CloudSpace: A Web Development Environment for CS1 Courses

Michael J. Woods

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science

Godmar Back, Co-Chair
Stephen H. Edwards, Co-Chair
Eli Tilevich, Committee Member

May 3, 2011
Blacksburg, Virginia

Keywords: CS1, Education, Web Technologies, Virtual Machines
Copyright 2011, Michael J. Woods
CloudSpace: A Web Development Environment for CS1 Courses

Michael J. Woods

Abstract

Since a massive decline of computer science graduates in 2002, computer science departments have been unable to reach previous graduation rates. In wake of this dramatic loss of graduates, researchers have been searching for the reasons students are avoiding computer science and choosing other majors.

To combat this decrease in computer science graduates, the CloudSpace environment provides additional context to entry level computer science courses. This shift in context removes boring assignments from the early computer science curriculum and replaces them with more engaging web centric assignments. The CloudSpace environment presents a model that maintains student’s focus on core computer science competencies while providing a highly simplified web development toolkit to develop feature rich AJAX web applications. This thesis includes the rational and implementation of a cloud based hosting service and a highly abstracted web tool kit that enables students to replicate modern web applications.

That this work received support from the National Science Foundation
3.3.3 Using Data Bindings ........................................ 27
3.3.4 Templates .................................................. 30
3.3.5 Event Handling ........................................... 31
3.4 Achievements ............................................... 32
  3.4.1 Feedback in a Learning Environment .................... 32
  3.4.2 CloudSpace Achievements .......................... 32
3.5 Logging ...................................................... 33
3.6 Persistence ................................................. 35
3.7 Authentication .............................................. 39
  3.7.1 Existing Authentication Methods ....................... 39
  3.7.2 Authentication in the Cloud ......................... 40
3.8 Exception Doctor ........................................... 40
  3.8.1 Examples of Common Exception Situations ............ 41

4 Teaching in the Cloud ........................................ 46
  4.1 Initial Configuration ..................................... 47
    4.1.1 Installation .......................................... 47
    4.1.2 Authentication Configuration ....................... 48
    4.1.3 Permissions ......................................... 50
  4.2 Project Composition ..................................... 51
  4.3 Teaching ZHTML .......................................... 52
  4.4 Teaching Persistence .................................... 53
List of Figures

2.1 State Transitions of an XML Request Object .................................. 5
2.2 Comet Streaming vs Comet Polling ................................................. 7
2.3 Typical Data Binding and Matching Server Side Object ......................... 13
2.4 An Example of the ZK Event Structure ............................................ 16
2.5 A Potential Class Loader Configuration that could Result in JAR Hell ..... 19
3.1 File Manager Interface ............................................................... 24
3.2 File Manager Upload Interface ...................................................... 24
3.3 An Example of the Import Tag. Both Attribute Values are Interchangeable. 27
3.4 An Example of the Variable Tag. Assumes the Person Class Has Been Im-
ported or is in the Default Package. .................................................. 28
3.5 An Example of Standard ZK Data Bindings ..................................... 29
3.6 An Example of CloudSpace Data Bindings ..................................... 30
3.7 An Example of Template Tags in CloudSpace .................................. 30
3.8 A ZHTML Page Containing a Button and In-line Event Handler ............ 31
3.9 Achievement Status Display ......................................................... 33
List of Tables

2.1 A List of Commonly Used ZUL Components . . . . . . . . . . . . . . . . . . 11

5.1 A List of Bytecode Directives and Descriptions . . . . . . . . . . . . . . . . 63
Chapter 1

Introduction

Since a massive decline of 11,000 computer science graduates since 2002, computer science departments have been unable to restore previous graduation and recruitment rates. In 2002, there were approximately 23,000 undergraduate computer science majors; however, in 2008 that number had declined to 12,000 students[18]. In wake of this dramatic loss of graduates, researchers have been searching for the reasons students are avoiding computer science and choosing other majors.

Much of the research in the area of computer science education is focused on reports that students are not finding computer science relevant to their daily lives. Studies have found that if students can relate to their course work they are more likely to be successful[10]. To add context to CS1 courses, universities have tried using robots and game design in their classes [16] [8] [17] [11]. In course offerings at Georgia Institute of Technology, a new media computation class was created to add contextual importance to computer science classes for nonmajors.

Instead of appealing to the needs of nonmajor computer science students, CloudSpace is designed to retain existing computer science students using the modern web as an assignment context. We chose the web as the context for CloudSpace because of the ubiquity of the
web to undergraduate students. Despite many CS1 students having never programmed before, most have used services such as Google, Youtube, and Facebook. The web cannot be immediately adapted to CS1 courses, the tools used to develop rich Internet applications are outside of the reach of most CS1 students. Instead, modern web tools are designed for experienced web developers with knowledge of technologies such as AJAX, database management systems, and servlet containers.

CloudSpace bridges the divide between CS1 students and modern web development tools. Our approach emphasizes the core concepts taught in CS1 courses without allowing complex web development tools to become the focus of the course. In our environment, students learn a small number of standard web technologies that can be used in future computer science courses and jobs.

The rich Internet application framework ZK is the foundation for the CloudSpace project. ZK was chosen because of its server centric design and its high level of abstraction. With ZK, CloudSpace is able to provide students an environment in which they are able to leverage technologies such as AJAX server side JAVA code execution and model view controller designs, without requiring them to understand all of the intricacies of the web.

Students create web UI’s in the CloudSpace environment using ZHTML. ZHTML is a dialect of XHTML, with added tags to provide more expressive behavior. This dialect provides them with access to AJAX tools and dynamic page generation techniques that, using traditional web tools, would require a huge investment for students to learn. Using ZHTML as the platform for UI development, students learn standard HTML tags and event handling techniques, both are skills that can be applied to future coursework.

CloudSpace supports desktop development and server side development. Students develop JAVA class files on their personal machines and test them with JUnit tests. After their classes are ready for use, they upload the class files to CloudSpace to begin UI development. CloudSpace provides a shared infrastructure for hosting their ZHTML and JAVA classes as a publicly accessible web page. As application files are modified, CloudSpace automatically
redeploys student’s web applications to host the most current version of their code. After completing their UI, the shared infrastructure hosts their projects indefinitely, allowing students to show projects to future employers and friends.

Many modern web applications contain relational databases to provide persistent functionality to user interaction. To avoid complex technologies required by relational databases, such as SQL and data schemas, CloudSpace provides students with a simple object based database for storing persistent application data. The object based database provides students with a tool to persist object data between web sessions and share application data with other students. The persistent store does not require students to define data schema’s or share their code base with other students.

Teachers are provided with a simple deployment scheme for CloudSpace that allows them to deploy their own CloudSpace server using standard deployment strategies supported by most servlet containers. In default deployment scenarios, CloudSpace is pre-configured to run without modification. However, CloudSpace is extensible to allow instructors to modify authentication and security schemes if needed.

Because student code is executing on the shared infrastructure provided by CloudSpace, a virtual machine is required to provide security, resource management, life cycle support, and isolation. The key features provided by the CloudSpace virtual machine are file system resource management through file name translation and console redirection and organization. These features allow CloudSpace to provide an environment similar to Sun’s JAVA virtual machine to execute student code.

Using surveys provided to students after completing CloudSpace assignments, data was collected about students perceived frustration and engagement with the CloudSpace environment. The surveys show that CloudSpace provides levels of engagement and frustration similar to Swing based GUI courses. Using this data, we have identified some CloudSpace features that could be improved.
Chapter 2

Background

This Chapter describes the underlying technologies on which CloudSpace is built. These include AJAX / Comet, J2EE servlet containers, ZK, and Javassist.

2.1 AJAX

Asynchronous JAVAScript and XML, or AJAX[1] is a technique used for communicating data between a client’s browser and a web server. AJAX technologies allow a communication channel to be created between the server and browser. This communication channel is used to communicate user interaction events with the browser to the server and updated UI values to be sent from the server to the browser. This communication channel allows UI logic to be distributed between the client and the web server[1].

AJAX requests are sent by embedded scripts within a HTML page. These scripts can be implemented using a variety of different libraries; however, they are always implemented with JAVAScript and the built in XML request libraries. Some of the earliest implementations of AJAX techniques were in Internet Explorers ActiveX. However, JAVAScript AJAX has become more widely used since its use in web applications such as Google Mail and Google
When AJAX requests are implemented, the first step is to create an XMLHttpRequestObject. Each object created manages the life cycle of the request. The XMLHttpRequestObject interface defines five states for the request object including unsent, opened, received headers, loading, and done. Figure 2.1 shows the life cycle of each of the request objects [7].

Browsers support sending AJAX requests as GET or POST HTTP requests. As a performance boost, browsers cache recently accessed pages for faster load times. To bypass web caching, additional parameters may be added to the end of the HTTP GET or POST request to make each request unique. HTTP GET requests are used when the data sent to the server is small. Conversely, HTTP POST requests are used to send large amounts of data to the server.
After the XMLHttpRequestObject has been created, the request is sent to the server for computation. The client registers a callback function with the XMLHttpRequestObject. This callback will be executed when the XMLResponseObject enters the loading state. The script does not wait on the result; instead the script continues executing and depends on a callback to process the server response.

After the server has processed the client request, it formulates a response document for the client. When received by the browser, the registered callback function is executed. The client can then use JAVAScript libraries to read the response as a text or as hierarchy of XML components. Depending on the content of the response, the data is manipulated by the JAVAScript callback function and used to modify the UI displayed in the browser.

The AJAX techniques so far in this section described AJAX communication limited to clients requesting information from server. However, in many cases the server might be interested in contacting the client. To achieve communication from the server to client, a technique called Comet or AJAX Push can be deployed.

There are two common techniques used to implement Comet. The first, Comet streaming, opens a persistent HTTP request channel and handles messages from the server incrementally. Comet streaming is not supported by all browser JAVAScript libraries.

The second technique, Comet polling, is typically used to implement Comet because it’s implementation is supported by all modern browsers. In Comet polling, the client initiates a regular or empty AJAX request to the server which is kept open until the server is prepared to send the client data. After data is sent to the client, a new AJAX request is created and the process starts over again. Figure 2.2 shows a summary of the two techniques.
Figure 2.2: Comet Streaming vs Comet Polling
2.2 J2EE Servlets

To manage and execute JAVA code on the server, Sun, the creator of JAVA, designed a standard set of interfaces for classes to act as a web server. Together, these classes are executed by a servlet container such as Tomcat or Glass Fish. The JAVA standard defines servlets and filters. A servlet class is responsible for interpreting the URL requested by the client and creating a file to send back. Filter classes are mapped to URLs in the servlet container's configuration files. These filters are responsible for performing preprocessing steps before the servlet is handed the request. For example, filters are responsible for redirecting requests and performing certain authentication checks.

Many servlet containers exist; however, in this document we will focus on Tomcat. Tomcat is a highly configurable servlet container that consists of two components important to the execution of CloudSpace: Coyote and Catalina. The Coyote component is the HTTP connector packaged with Tomcat. This component is responsible for listening for web requests, processing them, and sending a response to the client. As a middleman between Coyote and CloudSpace, the Catalina component is responsible for passing parsed requests to mapped filters and servlets.

Inside of the Catalina component, multiple web applications can be defined. Each web application consists of a configuration file and a set of resources used to serve HTTP requests. Each web application has its own JAVA class loader that delegates the loading of all JAVA utility packages to a shared system wide class loader. Each web application shares server side log files, security configurations, file system access, and console output. The security configuration used is configured at the JAVA virtual machine level. The virtual machine is capable of distinguishing between different web applications running within the virtual machine if configured ahead of time. The JAVA security manager is responsible for protecting class loaders, threads, JAVA virtual machine execution, controlling file system access, preventing unauthorized access to system resources (printers, copy and paste buffers, etc.), controlling network access, and controlling access to specified packages.
2.3 ZK

At the core of CloudSpace is the ZK rich Internet application framework. This framework is responsible for generating HTML pages, handling all AJAX communication, and managing all server side JAVA objects. The ZK framework as described in this document is at version 5.0. CloudSpace uses the open source LGPL version of ZK also known as ZK CE.

ZK is a server centric web framework that depends on web technologies such as AJAX, HTML, and CSS. To design the view that will be displayed to the user, ZK developers define ZHTML pages. ZHTML pages are a dialect of XHTML that includes additional tags for rendering complex components such as listboxes and grids. These pages are parsed when an initial request is received from a client. The parsed ZHTML page is instantiated as a server side document object model. The server side document object model serves as a container to synchronize all server side data with the browser DOM. When events associated with the ZHTML page are received by the server, the server side DOM is used to identify event handlers that should be evaluated as an event response. From this server side document object model, ZK generates an HTML page containing standard HTML tags, JAVAScript expressions, and CSS style sheets. The generated HTML pages can be rendered by any browser without needing specialized plug-ins.

ZK UI Components

Every ZHTML page consists of tags and attributes. Tags, or components, are typical HTML tags such as the body of the document, `<body>`/`</body>`, or a line break, `<br/>`. When parsed by ZK, these tags are converted to server side objects. These objects are simple JAVA objects that contain all of the attributes associated with the tag and provide published methods to modify its attributes. Each of the tags may have attributes associated with them. For example, an HTML tag may have a style attribute, `<p style="align: center;">`. When tag attributes are parsed, setters and getters on the server side component objects are called.
When the page is requested, ZK queries each of the server side tag objects for a pure HTML version of themselves. Each of the tags pushes HTML tags onto an output stream to be rendered for the client. For more complex tags, ZK renders JAVASCRIPT functions to generate the HTML in the browser.

ZK also provides a set of ZUL tags to developers. ZUL tags are a subset of the Mozilla XUL component set developed for the Mozilla platform to aid developers in designing UI components. The XUL components typically have a standard set of attributes that define event actions such as mouse clicks, style settings such as height and width, and a unique identifier for the component. When the HTML and JAVASCRIPT for client side rendering is created, each of the XUL tags are traversed in the document object model and converted into HTML. Table 2.1 contains a subsection of the more commonly used ZUL tags.

ZUL tags are parsed and stored server side in the same way that ZHTML pages are. The ZUL parsing algorithm is less forgiving than the ZHTML parsing algorithm. In ZHTML pages, unknown attributes and tags are ignored in the parsing stage. ZUL pages, however, will throw an exception if any of the tags within the page are unknown to server side objects.

In addition to the ZUL and ZHTML tags, ZK allows for embedded scripts in the pages using <zscript> tags. These <zscript> tags allow the developer to embed code in languages such as JAVA, RUBY, and JAVASCRIPT directly into the ZUL or ZHTML pages. In the case of embedded JAVA, the code is interpreted by a bean shell interpreter [2]. The bean shell interprets embedded JAVA code and stores generated JAVA objects associated with the ZHTML pages in a special bean shell scope. These JAVA objects are visible throughout the ZHTML page in other embedded scripts.

Client Server Synchronization

After the client has rendered the ZK page, the web application enters a cycle of user input events and server side DOM synchronization. Each component on the page has certain events
<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;button&gt;</td>
<td>The button tag creates a button on the page without requiring the developer to specify a style sheet. The button tag also allows the developer to hook into the click event on a button and request code to be executed on the server.</td>
</tr>
<tr>
<td>&lt;listbox&gt;</td>
<td>The listbox tag allows developers to specify tables quickly. It provides sub-tags such as &lt;listitem&gt; to easily denote where column entries go. The listbox tag also allows for template setup where an array or other iterable collection is specified and iterated over to populate a listbox.</td>
</tr>
<tr>
<td>&lt;borderlayout&gt;</td>
<td>The borderlayout tag allows developers to specify a layout for all components contained within the layout tags. The borderlayout tag supports sub tags such as &lt;center&gt; and &lt;bottom&gt; to specify where on the page components should be rendered.</td>
</tr>
<tr>
<td>&lt;checkbox&gt;</td>
<td>The checkbox tag is similar to the button tag. This tag draws a checkbox on the page and allows the developer to hook into the click action of the component.</td>
</tr>
<tr>
<td>&lt;combobox&gt;</td>
<td>A combobox tag provides developers with &lt;comboitem&gt; tags to specify items that should be listed in a pull down combobox. The combobox also allows the developer to hook into a click event occurring within the combobox.</td>
</tr>
<tr>
<td>&lt;html&gt;</td>
<td>The HTML tag allows for HTML content to be embedded directly into the page without rendering. This is helpful to speed up the parsing step on ZK because it does not create server side instances of the contained tags. It also allows a developer to quickly embed additional HTML at run time into the page.</td>
</tr>
<tr>
<td>&lt;tabbox&gt;</td>
<td>The tabbox tag is a highly complex tag that allows for an entire tabbed UI to be embedded into the page. The embedded tabbox contains a list of tabs at the top of the page and full HTML pages associated with each tab.</td>
</tr>
</tbody>
</table>

Table 2.1: A List of Commonly Used ZUL Components
associated with it. For example, a text box may fire an event when its contents are modified and a button may fire an event when it has been clicked.

When designing Internet applications, developers generally utilize a model view controller design. In this design, the model stores the raw data to run the application, the view describes the UI presented to the user, and the controller provides the links between the view and the model. Generally, the controller is implemented as a class containing callbacks to be executed when the view has fired an event. ZK, however, provides mechanisms for defining hooks within the view and automatically generating a controller based on these hooks.

ZK provides two methods for defining links between the view and the application’s model. The simplest method for event handling is embedding JAVA statements in special event handling attribute blocks. These JAVA statements are executed using the same bean shell interpreter as the \texttt{<zscript>} tags.

The second mechanism is ZK’s data binding technology. Data bindings allow the developer to create synchronization points between particular tags and server side JAVA objects. Data bindings are defined as annotated values inside of an XML tag attribute. Data binding annotations begin with \texttt{@} and end with \texttt{}}. Contained within the annotation brackets are JAVA bean statements to be bound to the attribute.

JAVA bean syntax is broken into two parts. The first part is the bean name. Bean variables are recovered from the pages bean scope. This bean scope contains bean variables for all of the components on the page and any variables defined in the \texttt{<zscript>} blocks. The bean variables are followed by a bean attribute.

JAVA bean attributes are a set of fields defined within the bean variable or accessible through getters or setters that allow modification of the fields within the JAVA bean. Using JAVA bean syntax to bind JAVA objects to attributes in ZHTML tags allows for two way flow of information with a single statement. When the data bindings are interpreted, they are automatically translated to a getter or setter method depending on the direction of the
flow. For example, a bound JAVA bean that references `fieldName` will be converted into a `getFieldName()` call if the attributes data is being updated or `setFieldName(Object o)` if the model is being updated. Once an attribute on a XML tag is bound, the developer may specify when the bindings should be updated. For example, if the client updates the value of a text box the developer can configure ZK to update the value of a button label. Figure 2.3 shows an example of bean syntax and matching server side JAVA object.

The updates to each of the components are handled entirely by ZK. When the server responds to an event on the client, the client’s browser contains JAVAScript for interpreting the results. This JAVAScript inspects the server response and manipulates the tags on the page to update their values.
ZK Framework

After understanding the API side of ZK, we can discuss the internal framework that powers ZK. A ZK application consists of a set of user defined ZHTML or ZUL pages and a set of user defined classes. The ZUL/ZHTML pages are stored on the server in their unprocessed form. As requests are received, the pages are processed and converted into a set of JAVA objects on the server. The conversion of tags to JAVA objects is performed using a language definition file. The language definition is loaded when ZK is initialized and contains all of the tags included for parsing.

In addition to parsing the document and creating a server side document tree, some per request objects are initialized for the request. Every ZK session is provided with a unique identifier. This identifier allows ZK to receive AJAX requests from clients and de-multiplex them for processing using the appropriate document tree. The identifier and associated session variables are stored in ZK’s desktop object.

All ZK code executes as compiled code at run time. However, code executing as implicit controller code defined in ZHTML pages execute as a mix of compiled JAVA code and dynamically interpreted JAVA code. After the page is parsed, each ZK script block is evaluated at run time. The run time evaluation is performed by the bean shell interpreter. The bean shell interpreter is a separate subsystem for parsing legal JAVA code, looking up class definitions, and using JAVA reflection libraries to execute the statements. By using JAVA reflection libraries, the developer code the bean shell interpreter is executing will be compiled JAVA code managed by the JAVA virtual machine.

Not all script code executes in the same bean shell scope. ZK provides multiple scopes for bean shell execution to prevent variable definitions from leaking between pages. Bean shell scope is also not limited to bean shell defined variables, ZK’s modified bean shell interpreter also allows for ZK UI tags to be referenced within these bean shell expressions. This allows for modification of UI internals such as visibility and style from within these \texttt{<zscript>} blocks.
ZK Events

Each of the UI components has hooks available for the events they can trigger. To hook into the UI component’s event, the developer uses an `<attribute>` tag and embeds a snippet of JAVA code to be executed when the event occurs. When the event is triggered, the code is executed using the same bean shell interpreter as the `<zscript>` blocks processed before it. The same scoping rules apply to these attribute blocks.

To signal the server that a client side event has occurred, an AJAX request is sent to the server. This request contains information about the component that triggered the event, the event triggered, the desktop currently executing, and other supplemental data. When the server receives the request, it matches it to the proper desktop and component within the desktop’s document tree. The component is then able to retrieve the event attribute code and execute it using the bean shell interpreter. The bean shell interpreter may modify any of the components within the document tree. If the document tree has been modified, ZK creates an AJAX response containing instructions for how to render all of the modified components. When received by the client, the response is parsed and the client side browser is updated. Figure 2.4 shows an example of the typical client server event loop.

It is sometimes required that the server sends updates to the client without a preceding client side event. To accommodate this, ZK provides an implementation of AJAX long polling. In this scheme, the client sends a dummy event to the server every second. When the server receives the event, the response is attached to the list of responses and sent to the browser. This system uses the exact same Ajax request and server response loop as normal ZK events. The dummy events are created by the client to provide the server with a constant stream of events which to respond.
Figure 2.4: An Example of the ZK Event Structure
2.4 **Spring Security**

CloudSpace uses the Spring Security Framework to authenticate and authorize users. Spring Security can be reconfigured to provide authentication schemes such as cookie and HTTP basic authentication into web applications with few modifications. Spring Security also provides authorization levels for authenticated users. These authorization levels allow the developer to restrict access to different areas of their web application.

Spring Security integrates into a web application through the use of standard J2EE Servlet filters. Using Spring Security configuration files, a set of filters can be configured to intercept web requests being processed by the web application. Using regular expressions, which are applied to incoming URL’s, Spring Security allows the developer to require authentication for particular areas of the web application. Spring Security also automates the process of obtaining credentials from the user. This automation includes prompting the user for user name and password and automatically attempting to authenticate the user against pre-configured authentication sources. After authentication, Spring Security creates a user object to represent the authenticated user. This user object contains all of the user’s authentication and authorization credentials.

2.5 **Class Loaders**

When the JAVA run time environment is executing a JAVA application, each JAVA class referenced is loaded lazily by the JAVA virtual machine at run time by a class loader. In JAVA, class loaders are responsible for searching the JAVA virtual machine’s class path for classes based on specific naming conventions. If a class is found with a matching name, the class loader will retrieve the class’s byte code and load the class into memory.

The default JAVA class loader hierarchy is split into three class loaders. The first class loader is the bootstrap class loader. This loader is responsible for loading all core JAVA libraries.
The second class loader is the extensions class loader. Extensions classes are trusted classes that are optional to the execution of the JAVA run time environment. The last level of class loaders is the application class loader. This class loader is responsible for loading application specific classes and libraries. These classes include all classes defined in the class path when the virtual machine is started.

Except for the bootstrap loader, class loaders attempt to delegate the loading of classes to the parent class loader. For instance, a request for `java.util.List` will be requested from the application class loader. The system class loader will delegate the loading to the extensions class loader who will delegate the loading to the bootstrap loader. The bootstrap loader will find the class and return it. If the class was not present in any of the parent loaders, the application loader will attempt to load the class. If the class definition is not found, a run time exception will occur.

In addition to these standard class loaders, developers can add their own custom class loaders to the hierarchy. If a custom loader is used to load a starter class, all subsequent classes being loaded as a result of the starter class’s execution will be loaded by the custom class loader. Care must be taken when using custom class loaders; however, because classes loaded by a different class loader are not assignment-compatible even if they have the same name. The difficulties resulting from these loader hierarchies are referred to as JAR hell [4].

Figure 2.5 shows a potential class loader hierarchy that could result in JAR hell. In this arrangement, a class loaded in loader 1 cannot be referenced or used by an object created using a class in loader 2. For example, if a Foo object was created using a class definition loaded by loader one. Similarly, a Bar object was created using a class loaded by loader two. If the Bar object attempts to refer to the Foo object as a Foo class, a class cast exception will occur. The only way for the Bar object to manipulate the Foo object instance would be to require the Foo object to be a subclass of a type whose definition is shared by both class loaders. This way, the Bar class can manipulate the Foo class using the shared interface.
Figure 2.5: A Potential Class Loader Configuration that could Result in JAR Hell
2.6 Javassist

JAVA code is compiled into architecture independent byte code. This byte code is translated to native code at run time by the JAVA virtual machine.

Javassist is a byte code engineering library that allows developers to modify and dynamically generate classes before they are loaded by the JVM. The Javassist ClassPool contains all of the classes available to be loaded by the application’s class loaders. When a developer wishes to modify a class, they simply query the class pool for a CtClass object from its records. After obtaining a CtClass object, the developer can access the class’s members and fields using method calls similar to JAVA’s reflection libraries.

In addition to modifying byte code statically, Javassist also allows developers to inject a new class loader into the class loading hierarchy and rewrite classes before they are loaded. When the Javassist class loader is used, it provides a hook method to execute code before any class is loaded for the application. At this point, the developer can use other rewriting tools to modify the class before it is loaded.

When rewriting steps are clear, the API can be used to perform byte code modifications. Sometimes a class file must be scanned and inspected for potential points to rewrite. Javassist’s ExpressionEditor class scans a CtClass file looking for byte-code expressions to modify. These expressions include exception handlers, method calls, field access, and other standard JAVA expressions. The ExpressionEditor then provides hooks to the developer to inspect each JAVA expression and rewrite the expression at their discretion.

The byte code engineering library also allows users to generate entire classes on the fly. This dynamic class generation allows Javassist users to create byte code definitions of classes and make them available through the JAVA class loader with other classes in the class path. These dynamic classes are helpful for use with JAVA statements rewritten at run time, because they allow the compilation of new JAVA statements after the dynamically generated classes are available in the class loader.
Chapter 3

Students in the Cloud

When students are working on projects for their CS1 courses, they need to focus on core computer science skills such as defining classes, implementing method calls, and traversing data structures. The CloudSpace environment provides students with access to web technologies that would ordinarily be considered outside the scope of a CS1 course without shifting the focus away from these core competencies.

Developing web applications is typically outside the scope of CS1 courses because of its dependence on concepts such as distributed computing, synchronization, and huge depth of technological understanding. Although, it is impossible to completely remove exposure to these concepts, CloudSpace attempts to hide implementation details from the users. CloudSpace presents students with a centralized view of the web that mimics local application development.

Students develop CloudSpace applications initially on their local machine. They then upload their compiled JAVA classes to CloudSpace, to develop rich user interfaces for their JAVA application. The following chapter describes the student’s view of the CloudSpace development process emphasizing the technologies provided to them and development paradigms they adhere to.
3.1 Local Development

The first stage in student development is the creation of simple JAVA classes on their local machines. This step is completely independent from CloudSpace. Students use a provided student library devoid of any ZK specific or UI classes. Students make calls to persistence libraries, application authentication requests, and other web utilities. These libraries also provide students with access to cloud related functionality using a emulated environment.

The JAVA code developed by students consist of simple data structures, getters, setters, and logic to manipulate the contained objects, which forms the core subject matter of their CS1 course. The student’s code is accompanied by a suite of JUnit tests, which are developed by students to exercise their code before uploading it to CloudSpace. Typically, the JUnit tests will mimic the operations that students expect to execute on CloudSpace.

The JUnit tests developed by students are intended to mimic the transactional nature of web applications. JUnit tests execute small portions of code in an isolated manner. Using the CloudSpace libraries provided to them, they are able to test the results of their code independently through API interfaces. For example, if a student executes a method that stores an object in a CloudSpace web application session level store, the student can use CloudSpace’s libraries directly within their unit tests to test if the object has been stored properly. This decoupled testing reflects the non-sequential nature of their code’s execution on CloudSpace.

In Virginia Tech’s CS 1114 class, student’s JUnit tests are expected to fully cover their JAVA classes. An automated grading system will run a battery of tests on their JAVA classes and assign a partial grade to the students accordingly. Using this system, students need to understand only JAVA fundamentals. This allows an instructor to separate the web development and JAVA development aspect of their grade.
3.2 File Manager

Previously, CloudSpace provided access to remote directories with a separate web application that only managed uploading and downloading files. When we began to improve CloudSpace, we decided that file system access is an integral aspect of the CloudSpace development process and should be integrated into the CloudSpace infrastructure using a web application hosted in the same web application as the student’s code. This web interface closely mimics the user interface used to access files in their local development environment.

When students have thoroughly tested their classes locally, they upload their compiled class files to the cloud. They log into CloudSpace using either university credentials or provided CloudSpace credentials and are immediately directed to their CloudSpace control panel. This control panel includes a file manager, achievements tab, and user guide tab.

In the file manager shown in figure 3.1, students have access to specific directories on the CloudSpace server. These directories are organized into group and private directories. Private directories are used for individual projects and homework assignments and are only accessible to a single student. Group directories, can be used to create group projects and provide instructor sample code to students.

Each directory supports permissions, which may allow a student to read, write, or read and write the contents of the folder. Write permissions are granted for creating new files, directories, specialized project directories, and compiling JAVA source code. Specialized project directories allow students to control the initialization of their web application. Currently, CloudSpace only supports the configuration of the project’s classpath. The server side compilation feature can be used to correct minor bugs instead of uploading new class files. Figure 3.2 shows the upload window in the file manager. The upload window allows students to upload any resources they need for their project. This includes JAR files, images, and source code.

The file manager provides hooks for additional CloudSpace features. Using these hooks, we
Figure 3.1: File Manager Interface

Figure 3.2: File Manager Upload Interface
have begun adding new tools to aid in the student’s development process. These include project a submission feature, user help guides, and server side compilation. In the future, we plan on implementing new features in the file manager framework. These include a system for providing TA and instructor help to students and XHTML validation.

3.3 ZHTML Development

Once students have uploaded their class files to CloudSpace, they begin crafting the user interface that will manipulate their JAVA objects. The CloudSpace UI development process begins with the definition of a ZHTML page. ZHTML pages include a layout tags and server side object bindings.

3.3.1 UI Philosophy

One of the most common UI architectures used in web development is the Model View Controller design[5]. In this design, the web application is separated into three distinct sections. The first section, the model, is responsible for all transactions with the data stored in the program. The view is responsible for displaying all of the information stored within the model. In the case of a web application, the view is an HTML page rendered in a browser. Typically server side scripts are used to generate the HTML. The final aspect of the design is the controller. The controller acts as an intermediate that converts input and interaction from the view into data to be manipulated or stored in the model.

In CloudSpace, we attempt to preserve the Model View Controller design without requiring students to understand how it works. The model running on CloudSpace is developed on their local machine as described in Section 3.1. The next section will describe the development of the view portion of the web application. The controller; however, is hidden and automatically generated for students. While they place snippets of code in their view to help CloudSpace
generate a controller for them, they never need to understand how the data is given to the model or how to design their own Model View Controller infrastructure. Still, this exposure to Model View Controller design is helpful in future classes where advanced topics may require learning MVC, Model View Controller, layouts.

3.3.2 UI Development

When defining ZHTML pages on CloudSpace, students use standard HTML tags when possible. Knowledge of HTML tags provides students with web development skills that can be transferred to future course work. However, to express more complex UI components such as lists and tables, students are taught a limited subset of ZUL tags. These ZUL tags include lists, grids, buttons, and comboboxes. Students who wish to expand their knowledge of ZUL tags are provided ZK’s development guides. CloudSpace has been designed to allow students to mix both HTML and ZUL tags in the same ZHTML document.

In the standard ZK environment, the ZUL component set must be explicitly included in the ZHTML language through the use of XML namespace prefixes. To reduce confusion about when to use namespace prefixes and to allow us to change the implementation of any tag the student is using, CloudSpace includes the entire ZUL component set in the ZHTML language, without requiring the use of XML namespace prefixes. Unlike the standard ZUL parser, CloudSpace’s ZHTML parser tolerates mistakes in ZHTML pages such as undefined tags and attributes, and still renders the ZHTML page partially.

In the UI development phase, students use the CloudSpace server side file editor or an IDE based text editor to create their ZHTML page. Using the CloudSpace editor allows instant verification of the ZHTML page on the server. We are also considering providing students with local IDE support for developing their ZHTML pages.

The ZK framework provides a plug-in for the Eclipse IDE to aid ZK developers in the creation of their ZHTML pages. The plug-in assists developers with including ZHTML tags
**Import Tag**

<import class="java.io.File" />

OR

<import classes="java.io.*"/>

Figure 3.3: An Example of the Import Tag. Both Attribute Values are Interchangeable.

and identifies which attributes are available for each tag. The ZHTML editor also allows for the ZHTML page to be auto-formatted. The auto-format feature is helpful for ZHTML development, because students commonly miss ending ZHTML tags. Using the auto-format functionality, students can format their document and easily identify syntactically incorrect ZHTML.

### 3.3.3 Using Data Bindings

To associate JAVA objects with ZHTML pages, CloudSpace provides students with a set of custom ZUL tags. These tags augment the `<zscript>` tag standard ZK provides. Unlike `<zscript>` tags, CloudSpace’s custom tags provide a single global scope. Using the custom ZK tags also allow CloudSpace to provide more specific error messages when students make mistakes, as compared to the generic exceptions that would be thrown when using the `<zscript>` tag.

The first ZHTML tag is an `<import>` tag. This tag allows students to import packages into the ZHTML page. The import tag supports only one attribute. This attribute allows for a class or wild card package to be imported into the ZHTML page for use in declaring variable. Figure 3.3 shows an example of the `<import>` tag.

The `<variable>` tag allows students to construct JAVA objects to be associated with a ZHTML page. The `<variable>` tag allows students to define the type, name, and value of a object which is then available for use in all JAVA statements in the ZHTML page (i.e., ZK.
event handler code). Figure 3.4 shows an example of the `<variable>` tag.

In traditional web development, a controller commands the client side HTML and the JAVA objects living on the server. CloudSpace students use ZK’s data binding feature to define points within their ZHTML page that are bound to values stored in their server side JAVA objects. These data bindings allow students to bind their objects as a data sink, data source, or both.

The type of binding is determined by the attributes that replace the object and the getters and setters available in the object. For example, a ZUL label tag with the value attribute annotated with a data binding will result in a call to one of the bound JAVA object’s getters. The result of the "get" method call is displayed by the ZUL label. However, textbox elements will be bound to both a getter and setter. This binding will populate the textbox with the initial value of the JAVA object retrieved and any updates to the textbox will be propagated to the JAVA object. One of the uses of these data binding’s is binding a JAVA object to the visibility attribute of a ZHTML component.

The flow of data between the client and server is automated for the student. Unlike standard ZK, students do not need to indicate when to trigger binding synchronizations. Updates to a server side object push the new data to the client the moment they occur. That way, a student can manipulate the JAVA object or browser object without understanding the distributed nature of their application.

The ZK data binding engine requires that all bound expressions use bean shell syntax, which is slightly different than the JAVA expression syntax students are learning in their class. CloudSpace implements the data binding framework to allow students to use regular

**Variable Tag**

```xml
<variable type="Person" name="user" value="new Person()"/>
```

Figure 3.4: An Example of the Variable Tag. Assumes the Person Class Has Been Imported or is in the Default Package.
**Standard ZK Bindings:**

```xml
<variable type="Person" name="user" value="new Person()"/>
<label value="@{user.name}"/>
<textarea value="@{user.status}" />
```

Figure 3.5: An Example of Standard ZK Data Bindings

JAVA expressions in the data bindings.

The use of full JAVA expressions in data bindings is simple and sufficient for one way bindings. For example, a ZHTML page that displays the data contained in a data object requires only a getter to function. However, JAVA expression data bindings become more complicated when the data binding needs to process user data contained in the browser and push it into a server side object. In this case, students can either use the traditional bean shell syntax or use special variables introduced by CloudSpace. CloudSpace allows students to refer to the existing value of an attribute tag using a "$$" special variable, which is replaced with the value of the attribute.

Figure 3.5 shows examples of the original ZK style of data bindings. In this figure, a Person object is bound to the value attribute of a label and textarea. These bindings use bean shell syntax. Figure 3.6 shows the new bindings introduced by CloudSpace, where JAVA expressions are used to bind a value and to an attribute. This figure also shows some of the complications of using JAVA expressions for a two way data binding. In the value binding of the textarea, the status is being set when the user enters a new value. However, when the textarea is initially created, CloudSpace will automatically attempt to load the initial value from the return value of the `setStatus` method. This will return an exception if the `setStatus` method does not return a String value. In this case, it is better for the student to use the bean shell expression to force CloudSpace to use `getStatus` to obtain the initial value. Using a JAVA expression to explicitly execute a setter method should be restricted to one-way binding.
CloudSpace Data Bindings:

```xml
<variable type="Person" name="user" value="new Person()"/>
<label value="@{user.getName()}"/>
<textarea value="@{user.setStatus($$)} />
```

Figure 3.6: An Example of CloudSpace Data Bindings

**Template tags**

```xml
<!-- ArrayUtil.generateStringArray(int arraySize) generates an array of random strings of size arraySize -->
<variable type="String[]" name="tenItems" value="ArrayUtil.generateStringArray(10)"/>
<variable type="String" name="eachItem" value="null">
<listbox model="@{tenItems}">
  <listitem each="@{eachItem}" index="db_index">
    <listcell>
      <text value="@{eachItem}" />
    </listcell>
  </listitem>
</listbox>
```

Figure 3.7: An Example of Template Tags in CloudSpace

### 3.3.4 Templates

Students are also able to use data bindings to generate ZHTML components on the fly. Students bind iterable data structures to ZHTML tags such as lists, grids, and comboboxes to build the ZHTML components on the fly, without needing to programmatically manipulate the ZHTML page within JAVA code. Students can customize the generated tags to display any subset of the iterated data.

Figure 3.7 shows an array of Strings bound to a list tag. The CloudSpace data binding module iterates over the array and creates a listitem and contained tags for each item in the array. In the example, the template begins with the `<listitem>` tag. This tag has the `each` attribute evaluated for each item inside the array. The current array item is then set to the `eachItem` variable for use within the template code. CloudSpace also initializes and increments the variable specified in the index attribute. Starting at value 0, the `db_index`
Event Handling

```html
<html>
  <variable type="Person" name="user" value="new Person()">
  <label id="nameLabel" value="" />
  <button label="Push me!">
    <attribute name="onClick">
      nameLabel.setValue(user.getName());
    </attribute>
  </button>
</html>
```

Figure 3.8: A ZHTML Page Containing a Button and In-line Event Handler

variable will be incremented for each iteration of the template. Finally, the value of the text tag is populated with the value of the eachItem variable as the template is processed.

3.3.5 Event Handling

Because binding a JAVA object to a ZHTML component cannot express all required logic, CloudSpace also allows students to embed event handling logic directly into the ZHTML page. This event handling logic can be embedded directly into a component that invokes a particular event. This way, a snippet of JAVA code will be executed every time the event occurs. This event handling is needed only when multiple JAVA statements must be executed when an event occurs.

CloudSpace allows students to give UI tags ids. These ids can be used to refer to the underlying JAVA objects within event handling blocks so that students can manipulate ZK components directly. Figure 3.8 shows examples of an event handling block and direct manipulation of a ZK component. When the button is pushed on the ZK page, the `nameLabel` is programmatically set to the value returned from the `getName` method.
3.4 Achievements

3.4.1 Feedback in a Learning Environment

Achievement systems are one of several feedback mechanisms present in most video games today [15]. Achievements are a set of goals presented to the user with varying difficulties. Easier achievements reward the user for learning simple skills, required later in a game. These include the first killing of an enemy or basic interaction with the environment. The next set of achievements reward the player for mastering various game skills. These types of achievements are generally given to the user midway through their playing experience. They signify that the user can wield magic or fire a weapon at an advanced level. The final level of achievements signify achieving one of the overarching goals of the game, such as completing the last level and collecting a set of rare items. For the course of this paper, we will refer to the tiers of achievements as bronze, silver, and gold.

3.4.2 CloudSpace Achievements

To encourage students to go beyond the minimum requirements of their assignments, CloudSpace implements a statistic gathering framework within its infrastructure. Statistics gathered by CloudSpace include the number of page requests, the number of events fired, and the last exception thrown by the web application. CloudSpace presents these statistics to the student in a tab next to the File Manager. CloudSpace’s statistics are presented to the student as a set of achievements, similar to the trophies and achievements used in modern video game consoles.

The CloudSpace achievements encourage students to explore the features provided in CloudSpace. The control panel area of CloudSpace contains a tab for viewing all of the student’s achievements as they are developing their application. The achievement tab persists between projects so that at the end of the semester, they can look back at their work and be given
feedback on how well they utilized the CloudSpace platform. Figure 3.9 shows the achievement tab with multiple earned achievements.

### 3.5 Logging

When students are working on their desktop machine, they have access to all output printed to the console. In environments such as Eclipse and BlueJ, the console output is displayed within the IDE as the student’s code is executed. When code is run on a server; however, this output is usually redirected to log files on the server. In an environment such as Tomcat Servlet Container, all of the logging output from a single web application is redirected to a single log file. It is not practical to provide students with direct access to these log files. For example, if students were given direct access to these log files, students would be able to cheat off of other student’s solutions or violate their privacy.

To provide students with access to logging information, CloudSpace allows students to embed an interactive console in the bottom of their ZHTML page. This console acts like a console embedded in most IDE’s, which captures all output to standard error and standard out and displays the captured output in real time to the student. Students can customize the console output to display all output from their web application or limit the output to a specific stream. Using attributes on the console tag, students can customize the look and feel of the console, such as changing the color and layout. Figure 3.10 shows the Console embedded in
CloudSpace also uses this console to display CloudSpace specific log messages to the student. These include actions being performed by CloudSpace on behalf of the user. For example, in Figure 3.11 the console includes the same output from Figure 3.10; however, now it also includes all of the CloudSpace logging information. All of the green text displayed are the events that are being communicated from the client to the server. These events help students understand some of the behind-the-scenes actions as they occur.

In addition to cosmetic changes, students can also change the console from displaying the output from a single web application to displaying all output from their web application running on every client. While CloudSpace tries to hide the distributed nature of the web from students, it still creates multiple instances of an app, one for each visiting user. In these cases, students are able to add logging output to their web application and watch all output from all instances of their web application. In addition to real time distributed access, students are also able to view logs created while another user is visiting their site, allowing them to retrieve output and crash reports the next time they login.
3.6 Persistence

Modern web applications are generally able to remember user data and retrieve them in later sessions. Most enterprise persistence libraries require developers to understand JAVA annotations, transaction processing, and/or SQL. To avoid this complexity, CloudSpace’s persistent libraries provide simple API’s to store and retrieve objects in a web application. When working on their local machine, the persistence libraries store objects locally for retrieval in JUnit tests. When uploaded to CloudSpace, their web application will begin persisting objects for retrieval in the cloud.

The persistence layer in CloudSpace maps unique strings to persisted objects. Students interact with the persistence layer using check-in and check-out operations. In addition to check in and checkout semantics, CloudSpace also provides multiple scopes of persistence. The inner most scope of persistence is session level persistence. This persistence scope provides sharing inside a single instance of a web application. It is used to share data between two different pages in the same web session.

The application scope is shared between all instances of their web application. This level
is used to share persisted information such as web application configuration information. CloudSpace ensures that only one instance of a student’s web application is modifying the application level store at any time. This synchronization logic matches the sequential semantics the student expects on their ZHTML page. Objects stored in the application scope are unchanged while a specific event loop is executing; updates are not visible until the next event is processed.

The final level of persistence is shared between all students’ web applications executing on CloudSpace. The shared persistence layer allows students to share objects in a particular project with each other if these objects exhibit similar structure. For example, if students were developing a Facebook web application, they can share user profiles they have created with other students. By properly designing their user profile classes, CloudSpace allows students to load user profiles created in one application in a different student’s application.

To facilitate exchange of data from one implementation of an object to another, students are given naming conventions for the classes they expect to share with other students. These guidelines include field naming, type names, and initialization conventions. This means that for an object to be shared between two students, the class names and field names must match between the student’s class definitions, because the sharing process is based on field name to value mappings. If two students use a field with the same name, the data will be shared automatically. Otherwise, when a student attempts to load the object into memory, missing fields will be set aside and not loaded into their object. Their stored values will be recombined with the final object when it is stored in the persistence layer. Section 5.6 details the infrastructure required for persisting and reconstituting these fields.

Figure 3.12 shows an example of the conversion process for loading one student’s object using an implementation created by another student. In the example, there is a User object defined by student one. This object contains the fields **firstName**, **lastName**, and **age**. When stored in the persistence layer, the mapping of all field names to their values is preserved. When the persisted object is loaded by student two, the field name **firstName** matches a field in
student two’s implementation and the value is retrieved. However, last has no value in the persistence store. This field is left null during reconstitution. The field age and lastName are not retrieved because there is no matching field in student two’s object. When student two persists their object, all of the fields stored from student one are retrieved and stored along with student two’s changed and added fields.

When loading an object from the persistence store, the object is reconstituted with all of the matching field name to value pairs. If a mapping does not exist for a field, or the type does not match, the field is initialized to null. However, students may optionally specify special methods to initialize their classes when loaded into memory.

Special initialization functions are used to provide students with a means to specify assumptions within their object. For example, one student may allow for null values to be stored within a string field; however, when loaded into another student’s application there may be

---

Figure 3.12: Basic Demonstration of Storing an Object and Reloading it into a Different Object
an assumption that the string is never null. In another situation, one student may create an object with a birthday field while another student may create an object without a birthday field. If the object containing no birthday field is loaded into an object containing a birthday field, the birthday field in the new object will be initialized to null. Because of these inconsistencies, unexpected errors may occur inside of the class methods.

To prevent these errors, students must take precaution and provide methods to properly initialize their classes. Students must include either a method called `initializeContext` or implement a default constructor in their persisted object. These methods are executed to initialize an object to a default state before loading the persisted fields into them. Without these methods, all fields would be initialized to null.

CloudSpace logs any errors and warnings encountered while reconstituting the object. For example, loading a persisted object from the persistence layer that is missing fields CloudSpace will report that the student’s object has not had every field initialized and suggest that the student harden their code.

CloudSpace also allows for consistent loading of the data being put into the persistence store. For example, if students were sharing a user object with a `name` field and the `name` field was required for all user objects, it would be important for the value stored in the name field to be non-null. CloudSpace will check objects as they are put into the persistence store to ensure they conform to these requirements. If a student attempts to put an invalid value into the persistence store, CloudSpace will deny the action and throw an exception that reports which rule is violated to the user.

To access the persistence store, students use an implementation of the `java.util.Map` interface. To access the different layers, the student creates an instance of one of three maps (`SharedPersistenceMap`, `ApplicationPersistenceMap`, and `SessionPersistenceMap`). When creating these maps, students pass a generic type argument to the constructor. This limits the persistence map to the storage and retrieval of a single type of class, allowing strong type checks on any values persisted and retrieved. Figure 3.13 shows an example of code a
SharedPersistenceMap<User> userMap = new SharedPersistenceMap<User>(User.class);

public static User authenticateUser(String userName, String password) {
    User lookUp = userMap.get(userName);
    if (lookUp != null && lookUp.getPassword().equals(password))
        return lookUp;
    return null;
}

Figure 3.13: An Example of the Shared Persistent Store in Client Code.

student would write to access the persistence store.

3.7 Authentication

3.7.1 Existing Authentication Methods

Authentication schemes associate user names to web sessions. There are multiple strategies
that support authenticating web sessions. One of the lowest level authentication schemes
is HTTP basic authentication. In this scheme, the web browser is instructed by the server
to send credentials to the server with each request. In a more complex setup, the server
instructs the client to send a special unique sequence of bits to identify themselves. Then
the server is responsible for retrieving the client’s authentication credentials and linking them
to the current session.

To implement HTTP basic authentication, students would have to understand the HTTP
protocol and import special libraries into their desktop workspace. Developing HTTP basic
authentication would put too much focus on the web when their focus should be on core
CS1 concepts. Alternatively, cookie authentication schemes require students to track unique
identifiers and store them for use in multiple instances of their web application. However, we
attempt to hide the distributed nature of the web from students. The idea of sharing a map
between instance of their web application and the generation of cookies for each instance
breaks this illusion.

CloudSpace already provides students with access to session level objects. As an alternative approach, CloudSpace instructs users to persist any profile information in the session persistent store. The persistent store is managed for the student creating a simplified API that abstracts a low level authentication scheme.

### 3.7.2 Authentication in the Cloud

The authentication system provided to students in CloudSpace is limited to simple setting and getting of a particular persisted object. This mechanism is provided to students using the `SessionPersistentMap`. Using this map, students set and get a particular key (such as `currentUser`). This object can be of any type; however, there can only be one of these objects per web application.

The authentication map provides a way for students to save the current user but does not provide any security features by itself. The map interface is provided to help students conceptualize the idea of server side authentication. It also provides students with an easy way to test their application using JUnit tests. Instead of testing their code with an implementation specific solution, they can perform an application log-on attempt and check the currently logged in user without coupling that check to their application.

### 3.8 Exception Doctor

As entry-level programmers begin learning JAVA, they soon encounter JAVA’s runtime exception mechanism inherent which requires understanding of stack traces and exception messages. To understand the messages that the JVM provides the user, the student must understand what causes that exception in the first place. CloudSpace integrates the tool ExceptionDoctor to rewrite JAVA runtime exception messages to be user friendly and context
sensitive.

ExceptionDoctor’s improved exception messages include three pieces of information: the offending line of code (if available), an explanation of the meaning of the exception and the most likely cause(s), and the original exceptions stack trace. The improved exception message is derived from the original exception and contains information derived from source code analysis, such as how certain variables may have contributed to the exceptions cause. The improved message is presented in paragraph form with ExceptionDoctor’s best guess on which portions of the source code caused the problem, how the issue might be avoided, and the name of the exception printed at the bottom.

3.8.1 Examples of Common Exception Situations

File I/O, null reference access, and type errors are the most common exceptions to arise for novice programmers [12]. Exception Doctor augments each of these runtime exceptions.

The first common exception type is file I/O related. File I/O exceptions can be caused by novices attempting to open a file or write to a file. If a student is attempting to open a file, there are generally two states that the file can be in that will result in an exception. The first state is a java.io.File that has no corresponding file on the file system. When the file does not exist, ExceptionDoctor checks the given path for errors in the directory structure. If the directory path is correct, then the user is simply told that the file does not exist. However, if the path is incorrect, it is more likely that there is a spelling mistake in the path. Figure 3.14 gives an example of output in this situation. File exceptions can also occur because of insufficient file permissions.

I/O errors are also caused by errors in writing and reading strings from input and output streams. Because IOExceptions can occur for many reasons, ExceptionDoctor does includes message support for every situation that could cause an IOException. However, through diagnostic tools such as ClockIt [13], an instructor can gather information about the types
Code:

```java
String fileName = "\home/mike/notreal/foo.bar";
File f = new File(fileName);
Scanner s = new Scanner(f);
```

Exception Message:

```
java.io.FileNotFoundException:
   In file Junit3Test.java on line 15, which reads:

   Scanner s = new Scanner(f);
```

It appears that the code was trying to operate on a file called \
/home/mike/notreal/foo.bar. However, it seems that this file may not 
exist. Check that the filename is spelled correctly. Analysis shows 
that /home/mike/ is a valid path. The remainder of the file name is 
invalid.

This error is called an FileNotFoundException.

```
   at Junit3Test.testNoFile(Junit3Test.java:15)
```

Figure 3.14: An Example of a FileNotFoundException
Code:

String nullString = null;
Integer i = 1;
Integer j = 2;
nullString.substring(i, j);

Exception Message:

java.lang.NullPointerException:
   In file Junit3Test.java on line 22, which reads:

   nullString.substring(i, j);

It appears that the code was trying to call a method or refer to a
member variable on an object called "nullString", which is null.
Make sure the variable has been initialized in your code. Remember,
declaring the variable isn’t the same as initializing it. You may
need to initialize the object using the keyword "new".

This error is called an NullPointerException.

   at Junit3Test.testNull(Junit3Test.java:22)

Figure 3.15: An Example of a NullPointerException

of exceptions that students are encountering and create a custom IO handler class to remind
students of lessons they learned in class.

The next most common kind of exception is due to attempt at dereferencing null references.
A null pointer dereference manifests itself as a NullPointerException. When a null pointer is
dereferenced, ExceptionDoctor attempts to locate the variable in the source code. If there is
only one candidate in the line, the explanation provides that variable name in describing the
cause. If there are multiple potentially null variables, a list of all such variables is included
in the message instead. Figure 3.15 shows example output for the single-variable case.

Bounds exceptions can be caused by data structures with access requests to variables outside
of the declared bounds. The simplest data structure that suffers from this error is the
array. When ExceptionDoctor encounters an ArrayOutOfBoundException, it attempts to
**Code:**

```java
Object[] tenElementArray = new Object[10];
Object foo = tenElementArray[15];
```

**Exception Message:**

Java.lang.ArrayIndexOutOfBoundsException:

In file Junit3Test.java on line 27, which reads:

```java
Object foo = tenElementArray[15];
```

It seems that the code tried to use an illegal value as an index to an array. The code was trying to access an element at index 15 of the array called "tenElementArray". The size of the array may be less than 15. Keep in mind that if the array size is N, the biggest index you can access is N-1.

This error is called an ArrayIndexOutOfBoundsException.

```java
at Junit3Test.testBadIdx(Junit3Test.java:27)
```

Figure 3.16: An Example of an ArrayIndexOutOfBoundsException

Narrow the exception cause down to three common mistakes. ExceptionDoctor can identify errors caused by negative array indexes and using an array index that is greater than the array size. The ArrayIndexOutOfBoundsException contains only the source code line that caused the exception and the value of the index. Based on these two pieces of information, ExceptionDoctor gives the student hints on the exception’s cause based on its best guesses about the array. Figure 3.16 shows the exception resulting from accessing a element 15 in a 10 element array.

NumberFormatExceptions occur when students use a different number type than is required to hold a particular value contained in a string. When ExceptionDoctor encounters a NumberFormatException, it attempts to determine a suitable type by repeatedly parsing the string in question. Figure 3.17 shows an example.
Code:

Integer.parseInt("3.59");

Exception Message:

java.lang.NumberFormatException:
   In file Junit3Test.java on line 31, which reads:

   Integer.parseInt("3.59");

   It seems that the code wants to convert a String to an integer. However, the String "3.59" appears to be a floating point value, not an integer. You may want to use a different datatype to store the value, like float.

   This error is called an NumberFormatException.

   at Junit3Test.testParse(Junit3Test.java:31)

Figure 3.17: An Example of a NumberFormatException
Chapter 4

Teaching in the Cloud

Teaching students CS1 concepts with the CloudSpace environment does not always require rewriting existing assignments. Instead of rewriting assignments, CloudSpace allows professors to enhance existing assignments so they can be used in a more interesting context. Most entry level computer science projects relate to a set of algorithms and data structures in which students acquire proficiency. Many of these algorithms are also building blocks of popular applications on the web, thus allowing students to transfer their acquired knowledge and skills.

CloudSpace hides many of the complexities of the web, reducing the amount of CloudSpace-specific material that must be taught to students. In CloudSpace, the distributed nature of web applications is hidden from the students to whom it appears as though the application exists entirely on the server. This means that web application development is no different from desktop GUI libraries such as swing and AWT. This server centric view of web development allows the instructor to ignore concepts such as AJAX requests. Students do not need to understand any JAVA Script to harness the full power of CloudSpace’s ZHTML pages. Instead of learning AJAX, students view their web applications as event processors. CloudSpace also forces every web application to execute in a single threaded fashion so that students do not need to understand concurrency.
4.1 Initial Configuration

4.1.1 Installation

One of the design goals for CloudSpace was to make adoption by other instructors as easy as possible. Our easiest deployment method is the use of an automatically installing WAR file. The WAR file can be downloaded from the CloudSpace distribution site and placed directly into a servlet container’s web application directory. The servlet container will then decompress the WAR and deploy CloudSpace into the servlet container. Alternatively, CloudSpace can also be deployed directly from the source tree distributed from CloudSpace’s source code repository.

The first step in the installation process is to download the entire source tree from the CloudSpace SVN repository. At the root of the tree, there is an ant build script for compiling CloudSpace as a web application. The ant file will download all necessary dependencies at build time.

CloudSpace can be installed into a servlet container with symlinks or deployable WAR files. WAR files are deployable compressed files which are automatically expanded and installed in a web application directory by servlet containers. Any upgrades require redeploying the WAR file. When a new WAR is deployed, all of the configuration files within the web application directory will be reset to defaults.

CloudSpace also allows for a configurable storage directory. When CloudSpace is started for the first time, the user is presented with a single user edition of CloudSpace. In this mode, CloudSpace requires no passwords and operates in a sandboxed environment. Using a web interface, the storage directory can be changed to a directory outside of the servlet web application directory. In this case, the web application can be upgraded without disturbing the configuration files.

The most flexible installation option is to use symbolic links to install CloudSpace. Using
a symbolic link, the WAR directory can be linked into the web application directory. In this case, the original directory can be updated with SVN and rebuilt and the servlet container will automatically pick up any changes. However, this method requires operating system and servlet container support.

### 4.1.2 Authentication Configuration

To configure user authentication, CloudSpace provides local and remote authentication schemes. Local authentication uses student ids and generated passwords that are stored in a hashed format on the local file system. This authentication scheme requires the least amount of configuration by instructors. All configuration steps are performed using the CloudSpace administrator control panel. This authentication also does not require the server to authenticate users over a secure channel. Because CloudSpace has access to the passwords in hashed form, HTTP digest authentication can be used to obtain credentials securely.

Remote authentication allows instructors to use institution provided credentials to authenticate users. To support remote authentication schemes that use standards such as LDAP, as well as those requiring special libraries or cryptographic protocols, a custom authentication class must be implemented to authenticate users using remote authentication servers.

All authentication requests in CloudSpace are parsed and processed by the Spring Security framework, which is integrated into CloudSpace. Spring Security’s configuration files allow an instructor to implement a custom authentication class and easily integrate it into the authentication logic. Spring Security provides implementations for LDAP and CAS authentication. Many institutions also require importing a private root certificate to authenticate institutional servers. This root certificate must be imported using a separate, system-dependent method. If remote authentication is chosen, an instructor must use secure sockets to transfer credentials to avoid sending passwords in clear text.

While local authentication allows the instructor to use insecure channels to obtain security
credentials, it is still a good idea to use secure channels for all CloudSpace web requests. Without a secure channel, CloudSpace suffers from the same cookie authentication weaknesses that plague web applications such as Google and Facebook (circa 2010)[3].

To activate student accounts on CloudSpace, instructors can add each student through a user control panel shown in figure 4.1 or upload a CSV spread sheet. If instructors add students one at a time they are able to configure directory access for each student and place students in special groups that provide students with group permissions. The user configuration also allows instructors to choose the authentication method that should be used for each user. Each user can also be enabled and disabled from this configuration page.

When a CSV file containing users is available, CloudSpace allows batch creation of users. Figure 4.2 shows the CSV upload window. When a role is uploaded to CloudSpace, instructors are presented with a CSV interpretation page. On this page, columns can be marked with the headers user name and password if desired. CloudSpace can also generate random passwords for students if requested.

After headers have been marked, a batch policy can be chosen for the new users. First, the
authentication method is chosen. New users can be authenticated with provided passwords, randomly generated passwords, or using a remote authentication system. Then, a directory is chosen in which to place each student’s workspace directory, which stores their personal work. Finally, new users can be put into a shared group for group permission configuration.

4.1.3 Permissions

In addition to configuring CloudSpace’s authentication, instructors must set up each student’s directory permissions. CloudSpace supports per-user and per-group permissions for individual directories. Students may have permissions to read or write within indicated directories.

CloudSpace also allows users to have permissions based on group membership. Each group in CloudSpace can be given permissions for directories similar to the permissions provided to standard users. CloudSpace’s group authentication is useful for both group projects and special instructor provided source code.
4.2 Project Composition

CloudSpace projects are composed of two parts. The first part is focused on standard entry level computer science concepts. In this phase, students focus their development on creating the data structures and algorithms that will form the backbone of their model. To develop this part, students can use techniques such as test-driven development to perfect their model without using CloudSpace. More importantly, this phase is devoid of any UI development. The only CloudSpace libraries used by the students are libraries they will use to access the persistence and authentication services. Using this development model, the user interface development is a completely separate creative process. This allows for JAVA development and UI development to be graded separately.

After students have completed their code, they upload the class files and potentially their source code to the CloudSpace server. This phase in the development process requires the student to define their ZHTML page and work through any bugs they may have missed. The skills gained through the development of ZHTML pages is not restricted to CloudSpace because ZHTML mostly contains standard HTML tags, an added small subset of ZUL components, and optionally CSS style sheets. Only the ZUL components are specific to CloudSpace, but they are very similar to widely used technologies such as Mozilla’s XUL markup language. CloudSpace does not require professors to teach web technologies such as AJAX, JAVAScript, or JAVA servlet containers.

Before students can use the CloudSpace environment, they need experience defining classes, instantiating objects, and performing method calls on objects. These skills are required for basic ZHTML development. Students must also have a basic understanding of generic types to understand the JAVA documentation provided for the persistence libraries.

CloudSpace embraces this separation of web technologies and CS1 concepts to ensure that the core curriculum of a CS1 course is still the focus of CloudSpace projects. If students had to learn technologies such as AJAX, JAVAScript, and other web specific terms they might
be unable to learn the skills that would make them successful in future computer science courses.

However, this philosophy comes at a price. Because we are providing so much technology to students, the underlying machinery is hidden. For example, all of the AJAX requests are automatically generated by ZK and the HTML source code sent to browsers is a divergent output from what students have created.

### 4.3 Teaching ZHTML

Instructors must teach students how to write ZHTML pages. The skills learned can be applied to future course work. To define the ZHTML pages, students must be taught how to define a HTML page and how to use some additional ZUL tags. While the HTML tags follow the HTML standard, the ZUL tags are derived from the XUL component set defined by Mozilla. These XUL components are used to describe the UI of the Firefox browser and are similar to technologies such as XAML, a Microsoft technology used for UI design. Optionally, instructors can teach students how to define CSS pages.

In addition to the UI layout, instructors must teach students how to define and use JAVA bean expressions. JAVA bean expressions are used to bind JAVA objects to the data displayed on a ZHTML page. To reduce confusion between JAVA bean shell and standard JAVA syntax, CloudSpace has modified the binding syntax that can be used in ZHTML pages.

In the original ZK binding syntax, students could use only JAVA bean syntax within the data bindings. To eliminate confusion, CloudSpace allows for both JAVA bean syntax and standard JAVA expressions inside of the data bindings. CloudSpace also automatically attempts to convert a standard JAVA expressions of the form `get<fieldName>()` into a two way expression. If students always write both a setter and getter for their fields and use the getter in all bindings regardless of the direction the data is flowing, students never need to
use bean shell syntax. This reduces the amount of CloudSpace-specific knowledge required of students and removes potential confusion about the differences between JAVA syntax and bean syntax. For example, if a student binds the expression @profile.getName() to a textbox, CloudSpace will automatically convert the binding into @profile.getName() for reading the value from the model and @profile.setName(textbox.value) for saving user input to the model.

ZHTML components such as grids, comboboxes, and lists allow data structures to be bound to a model attribute. When the component is built, the data structure is iterated over and any child or contained component is duplicated for each entry in the data structure. The creation of the template components closely mimics a foreach loop structure.

4.4 Teaching Persistence

One approach to teaching persistence is to restrict students to using application and session levels of persistence. This is an acceptable solution if students intend to persist objects only for use in their own application. When students use only application and session levels of persistence, instructors need to teach them only how to use java.util.Map classes.

To use the application and session levels of persistence, students use the CloudSpace libraries to request objects from a map using a unique key. The objects returned from the session and application are protected from concurrent access. Each student web application executing on CloudSpace executes events on a first-come, first-serve basis. However, between each instance of a web application’s execution the objects stored may change.

The shared persistence layer allows students to share JAVA objects they have created with other students’ web applications. For this layer to function properly, students must be instructed to use a common structure for their shared objects. For example, if two students are creating a Facebook application, they can share profile object data. To share this data, students must be given a set of field names and types.
Students may make different assumptions about the values of fields within their objects. For example, student A may allow a user name in their user profile to be null while student B explicitly sets their user name to the null string at class creation. When student B loads a class persisted by student A, their class is populated with an unexpected null value. This null user name can result in a null pointer exception that the student never tested for.

One solution to this problem is to tell students that certain shared fields between the classes must conform to certain requirements. However, students tend to miss instructions in design documentation. Once an invalid object is committed to the shared persistent store, the persistent store is poisoned with a malformed object that violates requirements. To provide instructors with more control over data in the persistence layer, CloudSpace can enforce object requirements when students store objects. The CloudSpace control panel allows for two methods of defining these requirements.

Instructors can upload a JAR with specially annotated classes to CloudSpace. The JAR is extracted and scanned for any classes annotated with \texttt{@PersistenceTemplate} annotation. Classes containing this annotation are considered as prototypes for persisted objects. The prototype classes are scanned for \texttt{@NeverNull} and \texttt{@Nullable} annotations. If a \texttt{@NeverNull} or \texttt{@Nullable} annotation is applied to the definition of a class, it is treated as the default annotation for all fields within the object. If no annotation is provided, CloudSpace treats \texttt{@Nullable} as the default field annotation. After determining the default annotation, each of the fields within the class are checked for \texttt{@NeverNull} or \texttt{@Nullable} annotations. Parsed annotations are stored in a configuration file for quicker access. The second way to define these persistence templates is to add them to CloudSpace directly. The CloudSpace control panel provides a user interface for adding and removing classes and annotations on fields. Figure 4.3 shows a persistent template for a User class.

After prototypes have been defined, each student object stored in the shared persistent store is verified using all applicable templates. If a template exists with a matching class name, the object data is inspected for template violations. If a violation is detected, a run-time
Figure 4.3: Control Panel for Creating and Maintaining Persistence Templates
exception is thrown and a message is displayed.

Problems can still occur when students check objects into the shared persistent store. For example, student A may have a user profile with a current school field where student B does not have a field representing the current school. If student A loads a profile created by student B, the current school field will be null because it is undefined in the persistence store.

To address this, CloudSpace performs a three step initialization process. First the default constructor is used to initialize the class (if present). Next, the initializeContext method is called (if present). Finally, each of the fields are reloaded into the object. If fields are still uninitialized, CloudSpace reports that some fields could not be retrieved from the store.
Chapter 5

An Infrastructure in the Cloud

5.1 ZK Modifications

CloudSpace contains modifications to the ZK infrastructure to make it better suited for student development. This includes modifications to the data binding syntax, the bean shell interpreter, and the ZHTML language.

5.1.1 Data bindings

Previous versions of CloudSpace used the stock ZK data binding support. We found that students had difficulty understanding the difference between bean shell syntax and typical JAVA syntax. This misunderstanding stems from the fact that many of the data bindings are read only. In this case, any right-hand side JAVA expression should be legal. However, when students entered such a JAVA expression, an error would be returned.

To simplify the data binding syntax, CloudSpace allows students to use fully qualified JAVA statements inside of data binding expressions. To implement this change, an intermediate variable was introduced into the binding mechanism. Instead of a single bean shell expression
being evaluated to obtain the value of a tag attribute, CloudSpace now executes two separate statements. When loading data from the server into an attribute, CloudSpace first executes a bean shell expression that runs the bound JAVA expression and sets an intermediate variable to its value. After the intermediate variable has been set, the normal binding framework is used to set the attribute to the value of the intermediate variable. When data is being retrieved from an attribute and loaded into a server side object, the opposite logic is used. The intermediate variable is set to the value of the attribute and the bound JAVA expression is executed. However, because the intermediate variable is now passed as a parameter, the user must use the special variable "$\$$" to refer to the intermediate variable in their expression. Figure 5.1 shows the flow from an attribute to the server and vice versa. CloudSpace has also modified the bean shell interpreter. In the stock ZK infrastructure, there are multiple "scopes" for bean shell expressions. Each scope encapsulates the next,
preventing inner scopes from referencing variables in outer scopes. Unfortunately, the scoping levels in a ZHTML page are not always easily recognizable.

To simplify the scopes within a ZHTML page, CloudSpace defines three different levels of scopes. At the top level, the ZHTML page itself is a scope. Almost all bean shell variables on the page are stored in the page scope. The next scope is the include tag. This tag allows students to include a file full of tags within the top level ZHTML page. This barrier mimics the barrier you would see in a method call to another class where the logic is stored in a separate class file. This also reduces page coupling and errors where students might assume a variable has been defined when it hasn’t.

The final scope level is macros. ZK macros allow students to define new tags that will expand into a set of multiple tags. Because of the similarities between the macro tag and the include tag, the same scope rules are preserved. Scope barriers have been removed from both template tags and window tags. Both of these scopes allowed for multiple scopes to be created in a single ZHTML page which conflicts with standard JAVA syntax scopes.

The changes to the bean shell scope also changes scoping rules for all components on the page. Now any ZHTML tag with an id can be referenced by any data binding or event handling code within the page. The new data binding scopes also removes scoping from template tags. When template tags were expanded, their JAVA expressions were unable to access any variables outside of the template tag itself.

### 5.1.2 Adding ZUL to ZHTML

The default ZK infrastructure separates the ZK tags into two distinct name spaces. The first name space is the ZUL name space. This name space includes all of the nonstandard HTML tags defined by the XUL Mozilla standard. The ZUL tags are available by default in files ending in the *.zul extension.

The second name space is the ZHTML name space. This name space only includes ZK’s
implementation of standard ZHTML pages. To make ZUL tags available to students in ZHTML pages, CloudSpace merges the set of tags in the ZHTML language with the tags in the ZUL language. This modifies the name space used when students define *.zhtml pages without requiring students to prefix ZUL tags with a name space declaration.

5.2 Virtual Machine

All student code executes in a virtual machine. The virtual machine infrastructure forms the backbone of all CloudSpace operations. It provides logging support, identifies root file systems, and manages class loaders for each student’s project. The virtual machine
infrastructure acts as a shim layer between the ZK framework and student’s application code. Figure 5.2 shows the lifecycle of a virtual machine, starting with a page request and ending with student code execution.

Each virtual machine in CloudSpace is tied to a single project. Projects on CloudSpace are identified by the root directory of a student’s web application. By default, each directory created by a student is treated as a project directory. When a page is initially requested by a client, CloudSpace enters an initialization phase for the virtual machine. The first initialization step is to identify the root directory of the page that has been requested. The root directory is characterized by the location of the compiled JAVA classes and all of the pages and directories required to build the student’s web application. Typically, the root directory is the directory in which the page is located. However, if a student wishes to create directories within their project directories, they can define “project” folders.

Project folders create a single virtual machine for multiple folders. Students create project folders through the File Manager provided by CloudSpace. Project folders allow students to make web applications with multiple ZHTML pages in multiple directories. Without project folders, a virtual machine would be created for each directory in the project directory. After the root directory for the virtual machine has been determined, the virtual machine is responsible for initializing the services each student’s web application depends on.

During the initialization phase, the virtual machine creates a custom class loader, which is used to load all of the code executing on behalf of the student’s ZHTML page. This class loader delegates all class loading requests to its parent class loader before attempting to load the class itself. Hence, infrastructure code takes precedence over any code created by a student. The custom class loader includes two directories in its class path.

The first directory is the student’s code. This code is loaded by the custom loader and rewritten for the cloud environment. The class modifications are discussed in section 5.3. The second directory is a directory shared by all students and all web applications. This directory contains all common page libraries, which are also rewritten by the class loader.
The created class loader is installed directly into the ZK bean shell interpreter. All code embedded in the ZHTML page is executed through the ZK bean shell interpreter. Since the bean shell interpreter for the page uses the class loader specified by the CloudSpace virtual machine, all code executing on behalf the page will use the student’s classes. All other student code on the server will is hidden from the class loader.

The virtual machine is also responsible for detecting changes in the student code. Student code changes when students upload new class files or modify JAVA sources and compile them server side. To detect such changes, CloudSpace inspects the student’s code looking for modified class files. If modified classes files exist, CloudSpace instantiates a new virtual machine for all future requests. The previously used instance continues to service all existing web applications; however, after all web application instances are destroyed, the old virtual machine is removed from CloudSpace.

5.3 Byte Code Engineering

After the CloudSpace virtual machine has been put into place and the custom class loader is installed into the ZK framework, classes are loaded as ZHTML pages are created. However, because these classes were designed for use on student’s local machines, the bytecode has certain built-in assumptions about the execution environment. CloudSpace uses bytecode rewriting to rewrite student’s JAVA code.

When classes are loaded by CloudSpace, CloudSpace hooks itself into the loading process to modify the class bytecode before it is cached by the class loader for use. To perform the rewriting of class bytecode at runtime, CloudSpace uses the Javassist framework.

CloudSpace developers can implement a configuration file that stores configuration directives for the Javassist rewriting framework. There are three different types of configuration directives: translations, macros, and proxies. These directives are combined to modify student code to run in a cloud context. Table 5.1 describes each of the directives.
The translation directive denotes a request to modify a JAVA expression or method within student code.

The proxy directive denotes a class that should be intercepted in ZHTML pages to use a wrapper that rewrites each method using translation directives.

The macro directive is similar to a method or function. A macro is a repeatable block of code to be used in translation directives.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>translation</td>
<td>The translation directive denotes a request to modify a JAVA expression or method within student code.</td>
</tr>
<tr>
<td>proxy</td>
<td>The proxy directive denotes a class that should be intercepted in ZHTML pages to use a wrapper that rewrites each method using translation directives.</td>
</tr>
<tr>
<td>macro</td>
<td>The macro directive is similar to a method or function. A macro is a repeatable block of code to be used in translation directives.</td>
</tr>
</tbody>
</table>

Table 5.1: A List of Bytecode Directives and Descriptions

The first type of directive is translation. Translations define the changes required for a particular JAVA statement. Each translation includes a translation type. This type defines the logic that should be used when replacing an existing expression in the bytecode. Translation types include replacing an entire JAVA expression with a new one, adding new logic at the beginning of a set of statements, and adding logic to the end of a set of statements. With each translation type, the developer also defines the type of statement that should be translated. These statement types include method bodies, constructor calls, and new expression calls.

Modification rules may apply to a single place in the bytecode or to multiple places. For example, a method body translation requires only a modification of a single class. On the other hand, method call translations trigger bytecode rewriting at each call site. CloudSpace uses a Javassist expression translator to handle such expression modifications. This translator automatically scans all class files for JAVA statements that may need modification and provides a hook to CloudSpace to modify the statements on the fly. Figure 5.3 shows an example of a translation directive.

The second type of directive is a macro directive. Macro directives are used to define behavior that will be repeated multiple times in other translation directives. Macro definitions are commonly used for JAVA classes that have multiple methods with similar logic. Macro directives are composed of a macro signature and the macro body. A macro signature contains a macro name and a set of typeless parameters. Using the macro name prefixed by
a % character and a set of parameters, the macro can be used like a C preprocessor macro in translation directives. The macro body contains the logic to be injected into translation directives. Inside of the macro body, parameters can be referenced using the $ character before the parameter name.

The final type of configuration directive is the proxy directive. This directive instructs CloudSpace to create a proxy class for all bean shell expressions used in the student’s ZHTML page. Proxy classes are generated classes that wrap all method calls to a particular class. For example, if the java.io.File class was proxied, each of the methods defined within the class would be recreated in a static file wrapper class. The wrapper class would contain a static method for each method call. Inside of each of the method bodies would be a call to the actual file methods. After the proxy class is created, it replaces all calls within beanshell expressions. When a ZHTML page uses a File object, instead of executing normal file logic, the execution will go through the wrapper. Using this proxy method, proxy classes are loaded using the same loading scheme as student loaded classes. This means that method calls within the proxy will be subject to the same rewriting rules.

By default, CloudSpace provides directives to remove assumptions about local environments that may be embedded in student code. These assumptions include file system layout and system calls. All configurations are contained in a single configuration file that can be modified to suit custom environments.

When students develop locally, they assume that the code is executing in a particular local directory. When their code is uploaded to CloudSpace, assumptions such as the location of
the current working directory or the file system layout do no longer hold.

CloudSpace provides a virtual root directory to each student. To virtualize the actual host’s file system, the Javassist rewriting framework translates relative local paths to absolute paths on CloudSpace. Most of the file system translation directives modify the strings that are passed into methods of the File, FileReader, and FileWriter classes. The translation ensures that each student’s files are contained in an assigned directory. In addition to adding or removing prefixes to the path, CloudSpace converts Windows style paths to JAVA standard format.

The second half of translation directives relates to the recovery of path information passed into CloudSpace by students. In addition to the assumptions about the paths used to create files, students potentially make assumptions about the paths returned from those files. To recover the paths for returning to students, the paths students use to create files are cached for later recovery.

## 5.4 Logging Console

To provide students with access to all of their printed output, CloudSpace captures print operations and stores the output for later retrieval. All printed output is associated with the current operating virtual machine.

Typically, standard output is captured by the JAVA virtual machine and redirected to the system’s output console. In servlet containers such as Tomcat, an intermediate output stream is added as a wrapper to the standard output stream installed by the JVM. CloudSpace uses a similar chaining mechanism to install a custom demultiplexing output stream implementation into the System’s standard console output stream object. This demultiplexing output stream captures all output created by code running inside the current JVM and determines if there is a virtual machine associated with the print operation. If there is no virtual machine associated with the print then the demultiplexer does not process the print operation and
Figure 5.4: The Flow of Prints Through CloudSpace’s Demultiplexer and an Attached ZHTML Page

passes the message to the next output stream in the chain. Checking for a virtual machine is performed in a robust way to prevent cross talk between multiple applications running in the same virtual machine. For example, if two instances of CloudSpace are executing in the same virtual machine, then two demultiplexers will be installed in the system’s console output logger. Each demultiplexer must identify print operations originating from its owning web app and the other web app executing on the server. Figure 5.4 shows the flow of a print request through the demultiplexer.

If the demultiplexer identifies the print operation as a relevant logging operation, CloudSpace will capture the logging message and store it for later retrieval. When each message is stored,
CloudSpace preserves the absolute order of the printed output, a time stamp representing the current system clock, and the ZK desktop the output is associated with. The absolute ordering of the output is important because print operations can occur at a faster speed than the granularity of the system’s internal clock. The time stamp of each line printed is still stored; however, to provide a visual indication to the student what time the print occurs.

In addition to tracking time sensitive information, CloudSpace also stores the id of the currently executing ZK desktop. This information allows printed output to be associated with particular sessions within a single student web application’s lifetime. It also allows for later retrieval of other desktops’ output.

After the logging information is gathered, it is stored within the currently executing virtual machine. The current virtual machine maintains a history of all printed output in a circular buffer to prevent unbounded memory allocation. In addition to storing all system console output, the virtual machine also stores all error console output. This increases the importance of the absolute ordering of print operations. If each error and system output log message is stored in separate buffers, there needs to be an absolute way of comparing their ordering. As an alternative solution to this problem, a single circular buffer could be implemented to store all output generated by the application. This approach; however, limits the ability to save important messages such as exception traces and error output. If an application printed huge amounts of standard output, the important error messages would be quickly overwritten.

Each virtual machine maintains the list of output for its lifetime. After class files or ZHTML file for the current application have been modified, the current virtual machine’s output will be discarded. This simple scheme resembles desktop environments such as Eclipse IDE, in which the console is also cleared between each execution.

When stored in the virtual machine, logging information is made available to students as an embedded component in their ZHTML page. The component is composed of two parts. The first part of the console is the text output. The text output is generated in a read
only text area. The logged output stored in the virtual machine is synchronized with the console in five second intervals using Comet long polling. This means that without any user interaction, logging information will display inside the box in near real time.

CloudSpace uses ZK’s built-in long polling algorithm. The built-in algorithm requires the page to send a ”dummy” event to the server every second. When the dummy event is received by the server, CloudSpace piggybacks on the event to push any server side changes back to the client. On CloudSpace, a scheduled task is created for each executing console. This task is run every five seconds and requests a status update from the virtual machine. If the virtual machine has new logged output, the update thread processes it and adds it to a server side text area element. During the piggy back operation, the ZK infrastructure identifies the new log content and ship it back to the client.

On the browser, the printed output will be appended to the text area by client side JAVAScript. As output is added to the text area, a separate JAVAScript algorithm constantly scrolls the text area to the bottom of the output. This auto scrolling feature closely mimics that found in standard IDE’s.

The second component in the client side console is a console toolbar. The toolbar allows students to control the output that is displayed in the console text area. This toolbar allows CloudSpace to provide students with control over the streams displayed and the format of the output. As discussed, the CloudSpace virtual machine console logging stores error and standard output streams. Using the console toolbar, students are able to turn the visibility of each stream on and off.

In addition to stream visibility, the toolbar allows students to decide what logging output should be displayed according to the session that created the logging output. By default the embedded console shows students only printed output of the current session. This means that any previous output stored by the virtual machine is hidden, similar to executing a new instance of a single-threaded desktop application. To debug conditions that occurred in previous interactions with their application, the console toolbar also allows students to view
all output generated by their web application during the current virtual machine’s lifetime.

When this option is selected, students can see all past logging output and all output being generated by currently executing instances of their web applications. To help students understand the temporal relationship between the prints, the console toolbar also allows students to turn on time stamp visibility. The stored timestamps are appended to the beginning of the line and may be turned off at any time.

In addition to menus for each of the options, the console toolbar can be configured using tag attributes. The attributes control all of the previously discussed options and additional options such as color.

## 5.5 Achievements

The achievement system in CloudSpace is extensible to include new achievements as new incentives. The achievement infrastructure is composed of statistic points and the statistic manager. Each time a statistic point is reached, the manager is notified that an event of interest has occurred. The statistic manager is able to determine the level of achievement the student has reached and award them a bronze, silver, or gold trophy.

Achievement configuration begins with a server side configuration file. This file defines each tracked statistic using six parameters. The first parameter is the achievement ID. This ID is used to uniquely identify each statistic for reporting and generation purposes. The next three parameters define the thresholds required for the bronze, silver, and gold achievement levels. The fourth parameter is the units used to display achievement data to users. The final parameter is the statistic point class used to track the achievement.

The statistic point class used to track the achievement implements the `java.util.Observer` interface. This interface defines a method that is passed an observed event and an optional additional object. The observed event contains all information about the statistic that was
just observed and the second parameter is unused.

CloudSpace includes default implementations of iterative and time based statistic points. Iterative statistics have levels based on a counter that starts at zero and advances each time the event occurs. When the statistic’s value becomes greater than the achievement threshold, the user is awarded the achievement.

Time based statistic points track the amount of time that has passed since the last event. When initialized, the statistic’s value is set to the current system time. Users achieve levels as the time since the last event becomes longer, discouraging bad behavior. For example, by default CloudSpace tracks the last time a student’s web application threw an exception, rewarding a student for causing exceptions rarely.

The default configuration in CloudSpace includes four statistic events. The first three events are positive reinforcement. These events track the number of files opened, number of ZK events fired, and the number of times all of the student’s pages have been loaded. CloudSpace includes one negative reinforcement statistic for tracking the amount of time that has passed since the last exception thrown by their web applications.

Each time a statistic has been tracked, the statistic service persists the changes to disk. To preserve the statistics, each user’s data is converted into a JAVA property object consisting of the statistic ID and current value. Each time the statistic service is initialized for a user in the future, the persisted property file is used to rebuild the tracked statistic.

5.6 Persistence

When the persistence library was designed, three different designs were considered. The first design, which was implemented for use in beta versions of CloudSpace, used an Application class that students subclassed. This application class provided utility methods to access the different levels of the persistence store. In this model, students would make calls to setters
and getters for each of the levels.

To access the session layer, students use `setSessionObject` with two parameters. The first parameter is the key that will be used to look up the object and the second parameter is the object to be stored. All objects stored in the session persistence layer are stored in a map that is associated with the currently executing ZK session. The ZK framework manages the storage and destruction of the map. Because this layer is not shared or even persisted across application instances, the object reference is stored in the map with no conversion. When students are on their local machines, the session store is associated with a single thread’s execution.

The Application class also provides access to the application and shared persistence layers. Using the same logic as the session persistence layer, the setters take an id argument and value to be stored. However, unlike the session and shared layers, the application layer modifies the id before storage. The application layer appends a unique application id to the object id. This prevents the shared layer and other applications from discovering other applications’ objects.

Objects stored in the application or shared layer are stored in the file system as XML files. The XML conversion process is performed by the Xstream library. Figure 5.5 shows the conversion of a student object into XML. First, Xstream obtains the class name and wraps the entire XML structure in it. Then each primitive field is converted into an XML tag. If a field refers to a more complex object, the object undergoes the same object to XML conversion. Certain built in JAVA classes are considered primitives and are persisted using simpler logic. For example, an instance of `java.util.Date` is converted into string form; container data structures are persisted by iterating over their elements and persisting them.

To retrieve objects out of the persistence layer, students used `getSharedObject` or `getApplicationObject` methods. Both of these methods take two parameters, the first is a unique key for the object and the second is the expected type of the object. The expected type is used to reconstitute the object into the correct type for the correct class.
Code:
```
public class UserProfile {
    private String email = "mjw87";
    private String name = "Mike";
    private String pictureUrl = null;
    private String password = "cheese";
}
```

XML:
```
<UserProfile fieldset='true'>
    <no-comparator/>
    <entry>
        <string>email</string>
        <string>mjw87</string>
    </entry>
    <entry>
        <string>name</string>
        <string>Mike</string>
    </entry>
    <entry>
        <string>pictureUrl</string>
        <null/>
    </entry>
    <entry>
        <string>password</string>
        <string>cheese</string>
    </entry>
</UserProfile>
```

Figure 5.5: Example of a Student Object and the Matching XML

loader. The object that is returned the first time is a newly created object. This object is cached by the Application class to be returned every time a student requests an object with that same ID. This caching was performed to prevent any synchronization issues from exposing themselves to the student. If a student wanted to obtain the most recent version of the object, they could use a `reloadSharedObject` method to wipe their changes out of the object and load the most current data into it.

After discussing the disadvantages of the application class, this design was scrapped. The application class created design restrictions that the students were forced to follow. If a
student wanted to create a multi-page app that accessed the persistence store, they would have to pass around a reference to their application object or work with two different caches into the persistence store. If a student created multiple application classes, then it became difficult to identify the most recent copy of an object with two different caching schemes.

In addition, requiring students to pass a class into each of the getters and setters was also difficult. Many of the students did not understand what class needed to be passed into the method to specify the return type or how it impacted the method’s operation. The JAVA doc for the methods also became complex because of their extensive use of generics for type checking.

After the application class was scrapped, a Map design was adopted. In this design, there is a Map for each of the persistence layers. Each map is instantiated with a type. This type determines the type of objects that can be retrieved and stored within the map. Using a type parameter at instantiation, reduces the type problems that students encountered and provided type checking on all other methods. Implementing the map interface also provided students with a variety of utility functions and more experience using map data structures.

Unlike the Application class, each persistence map only allows access to a single type of object. Figure 5.6 shows how the shared and application layer project a map on the persisted objects. In the figure, there are five objects in the persistence store. Four of the objects are UserProfile objects and the other is a GroupProfile object. If a student creates a SharedPersistenceMap with the type argument of UserProfile, only the four User Profile objects can be retrieved. The GroupProfile object will be returned as null. However, if the containsKey method is called on the map, true will be returned. Therefore, in this map there is a mapping to the GroupProfile but it is considered a mapping to null in the persistence map. Students cannot tell what type the mapping is because it may be an object that was defined using a class definition of another student.

In addition to changing the interface, the new persistence libraries also allow for separate directories for persisted objects to be stored. Instead of an ID structure that separated ap-
When implementing the methods to access objects stored in the application and shared persistence levels, the same caching problem was presented to us. When objects are retrieved from the persistence store, should the persistence map return a reference to the same object every time, or should a new object be constructed for every call? In the map interface specification, the get method’s documentation describes an algorithm that returns a single value mapped to a key. Therefore, if the value in the map is not modified, then the value stored in the map should not change.

The map we are creating; however, is a projection on persisted objects on the file system. Therefore, there is an element of synchronous activity executing on the persisted objects. At any time, another application running on CloudSpace might modify one of the objects stored persistently. If the map was a shared map, then each application would be able to modify the objects populating another applications program. Due to the novice nature of CloudSpace
users, this level of synchronous activity cannot be exposed to the user. Instead, we hide the synchronous activity by creating points in the code where concurrent modification is exposed to the student. For example, in the old Application class students would perform a method call to reload an object and discard their changes. If the persisted object had changed, this reload method would update their object with the most recent copy of the object. In our new infrastructure, we had to decide when students would be exposed to the synchronous activity using a map interface.

Two different map implementations were proposed. The first implementation uses a cached object for map retrieval for students. This way, the back end map returns a consistent object each time. If the changes in the current instance of the persisted object were to be loaded onto disk, a separate put method must be called. If a student calls ”get” after retrieving an object once, it is treated as a synchronization point with the persistent store. The student would receive a reference to the same object as before; however, all of the data inside of the object will be updated to match the data on disk.

This caching method is helpful when the retrieved object is stored outside of the map in a session store. In this case, if the student accesses the shared object in a different class, they will be working with the exact same object. However, using the get method to synchronize the object with the persistence store breaks a fundamental map contract with the user. Classes implementing a map should never modify the objects stored within. A call to get would have side effects on objects not contained within the transaction.

The second approach considered was to require that every call to get would return a new copy of the object. This approach drastically simplifies the implementation of the persistence store and simplifies the map algorithm that students must understand. When new objects are returned each time, students only need to understand that every get operation is completely separate and independent and the map is actually an object generator instead of a storage mechanism. Even though the logic is simple, students could easily lose track of the objects they are creating. With multiple copies of the same object existing in their application, they
might not know which one is the most current or what has been modified.

Each approach has its flaws. In the first approach, the get method may result in unexpected data changes for the student. In most map APIs a get command does not modify any data. However, creating new objects for every get request could make a massive number of the same profile. In this case, it could be difficult to figure out which object is the most current within a single application. The correct approach stems from what is least confusing to the student and most reliable to implement.

To determine the best approach, the different situations students would be using the persistence store was considered. There were two forms of execution for their code. In the first form, they would be executing their code locally in JUnit tests. These tests would be exercising their code in a sequential manner. When executing their code through unit tests, students will rightfully expect that no other program is accessing the shared persistence store. In this case, multiple calls to get should return equivalent objects. When testing equality; however, an additional burden of understanding the persistence layer would be imposed on them. They would need to either check the value of fields or check for equality, because that referential equality would not be guaranteed on CloudSpace. It would be good practice to assume that referential equality never holds true. Therefore, returning a new object each time might be best practice.

The context of their code execution changes when they upload their work to CloudSpace. On CloudSpace, each event execution is completely separate and equality checks are much rarer. In this case, if students do not cache their persisted objects there is a low chance that they will ever run into aliasing issues with their objects. In this case, aliasing might be preferred because CloudSpace could retain more control over a single aliased copy. CloudSpace could provide some powerful services to students; such as, automatic updating and automatic committing.

Another important factor is how simple the persistence layer is to explain to students. A caching scheme is more complex to teach then other schemes. While students might be
oblivious to many of the features, they are more likely to receive unexpected results from their persistence transactions. Alternatively, a scheme where a new object is returned every time is simple to explain to students. Additionally, if students properly use interfaces like the current user map, then they won’t need to worry about duplicate copies of the same object.

The latest version of the persistence layer implements a caching scheme for objects retrieved from the persistence layer. This approach is used because of the extensive problems students would have when writing JUnit tests. To an experienced developer, the difference between \texttt{==} and \texttt{.equals} is obvious. However, CS1 students only learn about the difference between the operators and may not be required to implement \texttt{equals} methods for their classes. This means that their JUnit tests could easily assert that a UserProfile returned from their application is equal to the UserProfile they retrieve directly from an instance of SharedPersistenceMap. Without a caching scheme, this assertion would fail and require students to implement a custom \texttt{equals} method.

It is also unlikely that students will ever attempt to retrieve the profiles again after the initial ”get”. After students request the object from the store, they are going to perform operations on it and push the changes back into the store. Because CloudSpace does not support transactions, providing students with multiple instances of the same object is pointless. CloudSpace provides a student with one instance of an object and that object’s data can change over time as other students modify it.
Chapter 6

Related Work

According to the Computing Research Association, enrollment in undergraduate Computer Science Programs has been falling since 2002. In 2002 there were approximately 23,000 computer science undergraduates in the United States. In 2008; however, enrollment had declined to about 12,000 undergraduates.\[6\] This decline in Computer Science enrollment could be attributed to the end of the start-up era of the Internet or the relocation of jobs to other countries.

6.1 Perception, Attitude, and Success

Due to the decline in Computer Science graduates, many members of the computer science education research community began investigating reasons for the enrollment decline. The biggest focus of this research has been on student’s attitudes and perception of the computer science field. In these studies researchers are attempting to identify what factors are being considered in student’s decisions to avoid computer science.

In a study published by the University of Berlin[16], researchers investigated what attitudes students have that serve as barriers or starting points for an education in computer science.
The paper describes visualizing computers as something to use and to design. Computer "users" simply see the computer as a tool to use. These users generally practice defensive learning where new knowledge is obtained as needed to continue using the computer. The second type of computer use is "design". These Users see computers as a tool they can use to practice expansive learning.

After identifying these two groups, the scientists interviewed students affiliated and non-affiliated with the computer science field. Non-affiliated students are students who do not consider themselves members of the computing community. The researchers found that these students generally viewed computers as something to "use". When interviewing affiliated students, the responses were of people that viewed computer use as "design". Many of these students mentioned pathway that was provided to them to transition from a computer "user" to a "designer".

In many other studies, researchers attempted to find specific attitudes that contribute to a decision to not major in computer science. In a study by Carter [8], the relationship between an apparent aptitude for computer science and the decision to not major in computer science was investigated. In this study, she discovered that many students do not understand what computer science is. Many students believe that computer science is about general computer maintenance. The biggest factors deterring students from studying computer science were the image that computer science consisted of working at a computer all day devoid of human contact.

In a study by Wilson and Shrock[17], 12 factors were identified that contributed to success in computer science courses. Of the twelve factors, the most important was the student’s comfort level. The second most important was their math background. Interestingly, previous programming experience was considered a non-factor to success.
6.2 Context in CS1

With an understanding of what might have caused the decline in computer science enrollment, many scientists have decided to shift existing curriculums to address these issues. In the past, computer science curriculums have been plagued by dry entry level computer science courses that put so much emphasis on basic concepts that students are unable to see the value of their education.[14] Changes in curriculum design include specialized courses for non majors, the addition of context to courses, and a more collaborative approach to computer science education.

In a study by Imberman and Zelikovitz[11], the use of robots was introduced to educate nonmajors about computer science. The course used Lego NXT robotics kits and a special programming environment to design algorithms for the robots to process. This course was viewed very positively by students because of the interesting context the course was presented in. However, many students struggled having to learn hardware, software, and programming assignments at the same time.

In multiple papers by Forte and Guzdial[9][10], the researchers propose a course called media computation. The media computation course attempts to use media manipulation as a context for elementary computer science concepts. In the media computation course, students manipulate sound, images, and video using the programming language python. In their python programs, students use specially designed libraries to modify media while learning basic computer science concepts.

The media computation class has been in place at Georgia Institute of Technology for multiple semesters. At Georgia Tech, the course has been very successful. In particular, students enjoy the creative and collaborative aspect of the course. However, some students fail to see the advantage of the time investment required to learn media concepts. In follow up interviews, many students found that the media computation aspect of the course peaked their interests; however, as they realize the power of computer science they wish for a more
6.3 Into the Cloud

CloudSpace attempts to take the findings discussed and create an environment for in major computer scientists. CloudSpace attempts to create an environment to address the attitudes and perceptions presented in section 6.1. The CloudSpace environment encourages students to collaborate and share the work they have created. CloudSpace provides them with a mechanism to share data created by their web applications and even share the web applications themselves with little investment by the student.

CloudSpace is unique because it attempts to provide context for students who have chosen to major in computer science. There are additional requirements in doing this that are not present in previous work such as the media computation or robotics courses. In those courses, the curriculums can be more flexible than a standard introduction to computer science course. In the courses that CloudSpace is used in, the basic computer science concepts must still be highly emphasized. This creates a unique challenge for CloudSpace to provide an environment that is transparent as possible for students to learn in.
Chapter 7

Results

7.1 Survey

To gauge the preliminary success of my work in enabling intro level computer science students to create engaging projects in their first computer science courses, surveys have been given to students after they complete assignments. The survey attempts to gauge the student’s attitude toward each of their completed assignments. The survey can be divided into three types of questions: engagement, frustration, and learning.

The first three questions attempts to measure how engaging, or fun, the students perceived the project. Question one asks the student to rate how engaging the assignment was on a scale of one to ten. The second question asks the student what was specifically engaging about the project. The third question asks the student what they feel would make the project more engaging. The next group of questions asked students about their perceived frustration levels. Question four asked students how frustrating the assignment was on a scale of one to ten. Question six asked students what aspect of the project was most frustrating to them. In this question, they chose from a list of eight options. The final question, question five, asked students to rate the amount they learned on a scale of one to ten.
The survey was given to three different subsets of students. Subset one was all of the students taking a second year computer science course. This course is a more standard computer science course that uses JAVA swing libraries. I will treat this data as a reference to more traditional GUI libraries. The second group of students are students in their first college level computer science course using the ZK framework to complete program assignments. The final group is all first year computer science students using the new infrastructure described in this paper.

The data collected from new CloudSpace users is limited. The spring 2011 semester is the first semester where students will be using the new infrastructure I have developed. Because of this, the data I have is only from two users in their first project using CloudSpace. I will examine this data the best I can in the later sections.

7.2 Engagement

Student’s engagement levels when completing their projects were measured to determine how "fun" students found each of their projects. In the control course where students used JAVA swing libraries, many of the students found the course very engaging with an average engagement score of 7.7. The computer science students using ZK without the new CloudSpace infrastructure reported an engagement rating of 6.8. The students using the new CloudSpace infrastructure reported an average engagement rating of 7. However, in the group of students using ZK and new CloudSpace infrastructure their initial project included no web development. After removing project one from the ZK students, the average engagement rating was still 6.8. There are only two users who answered the survey who have used the new CloudSpace infrastructure. Both of these users answered with a level 8 engagement rating.

While the new CloudSpace data is limited, it shows that students are responding well to the changes we have made to the CloudSpace environment. The biggest changes exposed
to students are the modification of the ZK data binding framework. The group of students using ZK showed a lower engagement rating than the Swing users. However, the data shows that the average engagement in Project 4 was two points below the mean for the rest of the projects. Without this project, the average engagement rating of the projects was 7.2.

Investigating the responses of how project 4 could have been more engaging, many of the students felt that ZK was holding them back from creating the web application they intended. Two of the users complained about a lack of debugging information. Other students found that the engaging aspects of ZK had begun to wear off. These comments lead me to think that students find the projects more interesting if they feel empowered by the tools they are using. They may lose their feeling of engagement if they feel like they are fighting with the tool. Engagement being a function of tool empowerment might also be the cause of high engagement ratings in the Swing JAVA course. In this course, students have already completed their first computer science course and have a better grasp on the tools they are using.

7.3 Frustration

The next survey group recorded student’s frustration levels when completing their required projects. When including the first project, the ZK based computer science course reported a frustration rating of 6.2. When the first project was removed, the course had a frustration rating of 7. In the swing JAVA course, the average frustration level was 6.2. Similar to the ZK computer science course, when the first project is removed the frustration level jumps to 7.

If we look at the new CloudSpace users, these users report a low level of frustration from the second project. Their average frustration rating is 3.5. Because the sample size is so small, it is hard to tell if this is representative of the entire class. Although, one of the students said the only frustrating aspect of the project was their IDE, BlueJ.
When I surveyed what aspects of the projects were the most frustrating part, swing JAVA users generally choose two categories about testing their solutions. The students working with ZK; however, were frustrated with an even distribution of the aspects of the project. The most prevalent frustration was the designing of classes and methods. Our preliminary results showed that the most frustrating aspect of the new CloudSpace environment was understanding the requirements and working with BlueJ. This was interesting because the limited responses from our new infrastructure suggest that students might be struggling less with CloudSpace.

### 7.4 Frustration vs Engagement

Finally, I looked at the perceived impact of frustration and engagement on the last category perceived learning. First I looked at the swing JAVA class. In this class, engagement had an impact on learning with a P value of .0016. However, frustration seemed to have no impact on how much the student learned. Similarly, the engagement of the ZK students had a positive impact on their learning with a P factor of .0016. While there are not enough responders from the new CloudSpace infrastructure, we might expect a high level of learning because of the high level of engagement. The two students that did respond had very divergent answers, one student rated his learning at ten and the second student rated his learning as a four. The second student; however, seemed to be knowledgeable about web development and might still not understand the full power of the CloudSpace infrastructure.

### 7.5 Conclusions

When looking at the responses to the survey questions, it is clear that students like the fact that in both classes, they were able to be creative and create user interfaces. In the ZK classes, students noted that they liked the progression of the assignments this progression
allowed them to see the entire process of creating a web application. However, many of
the ZK students noted how difficult the ZHTML was. It makes me wonder if ZHTML has
become too much of the focus of the course. Instead, it might be better to provide students
with a template for their ZHTML files. If this was done, it would be very similar to how
they develop their JAVA classes. A template could go a long way in reducing frustration
and changing the ZHTML development process into a creative outlet. If this was done, there
might be problems in later assignments where students are encouraged to develop something
unique for their application. In this step, they might not have enough ZHTML experience
to create an interface from scratch.

There was very little difference between students using the new ZK framework and those us-
ing JAVA’s swing framework. The only noticeable difference was a lower sense of engagement
from the students using the ZK framework. This is an interesting result; however, it might
be due to the student’s inability to express themselves through programming in their first
computer science course. Their responses indicated that feeling like the technology limited
them greatly hindered their engagement with the course.
Bibliography


