOP, these advices are written in Java. This facilitates the process of writing code to produce visualisations, as the visualisation developers do not need to learn a new language. Java's Graphic2D API can be used to produce sufficiently effective visualisations.

The main advantage of using AspectJ to produce visualisations is that all this information can be gathered without any form of intrusiveness. The target program does not need to be modified at all — the code to generate the visualisations is kept completely separate from the target code by residing in these aspects. Therefore, for the purposes of visualising program code, AspectJ is a powerful tool to collect information with. It is simple to use and maintains a good degree of separation between the code that generates the visualisations and the target program. This makes Aspectj an obvious choice for the task of creating visualisations.

6.5 JavanOwl visualisations

The advantages gained by using visualisations in education are many. In particular, the visualisation of the user's program code in JavanOwl has the potential to make the programming experience more entertaining. JavanOwl's program code visualisations are in the form of animated images where each frame shows a significant step in the user's program code. The final frame is a visual representation of state that the program is in at the end of it's execution. Not only does this give the user a different view of the final state of the program, but it also helps the user see what effect each important event (such as method invocations or object construction) actually has on the overall state of the program. It also highlights the order in which the statements in their code actually occur, visibly showing the user the nature of top-down nature procedural program execution. This is particularly important when control structures such as if statements or loops are used in the code, as these cause the flow of control to be invisibly passed around rather erratically.

6.6 Scalability

Since JavanOwl is web-based, it can be used by many users at the same time. By providing an online discussion forum, JavanOwl allows for a collaborative community of file and idea sharing to exist. This means that users can work together on JavanOwl, building upon eachother's projects and helping eachother learn. The system is therefore caters for larger groups well.

Chapter 7

Experience and Usability Studies

7.1 Science Experience

Every year our university hosts a “science experience” week for secondary school students. During the week there are various activities, and our school hosts an activity on learning to program. We have traditionally used Java and a conventional programming environment, but recently we ran the activity using JavanOwl.

The students were from various nearby schools, and were in their 3rd year of high school making them around 15 years old. This is the type of person we would like to support in getting more experience in programming before they arrive at university. The students came from a wide range of backgrounds and had varying abilities, but all had chosen to participate in the science experience.

Altogether there were about 70 students who took part in our activity on learning to program. The students worked in 8 separate sessions of gender-mixed groups of about 8 students each. Each session was one hour long, and each student participated in only a single session. During the sessions, the students were brought into a computer lab where after a brief explanation of some basic OO terminology and some simple Java commands and syntax (such as object creation and method calls), they were encouraged to log into the system and attempt to write some code that would utilize the classes provided.

A library containing two simple classes was provided: a ‘stickperson’ class and a ‘house’ class. These two classes were created to provide interesting exercises for the students to play with. When objects of these classes were created, an image of the appropriate object was shown in the ‘output’ side of the split-screen of JavanOwl. The students were able to modify the method calls and then re-run the programs generating other images. Overall, the feedback was rapid and visual as the objects they created were pictorially shown as soon as they pressed ‘run’, provided their code was correct.

At the end of each one hour session, the students were asked to complete a simple activity evaluation survey. This chapter presents some observations arising from the sessions, followed by some consideration of the results of the evaluation survey.

It is worth noting that from an administrative point of view, the use of the web reduced the administative work needed: for example, no special setup or cleanup was needed for individual computers or even user accounts.

7.2 Science Experience Results

7.2.1 Observations

As expected, all students wrote incorrect code at some point during the exercise. At the time we ran these sessions, we had considered utilizing a Java parser to catch errors in user code and to present them to users in a user-friendly manner, but this had not yet been implemented. Therefore, if a user attempted to 'Run' incorrect code, the browser would forward them a relatively unfriendly raw JSP 'Error' page, crudely formatted, that showed a terse error message, the line number of the error, and a detail stack trace. We were not happy about this, but had not had to time to improve it, so we stood by apprehensively.

We observed the following behaviour : students would write one or two lines of code, attempt to run it, and when faced with the 'Error' page, they would treat it as a web page error and immediately click the browser's 'back' button. It appeared that they did not read the details of their errors at all. When gently questioned about this behaviour, most students claimed that they did not need or want to look at the details in the 'Error' page because they felt that it was enough to examine the last line of code they had entered since their code had worked before they had added that line. This highlights an important usability issue regarding errors, which has lead us to conclude that it suffices to display only the first error in user code and allow users to manually fix that one error in the code. Novice programmers don't want to see detailed screens showing error stack traces. But they do have a strategy for coping that we can support.

All but one student claimed to use the Internet at least once a week. This point supports the decision to use the web to deploy JavanOwl through a web-browser interface. Since JavanOwl is targeted at novice programmers, who may not feel very comfortable using or installing new software, it is important that their attention is focused on learning to program, not on learning how to use the new application. Since Internet connectivity has skyrocketed in the past few years, more and more people have experienced the Internet in one way or another. This means that in general, people feel relatively comfortable using web browsers and thus JavanOwl. This was common to al the sessions and caused us to reflect on what we were seeing. For example, we did not have to explain how to use a web browser, fill in web forms, or click the 'back' button. This is why the error pages were relatively unproblematic: the students were used to clicking on web links that led to error pages: their already learned response was to ignore the detail of the error page, but instead to hit the 'back' button and re-assess what they had done.

The fact that the students would generally write one or two lines of code and would immediately attempt to run it indicates that they like the sense of immediate satisfaction and feedback provided by such a system. Even when their code worked well, they enthusiastically tinkered with the code, changing parameters, methods, and control flow, then watched the picture change: they repeated this throughout the session, experimenting more and more. This corresponds to Tanimoto's concept of 'liveness', which postulates the advantages of immediate semantic feedback automatically provided during programming [21]. Users write a bit of code and see immediately what effect that bit of code had on the state of the program. This highlights the benefits of having a split-screen system where the user can edit their code on the left and simultaneously see the output of their code on the right.

Lastly, in what was scheduled for an hour, but with formalities excluded was little more than 40 minutes, it was impressive to see that everyone did at least write programs, correct errors, see the effects, and so engage in programming.

7.2.2 Evaluation Survey Results

Evaluation surveys were distributed to all students who took part in the sessions. Overall, 31 females and 33 males filled in the surveys. The data from these surveys was collated and carefully examined . Below are some of the results.

Figure 7.1 shows a graph depicting the differences in general computer literacy between males and females. As expected, the females involved in the University Science Experience had had less prior programming experience than the males. Interestingly, more females claimed to have access to the Internet. This works well for JavanOwl because it suggests that these students have good Internet access and therefore would easily be able to use JavanOwl.

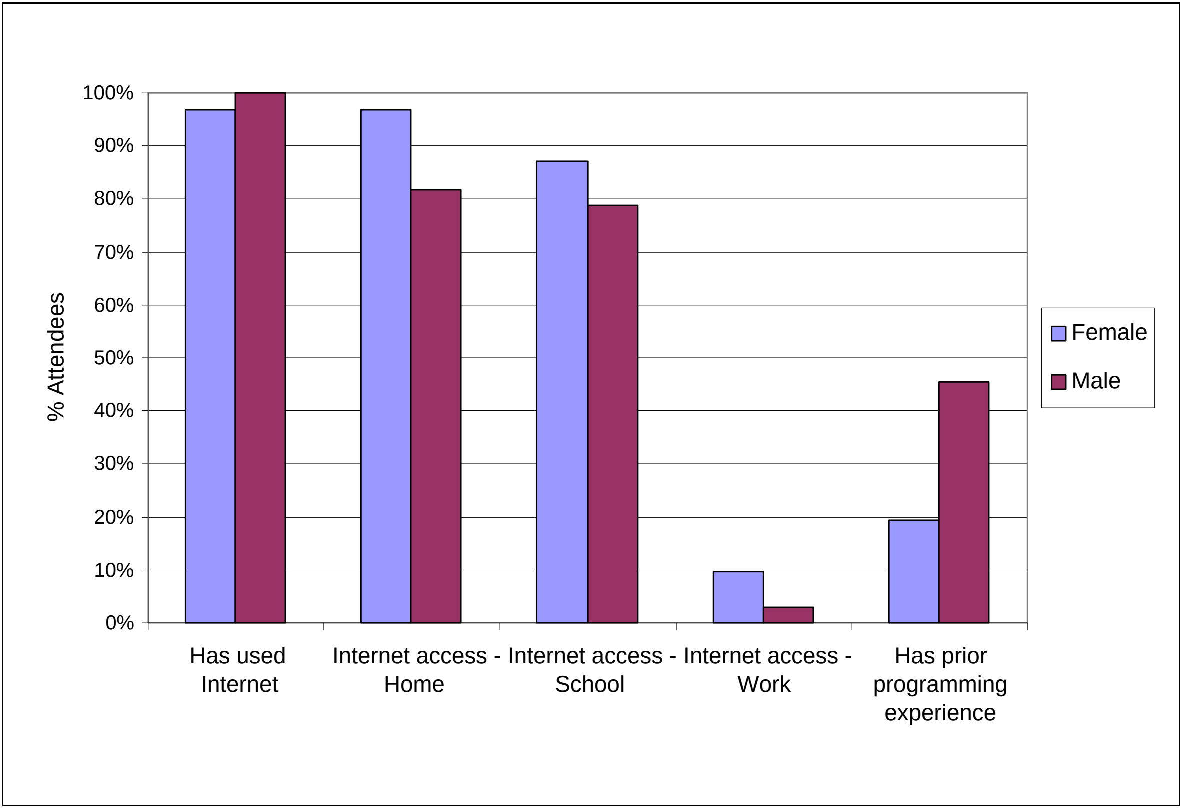


Figure 7.1: General computer Usage

Figure 7.2 shows a graph that illustrates the success rates of the exercises/tasks that the students performed during the Science Experience. The graph shows that overall, students found the tasks that were set easy to complete. Amazingly, even though the majority of these students had never programmed code before, they were able to write working code that created objects and ran methods in less than an hour and even rated these tasks as ‘easy’. Writing loops was considered to be only ‘moderately hard’ by students who attempted them. Another interesting observation is that females found all the different programming tasks only slightly harder than the males in the group. These results show that JavanOwl provides an easy environment for beginner programmers to learn how to program.

Other results show that most participants had no prior programming experience. In particular, the females involved in the Science Experience had only ever had experience writing HTML code and no actual programming languages. This again suggests that females are comfortable with the Internet and web-browsers, making JavanOwl an ideal place for them to begin learning how to program a more advanced OO programming language such as

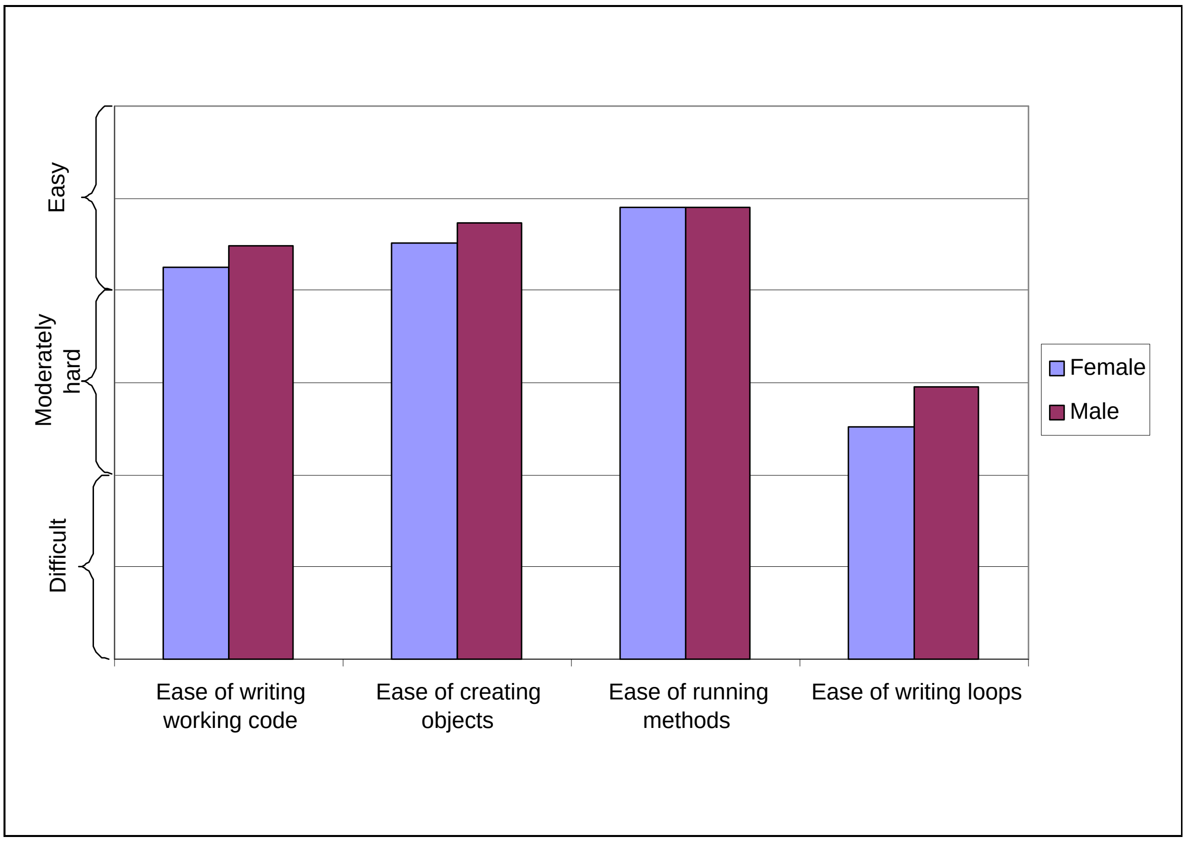


Figure 7.2: Writing code using JavanOwl

Java. Almost all students involved in the Science Experience indicated that they would use JavanOwl again if it were available, with slightly more enthusiasm from the females in the groups.

7.3 Heuristic Evaluation

Heuristic evaluation is a discount usability engineering method for a quick and easy evaluation of a user interface design, as described by Jakob Nielsen [36]. Heuristic evaluations are performed as part of an iterative design process as they highlight potential usability problems in the interface design of the system being examined. It is quick and easy — anywhere between 1 and 4 evaluators is sufficient. This type of evaluation is *formative* rather than *summative*. It is designed to highlight as many problems that can or must be addressed during later iterations in the design process as possible rather than evaluating how well the system works.

The process of performing a heuristic evaluation on an application involves 4 phases. The first step is a pre-training session during which the evaluators are given a chance to familiarise themselves with the application. The second step is the actual evaluation of the application. One testing technique commonly employed is to supply evaluators with a truly representative scenario that lists the various steps that would typically be taken in a usage of JavanOwl. During the evaluation step, the evaluators perform the tasks detailed in the scenario they are given, with or without the observer present. They are requested to make notes of the usability problems they find, keeping in mind the ten heuristic guidelines shown in Figure 7.3. The third step involves a debriefing session during which the evaluators can discuss their findings and reflect on possible solutions for the problems they found. The last step of a typical heuristic evaluation is to collate the feedback from the evaluators, compiling a list of all the usability problems that were discovered during the evaluation together with the relevant affected heuristics and the respective severities of the problems. These severities are based on a scale of 0-4, judged on the descriptions given in Figure 7.4.

Heuristic evaluations highlight specific usability problems encountered during the use of an application. The severity ratings that are attached to each problem provide the tester with a priority list of which problems are critical to application usage and which can be ignored if there is a lack of time to work on the project. By including the heuristics that are affected by each problem in the results, it enables heuristic based solutions to be found. This may result in solutions that more directly solve the issues found.

1. **Visibility of system status**
   The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

2. **Match between system and the real world**
   The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

3. **User control and freedom**
   Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

4. **Consistency and standards**
   Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

5. **Error prevention**
   Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

6. **Recognition rather than recall**
   Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

7. **Flexibility and efficiency of use**
   Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. **Aesthetic and minimalist design**
   Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

9. **Help users recognize, diagnose, and recover from errors**
   Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

10. **Help and documentation**
    Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Figure 7.3: Jakob Nielsen's Ten Usability Heuristics

0 - I don't agree that this is a usability problem at all.

1 - Cosmetic problem only: need not be fixed unless extra time is available on project.

2 - Minor problem: fixing this should be given low priority.

3 - Major problem: important to fix, should be given high priority.

4 - Usability catastrophe: imperative to fix this before product can be realeased.

Figure 7.4: Severity Ratings

7.3.1 Experiment

Four evaluators were chosen to perform a Heuristic Evaluation of JavanOwl. Each evaluator was presented with a scenario, a list of Nielsen's Ten Usability Heuristics (see Figure 7.3) and a description of the severity rating. The scenario was constructed by analysing tasks that actual users in the Victoria Science Experience performed so as to make an accurate approximation of the eventual usage of JavanOwl. Since JavanOwl provides many different features, the scenario was also designed to test every part of the system functionality in the hope of picking up as many problems as possible. Each evaluator was asked to perform the steps outlined in the scenario shown in Figure 7.5, recording any problems they came across at any stage during their usage, making particular reference to which of the ten heuristics described by Nielsen the problem pertained to. The evaluators were also asked to comment on any other problems they found that were not related to any of the ten heuristics. Until the results of the usability testing had all been completed, collected and collated, the evaluators were asked not to discuss among one another their findings. This is important as it ensures that the problems stated by each evaluator were their original feelings upon using the system, not influenced by other's comments.

Once all the evaluations were completed and returned, the results were collated. A list of all the usability problems found in the JavanOwl system was compiled. This list was presented to the evaluators, who proceeded to give each problem a severity rating. These results are discussed in the following section.

1. Create a new user account and log into the system.
2. Write a program that uses does the following:
   - Creates a Person object
   - Puts a hat on the Person object
   - Sets the hat colour to be green

   Save this program as 'first' and run.
3. Edit your program so that it now also does the following:
   - Raises the left and right arms of the Person object.
   - Creates a new Person object, of a different height and size.
   - Changes the second Person object's body colour to blue.

   Run this program.
4. Choose the Stack data structure to study. Run the code provided.
5. Edit the code provided so that it does the following:
   - Pushes "1" into the stack
   - Pushes "2" into the stack
   - Pops an element off the stack
   - Peeks at the next element
   - Pushes "3" into the stack

   Run this program.
6. Write your own class. Call it 'Greetings'. Define methods that do the following:
   - returns a string that says "hello world!"
   - outputs to the screen the line "goodbye world!"
   - outputs to the screen the word "hi" as many times as the user specifies (make it take an integer as a parameter)

   Save this class as 'Greetings'.
7. Write a program that uses the Greetings class. Make it use 2 of the 3 methods defined in the Greetings class.
   Save this program as 'usesGreetings' and run.
8. Edit your program called 'first'. Add to it so that it does the following:
   - Creates a House object
   - Puts a chimney on the House object
   - Sets the colour of the windows of the House object to blue.

   Save this program as 'second' and run.
9. Delete your program called 'first' from the system.
10. Log out of the system.

Figure 7.5: Scenario used in Heuristic Evaluation of JavanOwl

7.3.2 Results and Evaluation Summary

The table of results of the heuristic evaluation that was performed on JavanOwl is shown in Figure 7.6. The table is structured into sections corresponding to particular functions of JavanOwl since the scenario was designed to test every part of JavanOwl's functionality. The formative nature of this evaluation has highlighted several usability problems that can and should be addressed in future work with JavanOwl. However, the average severity of these problems is only 2, that is, on average, they were judged to be cosmetic problems that only needed to be fixed if extra time were available on the project. Only one usability concern with severity 4 was found: an unfriendly error page. This problem can easily be fixed and was not found to affect the actual usability of JavanOwl during the Science Experience earlier in the year. Therefore, apart from minor problems, the heuristic evaluation produced overall promising results of JavanOwl's usability.

Below is a more detailed description of the most prominent issues found within the main sections of JavanOwl's functionality.

Navigation within the JavanOwl System

Positive comments were made regarding the overall effect of the design of the interface of JavanOwl. The colour schemes and general design were complimented. However, one of the more prominent concerns regarding the facilitation of navigation in JavanOwl was the fact that the API and Forum links takes the user to a page that is hard to leave from without pressing the 'Back' button in the browser. This can cause users to feel trapped and confused. The evaluators agreed that the API and Forum links should perhaps open these pages in new browser windows. Another suggestion was to utilize frames in the JavanOwl system so that links such as these could be opened in the current browser window without forcing the user to leave their current task. Another issue that was found was that the evaluators felt that the navigation menu was not always particularly easy to see. Although the navigation bar was kept in the same location in the header of every page that it was included in, users felt it was sometimes hard to follow. The text used for the links in the navigation bar was coloured white and underlined. The white colour caused some visual problems and the fact that some other non-link text was also underlined in other places in JavanOwl caused some inconsistency concerns. Again, these types of problems would not be hard to change as they are features in the design of the HTML.

Write and Run Code Functionality

The ability to write and run code through JavanOwl is perhaps the most important feature of the JavanOwl system. Therefore, it was important to catch as many problems as possible within this section to ensure that later versions of JavanOwl are as stable and usable as possible. Evaluators felt that within this section, one of the most prominent problems was that once a user arrives at the 'Write Code' page in JavanOwl for the first time, it is unclear how they should go about writing some code. This page does not provide any template programs or classes automatically. JavanOwl's Help section does provide sample programs and classes that users can simply copy and paste into the form provided in the Write Code section. They can then save, edit and run this code. Although the evaluators did discover this after searching through the Help section, they believe it would be simpler for the user if they could just come into the Write Code section and be presented with a template program or code straight away. Another problem found in regarding the write and run code functionality was the fact that some options were available even when they most definitely did not apply. For example, the 'Run Code' button is available even when users are editting a class. Clicking on the ‘Run Code’ button in this situation would cause an error. It would be more user-friendly if these options were simply not deactivated or not visible at all to the user when they are not applicable. This type of problem can be fixed using standard HTML and JSP.

Visualisations

The evaluators had many positive remarks concerning the visualisations of program code provided by JavanOwl. They expressed the value in providing animated visualisations, as these helped show the exact effect of each method call well. The only problems that were found were regarding how much information to put into the visualisations. Adding into the visualisations the line of code that is being executed and thus causing the effect being witnessed would be beneficial to the users. Some of the terminology used in the Stack visualisations were not adequate according to the evaluators. The term ’Holding Bay’ was used to describe a ‘theoretical’ place where stack elements are held before or after `push` or `pop` methods are executed. This is not a typical term used in describing such a place and may need to change. Fortunately, since the code that is responsible for drawing these visualisations has been kept separate from the target program code, changes in the visualisations such as these would be easy to make.

Problem	Heuristic	Evaluator	Severity
REGISTRATION AND LOGIN			
When registration fails, data entered disappears and must be re-entered	1	2	3
Login not automatic after registration	7	3.4	1
Terms used in registration are not clear enough	1,2,10	1.3	2
NAVIGATION			
Navigation menu not always clear	1,6	1,4	2
API and foum should open in a different window	3,7	1,2,3,4	3
Underline texts sometimes links, sometimes not	4	2,3	3
Unnecessary section headings	3,4,8	1,2,4	1
Inconsistent naming of buttons and terms	4	1	2
HELP AND OOP CONCEPTS			
Unfriendly presentation of Help section	2	2	2
Insensitive wording in Help section	2,9,10	1,3	2
Help not helpful on what JavanOwl actually offers	1,10	1,2	3
Difference between OOP Concepts section and Help section unclear	8,10	2,3	1
Language used in Help may not be suitable for novice programmers	10	3	3
Methods available in stack description are misleading	5	1	2
Unfriendly error page	2,9	1,3	4
WRITE AND RUN CODE			
Unclear how to start coding - no template program or class shown	5,7	2,3,4	3
Empty drop down list and Open buttons available when no files or classes	1,2,5,10	1,2	2
System forgets which classes you previously used in a program	5,6,7	1,2	3
Code not automatically saved when user leaves Write code page	3,5	2,3	3
Length of wait for response increases with size of program when run	1	3	1
Object drawn off side of image doesn't create an error	5	3	3
Stack code appears in Write code section and causes problems - bug		3	3
Hard to syntax check class without writing a program that uses it	5,9	3	2
Problem with same named classes - program uses only the original class	5	3	2
Not much feedback after 'Save Code' is clicked	1	1	1
No accelerators to view help and methods available on write code page	6,7	1,2,3	2
No 'Clear' or 'New class' or 'New program' buttons	2,3,4,6,7,9	1	3
No ability to tab inside text box in browser	3,4,8	1,3	2
Option to Run Code available even when editting a class	2,5,6	1,2,3	3
API			
Some methods in API badly named (hasHat)	10	3	2
API method parameters, units not explained well (position, colours)	8,10	2,3,4	2
Methods that should not be visible to user are documented in API	5	2	3
No example code in API	10	3	2
MANAGING FILES			
Display code bug when displaying code that is a special HTML tag	1	2	2
ERROR MESSAGES ABOUT CODE			
Error messages about code not quite friendly and helpful enough	9	2,3	2
Error message sometimes misleading (missing ; points at next line)	2,5,9	2,3	3
VISUALISATION			
Unclear what the 'peek' method does in Stack visualisations	1	1	2
Use of term Holding Bay in Stack visualisation confusing	1,2,6,8	2,3,4	3
Line of code being executed not displayed in animation	2	3	2

Figure 7.6: Heuristic Evaluation of JavanOwl Table of Results

Chapter 8

Conclusions

This project has presented a study of the possible technologies that can be used to remedy the current problems associated with education in Computer Science. Online functionality and program code visualisations were found to be two key elements in the design of an effective educational programming tool. Aspect Oriented programming was explored as a program monitoring technique used to build visualisations for educational purposes. Following this, the design and implementation of JavanOwl, a web-based programming environment designed to teach Java through program code visualisations, are then introduced. A discussion of the effectiveness of the technologies used in JavanOwl were evaluated and presented. The results and observations of an actual experience of using JavanOwl to introduce Java programming to novice programmers were recounted. Finally, a heuristic evaluation was used to conduct a formative study on the usability of JavanOwl and its outcomes were discussed herein.

8.1 Contributions

- We built JavanOwl, a web-based programming tool that teaches Java using program code visualisations. JavanOwl was carefully designed to maximise the advantages of online systems while at the same time magnifying the educational benefits gained in using program code visualisations. This is a unique combination of key design elements that has not been vastly explored in the past. We successfully used JavanOwl to teach a group of approximately 70 novice programmers how to program.

- The results we have presented in this report show that JavanOwl works. The Science Experience provided qualitative evidence that students who used JavanOwl felt it was an easy and entertaining system to use. Although the majority of students who participated in the Science Experience had no prior programming experience, their overall impression was that programming is ‘easy’ to do with JavanOwl. Even the more complex programming structures such as loops were considered to be only ‘moderately hard’ to do with JavanOwl. The females in the groups fared just as well as the males, allowing them to maintain a reasonably high level of confidence when dealing with introductory Computer Science concepts. The results of a formative study of JavanOwl’s usability was performed using a heuristic evaluation of the system. This evaluation did not identify any major usability concerns.

- JavanOwl has been built using a careful balance of technologies that were found to best support the key design elements identified. JavanOwl provides support for learning a real language, Java, with no special software needing to be installed on learners’ computers. It does not impose any special requirements, apart from having a basic web browser, on the client side as the system was designed to be client side lightweight. The system therefore supports access from anywhere — school, home, Internet cafes, allowing early learners to practice their programming skills and gain a critical prior familiarity with the subject.

- JavanOwl explores the use of Aspect Oriented programming in Computer Science education — in particular, the application of AspectJ to produce visualisations of program code. The JavanOwl system also examines how an online compiler and runtime environment can be provided without any restrictions or requirements on the client side. Above all, JavanOwl is an attempt at building a new educational programming tool that incorporates both these key design elements without disadvantaging eachother.

8.2 Comparison With Related Work

In comparison to other existing educational programming tools, JavanOwl provides *both* the advantages of client side lightweight web-based systems such as ubiquitous access as well as with the benefits of animated program code visualisations. This technological combination has not been explored much in existing systems. In particular, BlueJ, LearningWorks and Jeroo, systems which provide good program code visualisations, do not function over the Internet. On the other hand, `www.publicstaticvoidmain.com` and ELP are both online systems, but they do not use program code visualisations. Jeliot is one system that is both online technology and supports program code visualisation, but its reliability on Java applets makes it client side heavyweight, imposing browser constraints and other technical requirements on the client machine. This makes JavanOwl different to the existing educational programming tools. The technologies employed in building JavanOwl make it easy to use, with a more suitable tool for the target audience.

8.3 Future Work

This project has essentially laid the groundwork for future extensions to be made to JavanOwl. The research conducted examined various approaches towards helping education in Computer Science, attempting to implement in JavanOwl those which were found the most beneficial. Further research into the topics covered in this project, namely the use of the Internet to deliver a programming environment and the use of visualisations to teach programming, may discover new techniques that can be incorporated and tested through JavanOwl. An example result of this is JavanOwl++, a programming tool that was built using JavanOwl technology. A description of JavanOwl++ is included in Appendix A. Other possible additions to the JavanOwl techology are described in the sections below.

8.3.1 JavanOwl API

JavanOwl currently has a small library of classes that users can work with. These classes all provide visualisations. This JavanOwl API was constructed as a proof-of-concept, to show that a more complete library could be also be created without too much effort. More classes to work with would provide users with a more interesting set of tools. They could experiment with more objects, gaining experience while using a more entertaining system.

8.3.2 Traces and Visualisation Library

Currently, JavanOwl creates only one type of visualisation of user code. It depicts a graphical representation of a trace of the execution of their program. A simple extension to JavanOwl would allow the user to view the trace of their program execution differently. The user could be given a choice of different visualisations to view, ranging from other graphical visualisations of the program trace or even a tree-structured textual trace.

Once aspects have been written to monitor some particular classes, creating different forms of output of trace information is simple to encode. This is because the same data, regarding calls to methods and constructors, is collected for all visualisations of traces. Therefore, all that would need to change would be how this information is manipulated and formatted for the user to view. A library of manipulations of this captured data for visualisation purposes could therefore be made. The aspects would monitor the user programs through the use of pointcuts and join points, capturing required information. The advices defined for the pointcuts would call upon a component of the visualisation library to format the data and output it a certain way depending on the user's choice of visualisation. The visualisation library to do this could be written in Java and could be entirely separate from the aspects.

This creates a degree of separation between the process of program monitoring and the process of converting this captured data into some form of visualisation. Therefore, Java developers — not necessarily proficient AspectJ programmers — could develop different visualisations or representations of the captured data from the executed program. Over time, extensive additions could be made to the current JavanOwl visualisation feature, allowing many other forms of graphical or textual representations of the output of the user's code. One useful example application would be to create UML diagrams of user programs.

8.3.3 Support for Visualistaions of User-Written Classes

JavanOwl creates visualisations of user program code by running AspectJ aspects that monitor the users program. This means, however, that the aspects that visualise library objects must be written and weaved in with the JavanOwl API by developers before the "Tomcat" web-server has been started and before the user can create such visualisations. An extension to the current JavanOwl system would allow users to create visualisations of programs they write that use their *own* classes, not necessarily the JavanOwl API.

8.3.4 Interactivity in User Programs

The programs that JavanOwl allows users to write and run are simple non-interactive Java programs. When the user clicks the 'Run Code' button, their code is processed accordingly. Only when the program execution ends is the output of the program displayed to the user. Future work could expand the types of programs that users are able to write and execute in JavanOwl. Users could create and run interactive programs that keep running, effectively simulating what Java applets can do but without the problems such as browser compatibility associated with running applets. The aim would be to continue to use visualisations with such programs. This would increase the level of 'Liveness' that JavanOwl's visualisations currently support since users would be able to change parameters and witness the changes in a visual representation of their program in real time.

Appendix A

JavanOwl++

JavanOwl++ is an application that was built for the 'Information Systems 409: Educational Technology in the Age of the Virtual University' course at Victoria University. This application was built using the JavanOwl technology. Like JavanOwl, JavanOwl++ is also a web-based system that is designed to help novice programmers learn how to program. It also uses program code visualisation to more clearly depict the effects of program code. However, it provides a much higher level introduction to programming than JavanOwl. Unlike JavanOwl, JavanOwl++ only teaches learners about object creation and object manipulation (method calls). It does not teach them about any other form of statement, iteration or selection like JavanOwl does. Also, unlike JavanOwl, JavanOwl++ does not allow the user to *write* any code. JavanOwl++ works through a point-and-click graphical user interface rather than allowing the user to write and run their own Java programs. The result of this is that learners can create programs without needing to know any Java syntax.

A.1 Overview of Functionality

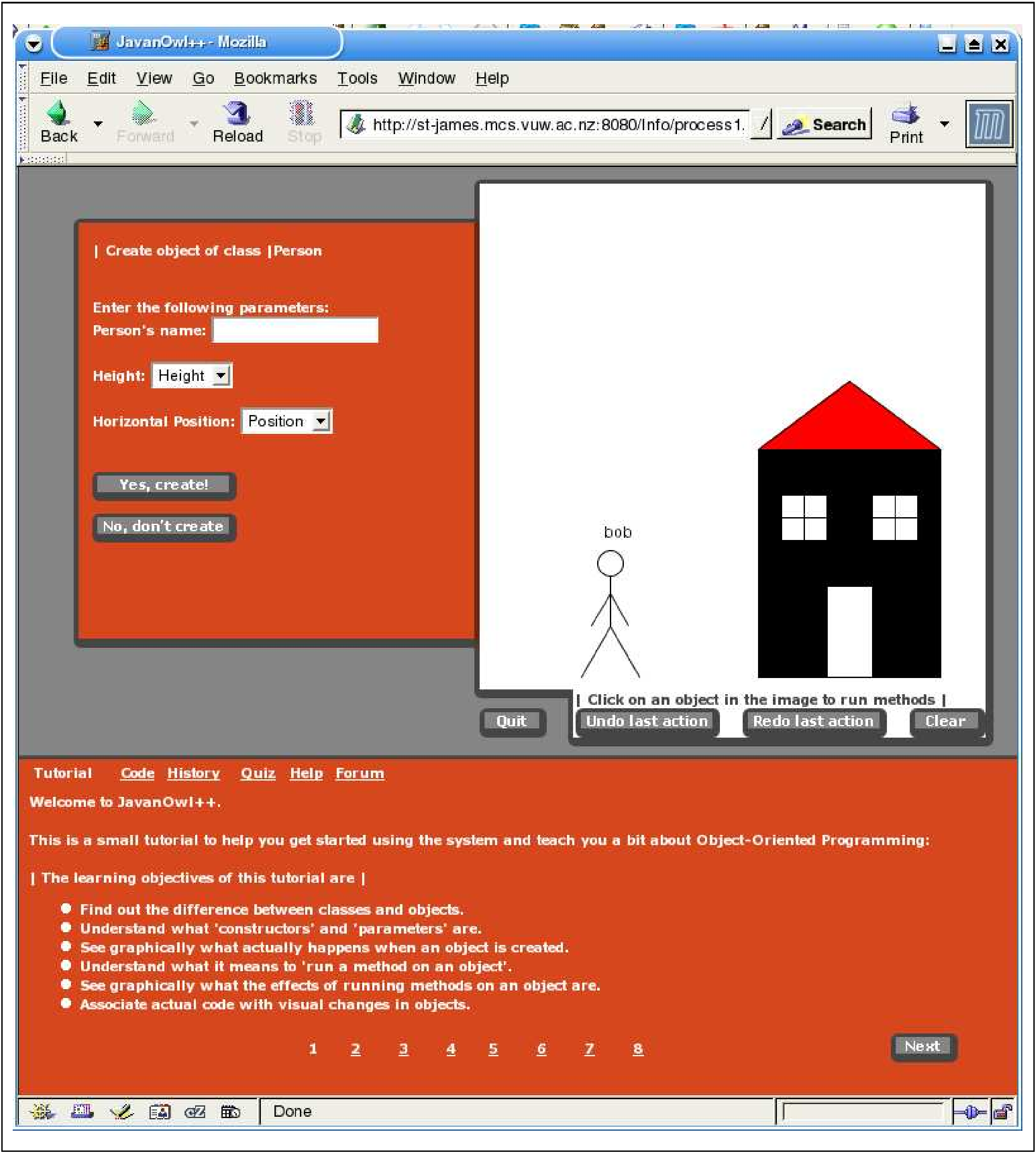
JavanOwl++ is a structured system designed for self-paced learning. A simple tutorial has been provided as a way of leading users through the system. The main application area provides users with a graphical interface through which they can create objects and run methods on them. The effect of this is that users can essentially create and run their own Java programs without actually writing any code.

A.1.1 Object Creation

Users create objects of the classes provided by clicking on the buttons labelled with class names. This then leads the user through a sequence of steps whereby the user is invited to enter or select the parameters required for the object constructors. Once the user has successfully created an object, they will automatically be shown an image of their object in what is essentially the output pane. This image is dynamically generated on the server side according to their action. Figure A.1 shows JavanOwl++ after a user has created both a Person and a House object and is in the process of creating another Person object.

A.1.2 Object Manipulation

Once the users have created an object, they can select the object by clicking on the graphical representation of their object in the image. This then provides the user with a list of methods available to be run on that object. The user selects a method, enters the required



| Create object of class |Person

Enter the following parameters:
Person's name:

Height: Height

Horizontal Position: Position

Yes, create!

No, don't create

| Click on an object in the image to run methods |

Tutorial Code History Quiz Help Forum

Welcome to JavanOwl++.

This is a small tutorial to help you get started using the system and teach you a bit about Object-Oriented Programming:

| The learning objectives of this tutorial are |

- Find out the difference between classes and objects.
- Understand what 'constructors' and 'parameters' are.
- See graphically what actually happens when an object is created.
- Understand what it means to 'run a method on an object'.
- See graphically what the effects of running methods on an object are.
- Associate actual code with visual changes in objects.

1 2 3 4 5 6 7 8 Next

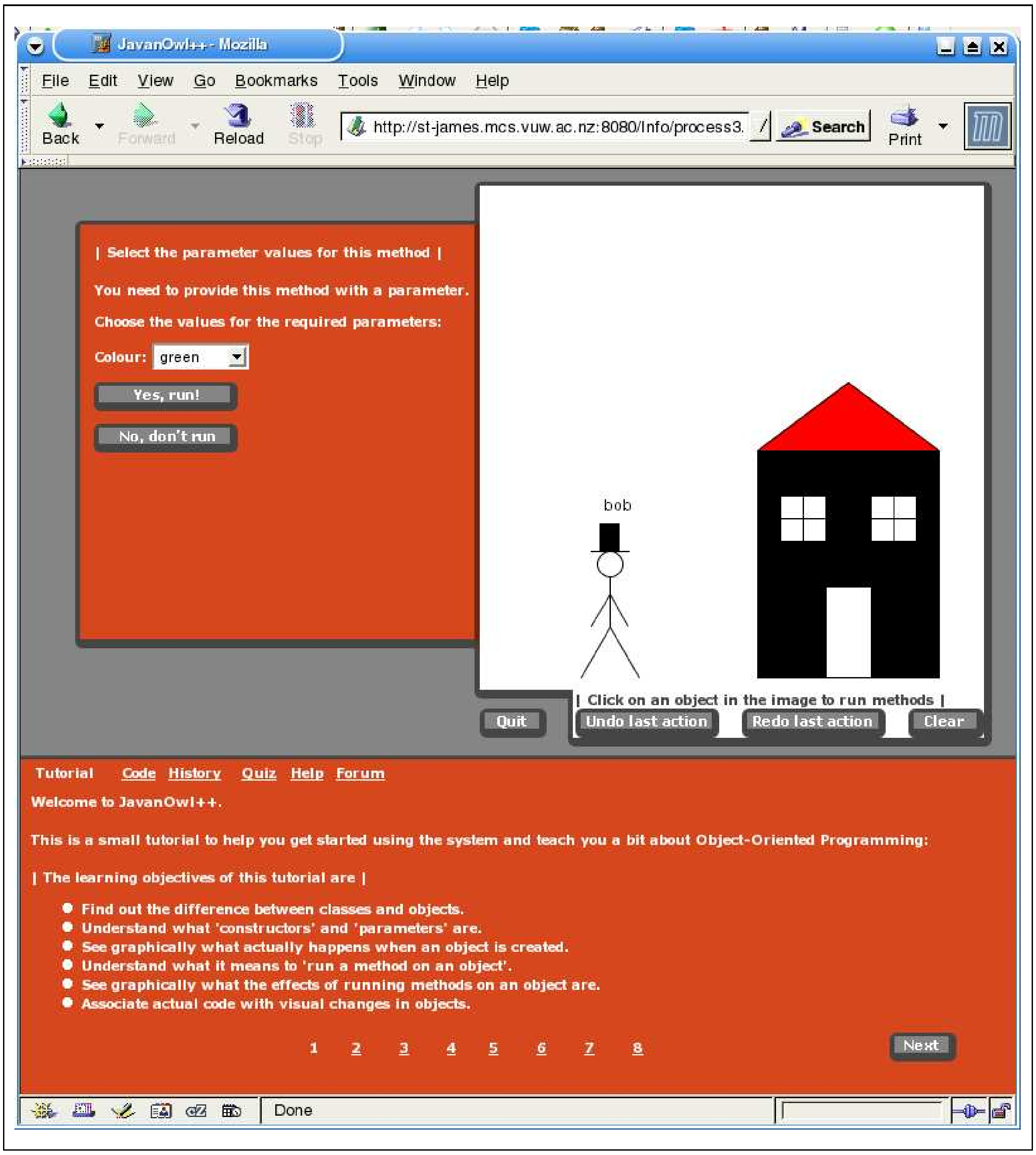
Figure A.1: Object Creation in JavanOwl++

parameters and 'runs' the method. The image is then regenerated and redisplayed, graphically showing the effect of the method just run. Figure A.2 shows JavanOwl++ after a user has run the `hasHat()` method on their Person object, and is attempting to now run the `setHatColour(...)` method on the same Person object.

A.2 How JavanOwl++ Uses JavanOwl Technology

JavanOwl++ demonstrates that the JavanOwl technology can easily be used to build other applications for different purposes. The same web-based technologies have been used in the creation of JavanOwl++: JavaServer Pages (JSP), JavaBeans, Aspect-Oriented Programming (AOP) and HTML, all used in conjunction with a JSP "Tomcat" web server. The JavanOwl++ back-end is very flexible — it will recognize and automatically allow users to use any suitable classes within the specified library. Currently, JavanOwl++ uses the JavanOwl library. The program code visualisation is a dynamically generated GIF, created in the same way to how the animated GIFs in JavanOwl were created. For the purposes of JavanOwl++ however, the visualisations are not animated. As in the JavanOwl architecture, the images are delivered to the user through a servlet. This avoids writing any image files to the server as well as any problems caused by caching.

Overall, although JavanOwl++ provides quite a different environment for learning how to program than JavanOwl does, it was built using the same technology. The result of this is that it emphasizes the power of JavanOwl's technology and shows how it can be applied directly to support other similar but different needs.



| Select the parameter values for this method |

You need to provide this method with a parameter.

Choose the values for the required parameters:

Colour: green

| Click on an object in the image to run methods |

Tutorial Code History Quiz Help Forum

Welcome to JavanOwl++.

This is a small tutorial to help you get started using the system and teach you a bit about Object-Oriented Programming:

| The learning objectives of this tutorial are |

- Find out the difference between classes and objects.
- Understand what 'constructors' and 'parameters' are.
- See graphically what actually happens when an object is created.
- Understand what it means to 'run a method on an object'.
- See graphically what the effects of running methods on an object are.
- Associate actual code with visual changes in objects.

1 2 3 4 5 6 7 8

Figure A.2: Object Manipulation in JavanOwl++

Bibliography

[1] M. Kölling, "Bluej - the interactive java environment." http://www.bluej.org/.

[2] A. Goldberg, S. T. Abell, and D. Leibs, "The learningworks development and delivery frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 78–81, 1997.

[3] D. Sanders and B. Dorn, "Jeroo: a tool for introducing object-oriented programming," in *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pp. 201–204, ACM Press, 2003.

[4] N. Truong, P. Bancroft, and P. Roe, "A web based environment for learning to program," in *Proceedings of the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Conferences in Research and Practice in Information Technology, 16* (M. J. Oudshoorn, ed.), pp. 255–264, Australian Computer Society, 2003.

[5] H. Bergsten, *JavaServer Pages*. O'Reily Associates, 2000.

[6] "Aspectj project." http://eclipse.org/aspectj/.

[7] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," *ACM SIGCSE Bulletin*, vol. 33, no. 4, pp. 125–180, 2001.

[8] M. Guzdial, "Summary: Retention rates in cs vs. institution," message posted on acm sigcse moderated members list, Georgia Tech, April 23 2002. http://listserv.acm.org/archives/wa.cgi?A2=ind0204D&L=sigcse-members&P=R112.

[9] H. Roumani, "Design guidelines for the lab component of objects-first cs1," in *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pp. 222–226, ACM Press, 2002.

[10] D. Gilligan, "An exploration of programming by demonstration in the realm of novice programming," Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand, 1998.

[11] E. Wallingford, "Toward a first course based on object-oriented patterns," in *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*, March 1996.

[12] M. Ben-Ari, N. Ragonis, and R. Ben-Bassat Levy, "A vision of visualisation in teaching object-oriented programming," in *Proceedings of Second Program Visualisation Workshop*, June 2002.

[13] J. Carrasquel, "Teaching cs1 on-line: the good, the bad, and the ugly," in *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pp. 212–216, ACM Press, 1999.

[14] M. Gunsher Sackrowitz and A. Parker Parelius, "Women in the introductory computer science courses," in *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*, March 1996.

[15] S. H. Rodger and E. L. Walker, "Activities to attract high school girls to computer science," in *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pp. 373–377, ACM Press, 1996.

[16] C. Toynbee, "Why women drop computer science." Department of Sociology and Social Work, Victoria University, Wellington, New Zealand, 1992.

[17] T. Camp, "The incredible shrinking pipeline," *Communications of the ACM*, vol. 40, no. 10, pp. 103–110, 1997.

[18] J. Brown, P. Andreae, R. Biddle, and E. Tempero, "Women in introductory computer science: experience at victoria university of wellington," in *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pp. 111–115, ACM Press, 1997.

[19] K. Brodlie, J. Wood, and H. Wright, "Scientific visualisation - some novel approaches to learning," *Integrating Technology into Computer Science Education*, June 1996.

[20] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of sofware visualization," *Journal of Visual Languages and Computing 4(3)*, pp. 211–266, 1993.

[21] M. M. Burnett, J. W. A. Jr., and Z. T. Welch, "Implementing level 4 liveness in declarative visual programming languages," in *Proceedings of the 1998 IEEE Symposium on Visual Languages*, pp. 126–133, 1998.

[22] A. Goldberg, "Building a system in virtual reality with learningworks," in *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pp. 5–9, ACM Press, 1998.

[23] R. E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1994.

[24] D. Sanders and B. Dorn, "Classroom experience with jeroo," *The Journal of Computing in Small Colleges*, vol. 18, no. 4, pp. 308–316, 2003.

[25] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Tersvirta, and P. Vanninen, "Animation of user algorithms on the web," in *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pp. 360–367, 1997.

[26] W. C. Inc., "Javascript origins / overview." http://www.woodger.ca/js_orig.htm.

[27] M. Hall, *Servlets and JavaServer Pages*. Sun Microsystems Press/Prentice Hall PTR, 1999.

[28] Sun Microsystems, *JavaBeans*. http://java.sun.com/products/javabeans/.

[29] R. Khaled, J. Noble, and R. Biddle, "InspectJ: Program monitoring for visualisation using aspectJ," in *Proceedings of the 26th Australasian Computer Science Conference* (M. Oudshoorn, ed.), (Adelaide, South Australia), Australian Computer Society, 2003.

[30] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, p. 154, 1996.

[31] R. Khaled, “Inspectj: Using aspectj for visualisation.” Honour’s Report, 2002.

[32] D. Mackay, R. Biddle, and J. Noble, “A lightweight web based case tool for UML class diagrams,” in *Proceedings of the 4th Australasian User Interface Conference, Conferences in Research and Practice in Information Technology, Vol 18* (R. Biddle and B. Thomas, eds.), (Adelaide, South Australia), Australian Computer Society, 2003.

[33] D. Leigh, “A brief history of instructional design,” 1999. http://www.pignc-ispi.com/articles/education/brief history.htm.

[34] C. McLoughlin and L. Marshall, “Scaffolding: A model for learner support in an online teaching environment,” in *Proceedings of the Teaching and Learning Forum 2000*, 2000.

[35] T. J. Project, *Security Manager*. Apache Software Foundation, 2002. http://jakarta.apache.org/tomcat/tomcat-4.0-doc/security-manager-howto.html.

[36] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1994.