

University “Alexandru Ioan Cuza”, Iași  
Faculty of Computer Science  
Master: Software Systems Engineering

# Master Thesis

**Coordinating Teacher,**

*Lect. dr. Cristian Frăsinaru*

**Graduate Master,**

*Dabija Theodor Răzvan*

JULY, 2017

University “Alexandru Ioan Cuza”, Iași

Faculty of Computer Science

Master: Software Systems Engineering

# **Learning programming using drawing**

Promotion 2017

## Content

I. Introduction.....	6
II. Used technologies.....	7
1. Google Web Toolkit.....	7
1.1. Overview.....	7
1.2. Advantages.....	8
1.3. Disadvantages.....	9
1.4. Architecture.....	10
a) Java-to-Java compiler.....	10
b) JRE Emulation library.....	11
c) Google Web Toolkit web UI class library.....	12
1.5. Basic Concepts.....	13
a) Modules.....	13
b) Host Page.....	13
c) Cross-Browser Support.....	13
d) Different Modes for Running Google Web Toolkit	
Applications.....	13
1.6. Features.....	13
a) Widgets and Layout.....	13
b) Server Communication.....	13
c) Testing Support.....	13
d) Advanced Features.....	14
1.7. Application Layout.....	14
a) Module descriptor.....	14
b) Public resources.....	14
c) Client-side code.....	14

d) Server-side code.....	15
1.8 Component Model.....	15
a) Widgets.....	15
b) Panels.....	17
1.9. Style with CSS.....	18
a) Primary & Secondary Styles.....	19
1.10. Event Handling.....	19
1.11. Internationalization.....	19
a) Internationalization Techniques.....	19
1.12. UIBinder.....	21
1.13. Remote Procedure Calls.....	21
a) Communication.....	21
b) Components.....	22
c) Communication Work flow.....	22
1.14. History Class.....	23
1.15. Logging framework.....	23
 2. ANTLR4.....	25
2.1. Overview.....	25
2.2. Parse-Tree Listeners and Visitors.....	25
a) Parse-Tree Listeners.....	26
b) Parse-Tree Visitors.....	26
2.3. Creating a Grammar.....	26
2.4. Lexers Rules.....	27
2.5. Parser Rules.....	29
2.6. ANTLR Tool, Runtime and Generated Code.....	30

2.7. Testing the Generated Parser.....	32
2.8. Integrating a Generated Parser into a Java Program.....	32
III. GeoDraw.....	33
IV. Conclusions.....	41
V. Bibliography.....	43

## I. Introduction

E-learning is the interaction between the educational process and information technologies that cover a wide range of activities, from computer-assisted learning to online learning.

Orientation of the lines, angles, number of sides and the arrangement of the geometric figures generate different associations in the mind of those who look at them.

For example, the resemblance of a square with the image of a building, a circle with the image of a ball influences the way that the figures are perceived, and some of the features of the objects that they look like are transferred to them.

GeoDraw is an e-learning application for children who want to enter into programming world by interacting with a mini-programming language. This application is very easy to use by introducing an *action*, *object* and *methods* in the text area and see the result in a drawing area. With GeoDraw users can learn a small programming language and also create drawings using geometric figures as: circle, ellipse, square, rectangle, triangle and lines. With these geometric figures, the user can draw them, move them on the screen, change their color or dimension.

The scope of this application is to let the kids interact with a mini-programming language and by using their imagination to create drawings.

## **II. Used technologies**

### **1. Google Web Toolkit**

#### **1.1. Overview**

Google Web Toolkit is an open source web development framework that allows developers to build large scale and high performance web application while keeping them as easy-to-maintain. With Google Web Toolkit, we are able to write front end in Java, and it compiles our source code into highly optimized, browser-compliant JavaScript and HTML.

Browsers are capable of a lot these days, but to take full advantage of the browser you need to be an expert in JavaScript and three or more browsers and their quirks, and potentially have a lot of time on your hands. Alternatively, you could have a good set of tools to help you out.

Google Web Toolkit provides a set of tools for just that. It lets a developer write an AJAX application in the Java programming language, taking advantage of all the tools available for Java, and then cross-compile that code into highly optimized, plain old JavaScript. JavaScript that's not only optimized per browser, but can be served from any web server.

The source package structure is divided into client and server packages. At deploy time, Java files in the client directory will be cross-compiled into JavaScript, and run in our users browsers. Files in the server directory will be run as bytecode on the server. Because the client code is also modeled in Java, we have the option of using the same objects across both my server and my JavaScript client.

Google Web Toolkit provides higher-level abstraction on top of JavaScript, enabling developers to be more productive by coding against widgets and events that works across all major browsers. It's worth noting that we can also mix handwritten JavaScript into Google Web Toolkit source code easily.

Part of the development-time magic on Google Web Toolkit is the hosted mode browser. During development, we interact with our application in the hosted mode browser, which runs our Java byte-code on the back-end to render our user interface

in an embedded browser.

This has two benefits. First, because we're running Java byte-code, it means we can debug as we would any Java applications, setting breakpoints, inspecting variables, and stepping through code. Second, we can develop using the familiar edit-refresh-view programming model of JavaScript. We don't have to compile our Java source into JavaScript to see changes in the browser.

Google Web Toolkit provides an optimized Remote Procedure Call mechanism for Java, which means we don't have to worry about the complexity of low-level HTTP interactions. Alternatively, we can also use easily JSON to communicate with the server.

When we are ready to see what our application will look like in production, we compile our source code into JavaScript. Google Web Toolkit handles the browser implementation differences for us, and creates an optimized JavaScript file per major browser. The code is contained in the cache.html files. Browser-specific JavaScript means a user downloads only what they need. An Internet Explorer user doesn't download code needed for Firefox.

The compiled script is heavily optimized, including compiler optimizations like method in-lining, code size reduction, such as removal of comments and unused code, and of course, the final script is obfuscated and minified. If we want to inspect the final JavaScript in a human-readable form, simply compile with the “pretty” option selected. Just be sure to switch back to obfuscated before deploying, as this results in faster code and less JavaScript for the user to download.

## **1.2. Advantages**

- The learning curve is very slow for a Java developer;
- Debugging it's as easy as debugging any pure Java applications
- Developing Google Web Toolkit is quick thanks to error checking and syntax highlighting
- Easy to manage client history that means that Google Web Toolkit provides



back button support

- We can use both JUnit and Mockito for unit testing. Google Web Toolkit makes very easy to apply TDD (Test-driven development) or BDD (Behavior-driven development) patterns.
- Google Web Toolkit provides cross-browser support that means that our application will work similarly on the most recent versions of IE, Firefox, Chrome and Safari.
- The UI Binder framework is an excellent tool to build our applications as HTML pages with Google Web Toolkit widgets sprinkled throughout them
- Exchange Java object over HTTP between our client and server, it is straightforward using the Google Web Toolkit RPC framework

### **1.3. Disadvantages**

- Main Google recent projects are not longer developed with Google Web Toolkit
- Used to have a good community behind but it is decreasing quickly
- There is too much boilerplate code. Creating a simple table in Google Web Toolkit could take you more than a hundred lines.
- Google Web Toolkit is only useful for Java developers, if we are coming from a different background I don't think Google Web Toolkit will be helpful for us
- UI test are too slow. Although, if you apply correctly the Model-View-Presenter design pattern, you could minimize the use of GWTTestCase, and write fast JRE tests
- The compilation time from Java to JavaScript is very slow. This is my main problem with Google Web Toolkit; I found that the Google Web Toolkit compiler performance is lousy.

## 1.4. Architecture

Google Web Toolkit has three major components:

**a) Java-to-Javascript compiler:** the heart and most impressive part of Google Web Toolkit.

The responsibility of the Google Web Toolkit compiler is to convert our Java code into JavaScript code, in the same way the Java compiler compiles our Java code into byte-code. We compile our program by running the Java program *com.google.gwt.dev.GWTCompiler*, passing it the location of our module definition file and some other parameters. The module definition contains an entry point, which is a Java class.

The compiler starts by executing the class, which contains the entry point, following dependencies required to compile the Java code. The Google Web Toolkit compiler compile only what is being used. This is useful because it lets us develop a large library of tools and supporting components, and it includes only those that are used by the entry-point class.

At the compiler time we have to determine what will the resulting JavaScript looks like. Default style is *obfuscate*, that makes the JavaScript look like alphabet soup. This style mode is nearly impossible to decipher, which is a benefit for preventing code theft and the resulting JavaScript file are smaller.

Next style is *pretty*, which generates readable JavaScript. This compiled code snippets is derived from the original Java source code. Now we can see the methods code, but we can't tell what class the code is for.

The JavaScript code which is produced by the last style *detailed*, looks like the *pretty* style, but it has in the method name also the full class name.

The last two styles are typically used only during development so that JavaScript errors are easier to track back to the Java source code. For production, the favorable style is *obfuscated*, because it keeps the JavaScript file size down.

Another aspect about GWT compiler is that it compiles from Java source code, not compiled binaries. This means that all the Java classes we are using must be

available.

When we compile our code to JavaScript, the results for each browser type and target locale are saved in different file. Each file is meant to run on a specific browser type, version and locale. Google Web Toolkit supports the following browsers: Internet Explorer, Chrome, Firefox, Mozilla, Opera and Safari. When the application is loaded, a bootstrap script, pulls the correct file.

**b) JRE Emulation library:** provide a mapping of the Java Runtime Environment onto JavaScript. It has two parts: first part deals with Java language support and the differences between the Google Web Toolkit emulation and the real Java version, and the second part deals with what classes are emulated by Google Web Toolkit and can therefore be used as a developer.

### **Language Support:**

When developing applications we can use the full extent of the Java language, but there are some differences between Google Web Toolkit and Java compiler, in order to avoid some problems:

- **Long support:** for a *long* representation, we can't pass it from Java code to JavaScript Native Interface (JSNI) methods. If the *long* types remain in the context of our Java code, we can use it without any problem.
- **Exception handling:** there are two special cases where exceptions doesn't work similarly as in normal Java code. The *getStackTrace()* method doesn't return anything useful when it is running as JavaScript. The second case is the default exceptions such as *NullPointerException* and *OutOfMemoryError*. This type of exceptions are never thrown. We must perform explicit null checks.
- **Single-threaded:** because JavaScript interpreters are single-threaded, our application become single-threaded as well. All methods and keywords related to threading are ignored by the Google Web Toolkit compiler.
- **No reflection:** we can't use reflection to load classes dynamically. In order to generate one optimized JavaScript file for a browser, the compiler must know about each class and method that we use. If we were allowed to load classes dynamically this would be impossible.

## Emulated Classes:

We can use most of the Java language when we develop a Google Web Toolkit application. But when we are developing code that must be translated to JavaScript and then run in a browser not everything is possible. We can't write into a file and query to an SQL database, but we can use only the classes that Google Web Toolkit provides an emulation for. Even for this classes, we can use only that methods that are actually emulated. From packages *java.lang* and *java.util* we can use all the classes, along with a limited subset of the *java.io* and *java.sql*.

### ● **Differences in classes:**

- Regular expressions: Java and JavaScript have different syntax for regular expressions;
- Serialization: Java serialization is unavailable in Google Web Toolkit, but it provides it's own mechanism for serialization.

### ● **Convenient Classes:** some basic classes aren't available as part of Google Web Toolkit emulated classes. It has its own convenient classes:

- *com.google.gwt.i18n.client.DateTimeFormat* – replacement for *java.util.Date-  
TimeFormat*;
- *com.google.gwt.i18n.client.NumberFormat* – replacement for *java.util.-  
NumberFormat*;
- *com.google.gwt.i18n.client.Timer* – replacement for *java.util.Timer*.

### ● **Library Usage:**

Using external libraries such as log4j isn't permitted. We can use only classes that are tailor-made for Google Web Toolkit, because this classes include the source code as part of the library.

**c) Google Web Toolkit web UI class library:** Google Web Toolkit comes with a large set of *widgets* and *panels*.

## 1.5. Basic Concepts

**a) Modules:** Configuration of an application is called a module. The module configuration is provided in XML which contains in details the configuration for an application. It defines the application startup class.

**b) Host Page:** because the application must be loaded and bootstrapped by a browser, it needs to be embedded in a HTML page. This page can be a static one, but it can be also a dynamically generated page.

**c) Cross-Browser Support:** Google Web Toolkit offers support for major browsers as Internet Explorer, Firefox, Safari and Opera and let the developer to focus on the logic of the application.

**d) Different Modes for Running Google Web Toolkit Applications:** one major advantages is that we can reuse our existing Java best practices. For this, we need to run the code in both Java and JavaScript environments. There are two different modes in which we can run our application:

-*Hosted Mode:* is the mode where our Java code runs as Java inside a browser environment. Here we can set breakpoints and then just debug it as normal Java code.

-*Web Mode:* is the mode where the Java code is translated to JavaScript and runs inside a browser.

## 1.6. Features

**a) Widgets and Layout:** building a Rich Internet Application (RIA) is creating the user interface. Library provided by Google Web Toolkit consist of *widgets* and *layout* classes.

**b) Server Communication:** Google Web Toolkit provides an easy-to-use RPC communication mechanism between client and a Java server. It also includes support for XML and JSON back-end.

**c) Testing Support:** we can easily test the code. Google Web Toolkit provides its own testing support and we can also use the JUnit framework.

**d) Advanced Features:** JavaScript Native Interface, deferred binding, image bundles, localization, history support.

## 1.7. Application Layout

The project structure doesn't care how we lay out the directories in the application. It is very important to lay out the structure package correctly within our source directory because this relies to infer many things about our application.

A typical application consists in three mandatory parts and one optional:

**a) Module descriptor:** *module* is the name that is used for an individual configuration of the application. The configuration defined in the descriptor contains:

- *Inherited modules* – import statements for Google Web Toolkit applications;
- *Entry point-class* – details which classes serve as the entry point. In Google Web Toolkit this classes have to implement the *EntryPoint* interface. There can be more than one entry point classes and are optional. The real entry point of the application is the method *onModuleLoad()* provided by the *EntryPoint* interface.
- *Source path entries* – the descriptor module allows us to set the client-side code to other location
- *Public path entries* – allow us to handle public path items as source path entries
- *Deferred binding rules* – advanced settings that we can use to customize all kinds of aspects of our application. Usually they are not used by the developer directly.

**b) Public resources:** these are public files which include the HTML host page and other resources such as style sheets and images. This are packed as part of the application. The host page and resources need to be placed in the public resources directory. We can customize the location of these resource by adding an entry to the module descriptor using `<public path="path"/>`. When we compile the application into JavaScript, all the files that can be found on public path are copied to the module's output directory.

**c) Client-side code:** this is the Java code written for implementing the business logic of the application which is translated into JavaScript and run inside the browser.

Location of this resources can be configured using `<source path="path"/>`.

**d) Server-side code:** this is the optional part, the server side of our application, the remote back-end to our client-side application. If there is some processing required at back-end and our client-side interacts with the server, we should develop these components.

## 1.8. Component Model

Google Web Toolkit API provides all classes that we need to create complex web-based user interfaces. Every user interface have three main aspects:

- **UI elements:** the core visual elements that the user sees and interacts with. All in this component hierarchy are derived from the *UIObject* base class.
- **Layouts:** define how the UI elements are arranged on the screen
- **Behavior:** events which occur when the user interacts with UI elements.

*UIObject* Class is the superclass for all user-interface objects. It provides access to a wide range of DOM functionality without accessing the DOM directly. After this class, all widgets must inherit from the *Widget* class.

### a) Widgets

#### ● Basic Widgets

- *Label Widget:* is the simplest widget. The purpose of this is to display plain text. Can not be interpreted as HTML;
- *HTML Widget:* is a special label that supports HTML content. Uses a `<div>` element which cause it to be displayed with block layout;
- *Image Widget:* displays an image at a given URL. These widget has two modes: “unclipped” or “clipped”;
- *Anchor Widget:* represents a simple `<a>` element;

## ● Form Widgets

- *Button Widget*: represent a standard push button that the user can click. His normal use is to let users to trigger an action. Button enables registration of click listeners that will be called when the user clicked it;
- *PushButton Widget*: represents a standard push button with custom styling
- *ToggleButton Widget*: is a special button that allows the user to toggle between up and down states. When this button is clicked toggles its state. This widget is widely used in desktop applications;
- *CheckBox Widget*: represents a standard checkbox;
- *RadioButton Widget*: represents a mutually exclusive selection radio button;
- *ListBox Widget*: represents a list of choices to the user, either as a list box or as a drop-down list;
- *SuggestionBox Widget*: represents a text area or a text box which displays a pre-configured set of selections that match the user's input;
- *TextBox Widget*: represents a standard single-line text box;
- *PasswordTextBox*: represents a standard single-line text box that visually masks its input;
- *TextArea Widget*: represents a text box that allows to enter multiple lines of text;
- *RichTextArea Widget*: represents a rich text editor which allows complex styling and formatting;
- *FileUpload Widget*: wraps the HTML `<input type="file">` element. To be submitted to a server must be used with a *FormPanel*;
- *Hidden Widget*: represents a hidden field in an HTML form;
- *Hyperlink Widget*: this widget is used for internal links.

## ● Complex Widgets

- *Tree*: represents a standard hierarchical tree widget. It contains a hierarchy of *TreeItems* that the user can open, close or select;
- *MenuBar*: represents a standard menu bar which can contain many numbers of menu items and each menu item can fire a *Command* or open a cascaded menu bar;



- *DatePicker*: represents a standard Google Web Toolkit date picker;
- *CellTree*: represents a view of a tree;
- *CellList*: represents a single column list of cells;
- *CellTable*: represents a A tabular view that supports paging and columns;
- *CellBrowser*: represents a brow sable view of a tree where only a single node per level may be open at one time.

## **b) Panels**

### **● The RootPanel**

RootPanel serves as the top-level panel in the application. It provides the following three main methods:

- a) *RootPanel get()* – returns the panel that represent the body of HTML page;
- b) *RootPanel get(String id)* – HTML host page needs to contain an element that match the ID.
- c) *Element getBodyElement()* - returns a Java object representing the body DOM element of the HTML host page.

### **● Layout Panels**

Panels are the building blocks that allow us to position widgets where we need them. Panels provide the application's layout and visual organization.

- *FlowPanel*: represents a panel that formats its child widgets using the default HTML layout behavior;
- *HorizontalPanel*: represents a panel that lays all of its widgets out in a single horizontal column;
- *VerticalPanel*: represents a panel that lays all of its widgets out in a single vertical row;
- *HorizontalSplitPanel*: represents a panel that arranges two widgets in a single horizontal row and allows the user to interactively change the proportion of the width dedicated to each of two widgets;

- *VerticalSplitPanel*: represents a panel that arranges two widgets in a single vertical column and allows the user to interactively changes the proportion of the height dedicated to each of the two widgets;
- *FlexTable*: represents a flexible table that creates cells on demand;
- *Grid*: represents a rectangular grid that can contain text, HTML, or a child Widget within its cell;
- *DeckPanel*: represents a panel that displays all of its child widgets in a “deck”, where only one can be visible at a time;
- *DockPanel*: represents a panel that lays its child Widgets out “docked” at its outer edges;
- *HTMLPanel*: represents a panel that contains HTML, and which can attach child Widgets to identified elements within that HTML;
- *TabPanel*: represents a tabbed set of pages, each of which contains another widget;
- *Composite*: is a type of widget that can wrap another widget, hiding the wrapped widget's methods;
- *SimplePanel*: represents a base class for panels that contains only one widget;
- *ScrollPanel*: represents a simple panel that wraps its contents in a scrollable area;
- *FocusPanel*: represents a simple panel that makes its contents focusable, and have the ability to catch mouse and keyboard events;
- *FormPanel*: represents a panel that wraps its contents in an HTML <FORM> element;
- *PopupPanel*: represents a panel that can pop up over other widgets;
- *DialogBox*: represents a form of a pop up panel that has a caption area at the top and can be dragged by the user.

## 1.9. Style with CSS

Every component has its own style classification associated with it. This components rely on cascading style sheets for visual styling. The class name for each component, by default, is *gwt-`<classname>`*. We can easily create a style sheet for

our application without the need to assign style names for every component that we create. This helps us to make sure that our code remains clean.

#### **a) Primary & Secondary Styles**

*Primary style* name of a widget is by default the style name for that widget class. When we change the style name of a widget by add or remove style name, those styles are called *secondary styles*.

The final appearance of a widget is determined by the sum of all secondary styles of that widget plus its primary style. For setting the primary style for a widget we can use the method *setStylePrimaryName(String)*.

### **1.10. Event Handling**

In a web application event handling ties visual elements that the user sees to the functionality of the application. In Google Web Toolkit all event handlers have been extended from *EventHandler* interface and each handler has only one method with a single argument. This argument is an object associated with the event type which has a number of methods to manipulate the passed event object.

### **1.11. Internationalization**

Google Web Toolkit has a flexible set of tools to help us internationalize our application and libraries. It provides a variety of techniques to internationalize strings, typed values and classes.

#### **a) Internationalization Techniques**

- *Static string internationalization*

Is the most efficient way to localize our application for different locales in terms of runtime performance. In this technique we create tags that are matched up

with human readable strings at compile time. Mapping between tags and strings are created for all languages defined in the module.

The code is generated from standard Java properties files or annotations in the Java source. Google Web Toolkit supports this technique through three interfaces and a code generation library for generating the implementations of those interfaces.

- *Extending the Constants interface*: allows us to localize constants value in a safe manner, all resolved at compile time. We can also allow running lookup by key names with the *ConstantsWithLookup* interface.

- *Using the Message Interface*: allows us to substitute parameters into messages. The interface we create will contain a Java method with parameters matching those specified in the format string.

- *Dynamic string internationalization*

Was originally designed to allow existing i18n approaches to be quickly incorporated into Google Web Toolkit application. This technique allows us to look up localized strings defined in a host HTML page at runtime using string-based keys. It's a slower and larger technique, but it doesn't required the application code to be recompiled when messages are altered or the set of locales changes. If we want to integrate in our application an existing server-side localization system, then we need to use this technique.

- *Java Annotations*

Is used for specifying the default values for *Constants* or *Messages* interfaces. The advantage is that we can keep the value with the source, so when creating new methods or refactoring the interface it is easier to keep things up to date. Also we need to use annotations if we are using a custom key generator.

- *Localized Properties Files*

The traditional Java *.properties* files, that we are using for internationalization, may be placed into the same package as our main module class. Must be placed in the

same package where corresponding *Constants/Messages* subinterfaces definition files are.

### **1.12. UIBinder**

Allows us to build an application as HTML pages with Google Web Toolkits widgets sprinkled throughout them. It's a more natural way to build a UI rather doing it through code. This could make the application more efficient.

#### **a) Advantages:**

- helps productivity and maintainability;
- makes it easier to collaborate with UI designers who are more comfortable with XML, HTML and CSS than Java code;
- provides a gradual transition during development from HTML mocks to real, interactive UI;
- encourages a clean separation of the aesthetics of our UI from its programming behavior;
- performs thorough compile-time checking of cross-references Java source to XML and vice-versa;
- offers direct support for internationalization;
- encourages more efficient use of browser resources by making it convenient to use lightweight HTML elements rather than heavyweight widgets and panels.

### **1.13. Remote Procedure Calls**

#### **a) Communication**

Many Google Web Toolkit applications need to communicate informations between the browser client and the server. Client side code runs in browser and server side code runs in web server.

In nowadays the browsers have a JavaScript object called *XMLHttpRequest* that allows communication between the client and server without making a refresh on

the browser page. This is the basis for making the Remote Procedure Calls.

Google Web Toolkit provides two tools of the XMLHttpRequest: first one is *RequestBuilder* class, which is a wrapper around this object and GWT RPC, which let us send and receive Java objects.

The *RequestBuilder* class lets us to create a request that is submitted to the server and provides us access to the result from the server.

The GWT RPC mechanism lets us send Java objects between client and server with additional work on both sides. They are based on the Java servlet architecture. It's asynchronous and client is never blocked during communication.

Using this mechanism the Java objects are send directly between the client and the server and they are automatically serialized by the Google Web Toolkit framework.

## **b) Components**

- A remote service that runs on the server
- Client code to invoke that service
- Java data objects which will be passed between client and server

## **c) Communication Work flow**

For using the RPC communication we need to follow some steps:

- *Create a Serializable Model Class*: implements *Serializable* interface;
- *Create a Service Interface*: define an interface for service on client side that extends *RemoteService* class;
- *Create a Async Service Interface*: define an asynchronous interface to service on client side which will be used in the client code;
- *Create a Service Implementation Servlet class*: implements the interface at server side that extends *RemoteServiceServlet* class;
- *Update web.xml to include Servlet declaration*: include servlet declaration;

## 1.14. History Class

Applications developed with Google Web Toolkit are simple page running JavaScript and they don't contain lot of pages thus browser do not keep track of user interaction with application. To handle this situation Google Web Toolkit provides *History Mechanism*. It uses a token which is a simple string that the application can parse to return a particular state. The application save this token in browser's history as URL fragment.

To use this mechanism, we must embed an iframe in our host HTML page and than add token to browser history. When users uses back or forward button of browser, we'll retrieve the token and update our state.

## 1.15. Logging framework

It uses the same syntax and has the same behavior as server side logging code emulating *java.util.logging*. This is configured using the .gwt.xml files.

We can configure logging and particular handlers to be enabled or disabled, or change the default logging level.

### a) Types of Logger

They are organized in a tree structure, with the Root Logger at the root of the tree. Name of the logger determine the relationships between Parent/Child by using “.” to separate sections of the name.

### b) Log Handlers

Google Web Toolkit provides default handlers that will show the log entries made using logger:

- *SystemLogHandler*: these messages can only be seen in the DevMode window in Development Mode;
- *DevelopmentModeLogHandler*: these messages can only be seen as wheel in the DevMode window when the method *GWT.log* is called;
- *ConsoleLogHandler*: logs to the JavaScript console, used by Firebug Lite, Safari

and Chrome;

- *FirebugLogHandler*: logs to the firebug console;
- *PopupLogHandler*: when this handler is enabled, logs will be seen in a popup which is reside in the upper left hand corner of the application;
- *SimpleRemoteLogHandler*: send log messages to the server, and they will use the server side logging mechanism.



## 2. ANTLR4

### 2.1. Overview

ANTLR stands for Another Tool for Language Recognition. It is a parser generator that we can use to read, execute, process, or translate structured text or binary files. It is used mostly to build all sorts of languages, tools and frameworks. ANTLR takes as input a grammar and generates a parser that can automatically build parse trees and also generates a listener interface or a visitor interface that makes it easy to the recognition of phrases of interest. The parse trees are data structured representing how our grammar matches the input. It also generates automatically tree walkers. With tree walkers we can use it to visit the nodes of those nodes trees to execute application specific code.

ANTLR v4 has new capabilities that reduce the learning curve and make developing grammars and language applications much easier. One of this feature is that it accepts every grammar. There are no conflict or ambiguity warnings when the grammar is translated. The parser will always recognize the input that we give, no matter how complicated our grammar is.

For parsing, ANTLRv4 uses a new parsing technology called *Adaptive LL(\*)* or *ALL(\*)*, which is an extension to v3's *LL(\*)*. This new technology can always figure out how to recognize the sequence using appropriately weaving through the grammar.

Another feature is that it simplifies the grammar rules that are used to match syntactic structures like programming language arithmetic expressions. Automatically rewrites left-recursive rules into non-left-recursive equivalents. It has only one constraint, the left recursion must be direct, where rules immediately reference themselves.

### 2.2. Parse-Tree Listeners and Visitors

In its runtime library, ANTLR provides support for two tree-walking

mechanisms. By default, it generates a parse-tree listener interface which responds to the events triggered by the built-in tree walker. The listeners are like SAX document handler objects for XML parsers. The methods in a listeners are callbacks.

### **a) Parse-Tree Listeners**

ANTLR's runtime provides class *ParseTreeWalker*. We build a *ParseTreeListener* implementation that contains application-specific code and this code calls into a larger surrounding application.

ANTLR generates a subclass named *ParseTreeListener* specific to each grammar with *enter* and *exit* methods for each rule. When the walker encounters the node for a specific rule, it triggers the *enter()* method of that rule and passes it the *Context* class parse-tree node. After the walker visits all children of that node, it triggers the *exit()* method.

The beauty of this mechanism is that it's all automatic. We don't write a parse-tree walker, and our listener methods don't have to explicitly visit their children.

### **b) Parse-Tree Visitors**

If we want to control the walk through the parse-tree, we can use parse-tree visitors, which explicitly calls methods to visit children. Using the option *-visitor*, ANTLR generate a visitor interface from grammar with a visit method per rule.

Visitors walk parse trees by calling interface *ParseTreeVisitor's* *visit()* method on child nodes. This method is implemented in *AbstractParseTreeVisitor*.

The difference between a visitor and a listener, is that they just ask the visitors to visit the tree created by the parser.

## **2.3. Creating a Grammar**

ANTLR has two different approaches to define a grammar: top-down and bottom-up.

### *a) Top-down approaches*

Using this approach we start from the general organization of a file written in our language.

When we already know the format that we are designing a grammar for this is the best approach. We start by defining rules which represents the whole file. To represent the main section we could include other rules. After that we move from the most general rule, to the low-levels rules.

#### *b) Bottom-up approach*

This approach consist in focusing in the small elements at beginning: we start defining how the tokens are captured, basic expressions are define. Then we move to the top until we define the rule representing the whole file.

Using this approach we can focus on a small piece of the grammar, build tests for that, make sure that it works properly and then move on to the next bit. It has the advantage of starting with real code that is common for many languages.

The disadvantage of this approach is the fact that the parser is the thing we actually care about. If we start to build our grammar with the lexer, we will might end doing some refactoring because we don't know how the rest of the program will work.

## **2.4. Lexers Rules**

Lexers are also known as tokenizers. They are the base of the grammar. A lexer takes an individual character and transforms it into a token, and then the parser uses this tokens to create the logic structure of our grammar.

For creating a lexer grammar we need to compose some lexer rules, which can be broken into multiple modes. With this modes we can split a single lexer grammar into multiple sublexers. A lexer returns only those token that matched by rules from the current mode.

Lexer rules cannot have arguments, return values or local variables. They must begin with an uppercase letter, to distinguished them from parser rule names.

First we need to create a grammar file. This file will contain any text that we

want to parse and attempt to match the string to a number of lexer tokens. It contains human-readable text appended with “.g4”. For the basic level it contains only simple key-value pairs of different tokens.

When we feed a string into our grammar, ANTLR will look at those strings and try to match them with Lexer rules.

```
/*
 * Lexer Rules
 */

//ActionKeywords

DRAW : 'draw';
MOVE : 'move';
FILL : 'fill';
DELETE : 'delete';
REMOVE : 'remove';

//ObjectKeywords

CIRCLE : 'circle';
SQUARE : 'square';
RECTANGLE : 'rectangle';
TRIANGLE : 'triangle';
ELLIPSE : 'ellipse';
LINE : 'line';

//MethodKeywords

DIMENSION : 'dimension';
POSITION : 'position';
COLOR : 'color';
LEFT : 'left';
RIGHT : 'right';
UP : 'up';
DOWN : 'down';
INSIDE : 'inside';
BORDER : 'border';

//Separators

LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
RBRACE : '}';
SEMI : ',';
COMMA : ';';

//Identifiers

ID : [a-zA-Z0-9]+ ;
```

```
//Whitespace
NEWLINE : '\n';
WS : [ \t\r]+ -> skip;
```

## 2.5. Parser Rules

Parser consist of a set of parser rules in either a parser or a combined grammar. To launch a parser, a Java application is invoking the rule function, generated by ANTLR, starting with the top rule. A simple rule is a rule name followed by a single alternative terminated with a semicolon.

Rules can also have alternatives separated by the operator “|”. Alternatives are a list of rule elements. To get more precise parse-tree listener events we can label the alternatives of a rule by using the operator “#”. This alternatives labels doesn't have to be placed at the end of the line and doesn't matter if we have a whitespace after the “#” operator.

ANTLR generates a rule context class definition for each label from our grammar. Each generated class contains *enter()* and *exit()* methods associated with each labeled alternative. The parameters to those methods are specific to alternatives.

We can reuse the same label on multiple alternatives for the parser-tree to know that should trigger the same event for those alternatives.

ANTLR generates methods to access the rule context objects associated with each rule reference. For rules that have a single rule reference, it generates a method with no arguments. When there is more than a single reference to a rule, provides support to access context object. It will generate a method with an index to access the *i*th element from a method to get cotext to all references for that rule.

The entry parsing rule for my grammar will be the *drawSentence* parsing rule.

```
/*
 * Parser Rules
 */

drawSentence: action objects '{' (methods)* '}';

action
    : 'draw'
```

```

| 'move'
| 'change'
| 'fill'
| 'delete'
;

objects
: object arguments
;

object
: 'circle'
| 'square'
| 'rectangle'
| 'triangle'
| 'ellipse'
| 'line'
;

arguments
: '(' ID SEMI* (ID)* ')'
;

methods
: NEWLINE methodName arguments SEMI
;

methodName
: 'dimension'
| 'position'
| 'color'
| 'left'
| 'right'
| 'up'
| 'down'
| 'inside'
| 'border'
;

```

## 2.6. ANTLR Tool, Runtime and Generated Code

For getting started to create a grammar we need to have ANTLR.jar. In this jar file we have two components: the ANTLR tool and the ANTLR runtime API. When we run a grammar, ANTLR generates a parser and a lexer, that recognizes sentences in the language described by our grammar. The role of a lexer is to break up an input stream of characters into tokens and passes them to a parser that checks the syntax. The ANTLR runtime is a library of classes and methods which we need to generate code such as *Parser*, *Lexer* and *Token*.

From my grammar *NewDrawGrammar.g4*, ANTLR generates lots of files:

a) *NewDrawGrammarParser.java*: contains the parser class definition specific to my

grammar *NewDrawGrammar* that recognizes my language syntax.

`public class NewDrawGrammarParser extends Parser { ...}`, contains a method for each rule in the grammar and support code.

b) *NewDrawGrammarLexer.java*: ANTLR automatically extracts a separate parser and lexer specification from our grammar. This file contains the lexer class definition. After analyzing all the lexical rules and the grammar literals, ANTLR generates this file. The lexer tokenizes the input and breaks it into vocabulary symbols.

`public class NewDrawGrammarLexer extends Lexer { ...}`

c) *NewDrawGrammar.tokens*: For each token I define, ANTLR assigns a token type number and save this values in this file. We need this file when we want to split a large grammar into smaller grammars so that ANTLR can synchronize all the tokens type numbers.

d) *NewDrawGrammarListener.java*, *NewDrawGrammarBaseListener.java*: parsers builds a tree from the input. A tree walker can fire callbacks to a listener object that we provide. *NewDrawGrammarListener* is the interface that describes the callbacks I can implement. *NewDrawGrammarBaseListener* represents a set of empty default implementations. With this class I can override just the callbacks that I'm interested in.

The only file that I am interested for listening is *NewDrawGrammarBaseListener.java*. This class has *entry()* and *exit()* methods for each of my parse rule. We simply inherit from this base class and override the *enterDrawSentence(DrawSentenceContext ctx)* method. This allow us to capture every time the parser hits the *drawSentence* rule. From here we get a *DrawSentenceContext* object and we can access the text within the drawSentence rule by simply calling *ctx.getText()*.

```
public class DrawListener extends NewDrawGrammarBaseListener {
    @Override
    public void enterDrawSentence(DrawSentenceContext ctx) {
        String object = ctx.action().getText();
    }
}
```

}

## 2.7. Testing the Generated Parser

After we run ANTLR on our grammar, we need to compile the generated Java source code. We just do that by compiling everything that we have in the `/tmp/array` directory. If we get a `ClassNotFoundException` error it means that we didn't set the Java CLASSPATH correctly.

For testing the grammar, we can use the *TestRig* via alias *grun* with the command: “*grun NewDrawGrammar drawSentence -tokens*”. Each line of the output represents a single token and we can see everything we know about that token.

We also ask for the parse tree with the option *-tree*: “*grun NewDrawGrammar drawSentence -tree*”. This command prints out our parse-tree in LISP-like text form. Or, we can use the option *-gui* if we want to visualize the tree in a dialog box: “*grun NewDrawGrammar drawSentence -gui*”.

## 2.8. Integrating a Generated Parser into a Java Program

After we have generated our grammar, we can integrate the code into our application. For that we need the entry point. Java code that tells ANTLR to lex the string, parse it, walk it and attach our listener to it:

```
DrawListener listener = new DrawListener(canvas);
String line = textArea.getText();
ANTLRInputStream input = new ANTLRInputStream(line);
NewDrawGrammarLexer lexer = new NewDrawGrammarLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
NewDrawGrammarParser parser = new NewDrawGrammarParser(tokens);
ParseTree tree = parser.drawSentence();
ParseTreeWalker walker = new ParseTreeWalker();
walker.walk(listener, tree);
```

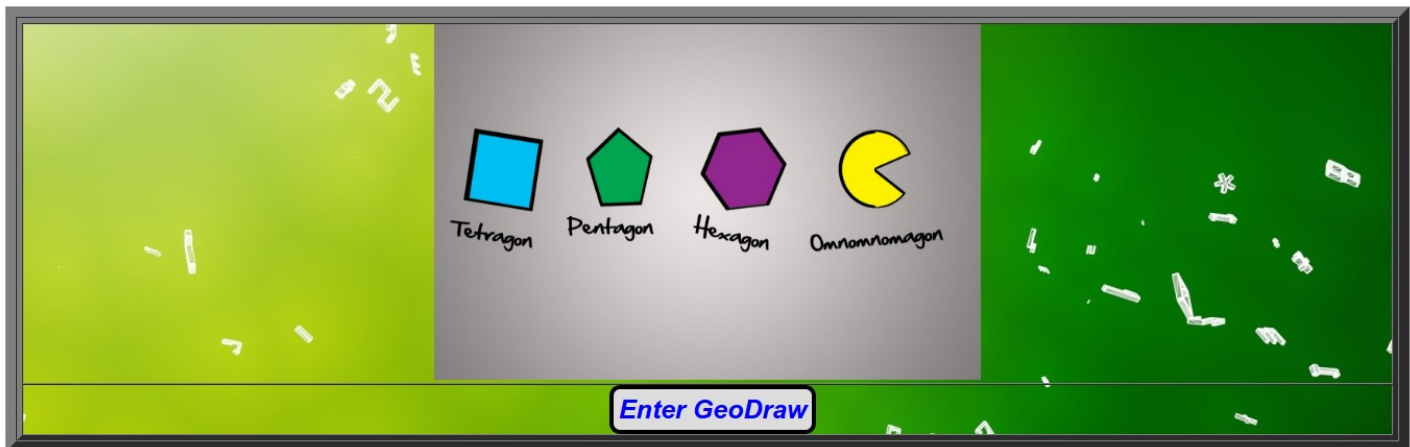
The program uses a number of classes as *CommonTokenStream* and *ParseTree* from ANTLR's runtime library.



### III. GeoDraw Application

#### First Page

When we run the application, it will open a page where we found a button in the middle of the page. After we click that button we enter in my application, named **GeoDraw**.



#### Main Page

In the main page we have the possibility to create a new account or if we have already an account to sign in. In the left we have a Text Area where we can write some commands which will be shown in the Drawing Area (if the commands are written correctly). We will see some example later.



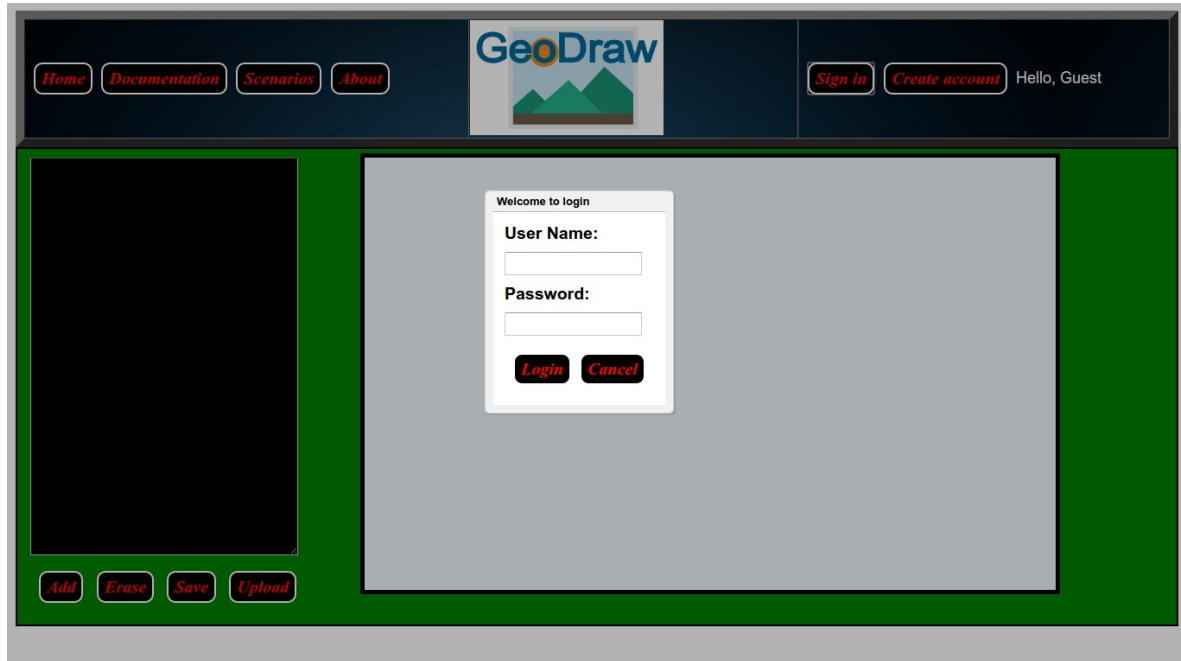
## Creating account

For creating an account the user must click on the button “Create account” in the right corner of the screen. By clicking this button will appear an info box where the user must complete all thields to complete the registration process.

The screenshot shows the GeoDraw web application interface with a registration form overlay. The header and main content area are the same as in the previous screenshot. The registration form is a white box with a grey border, titled "Welcome to register". It contains four input fields: "First name:", "Last name:", "User name:", and "Password:". Below the input fields are two buttons: "Submit" and "Cancel".

## Sign in

After the user completes the registration, he can sign in with his user name and password. If he isn't logged in, the user can't save his drawings.



The screenshot shows the GeoDraw web application interface. At the top, there is a dark blue navigation bar with links for Home, Documentation, Scenarios, and About. The GeoDraw logo is centered in the bar. On the right side of the bar, there are buttons for Sign in and Create account, followed by the text "Hello, Guest". The main content area has a green background. On the left, there is a dark gray rectangular area. Below it, there are buttons for Add, Erase, Save, and Upload. In the center of the main area, there is a white login form titled "Welcome to login". The form contains fields for "User Name:" and "Password:", and buttons for "Login" and "Cancel".

## Save draw

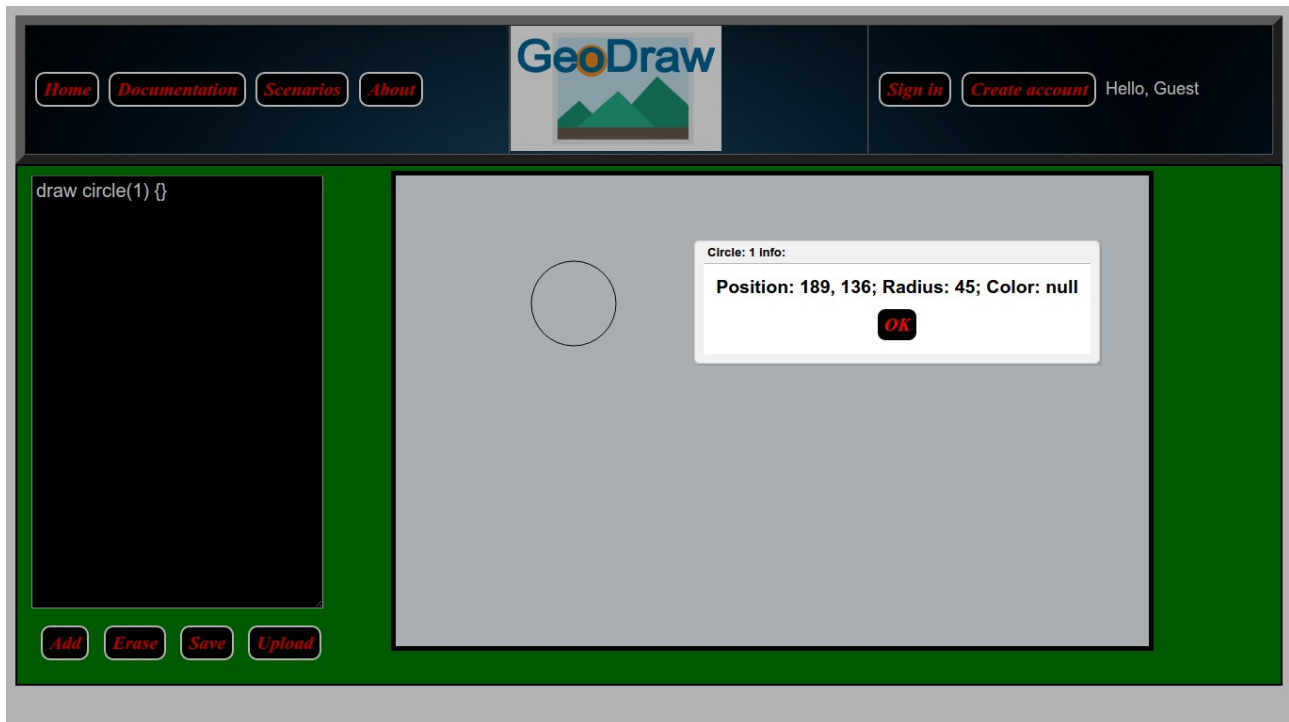
In the left down corner we have the save button. To save a drawing the user must be logged in and title is draw.



The screenshot shows the GeoDraw web application interface with the "Save drawing" modal open. The navigation bar and main layout are the same as in the previous screenshot. The "Save drawing" modal is a white box in the center of the main area. It contains a field labeled "Enter a title:" and buttons for "Save" and "Cancel".

## Draw circle random

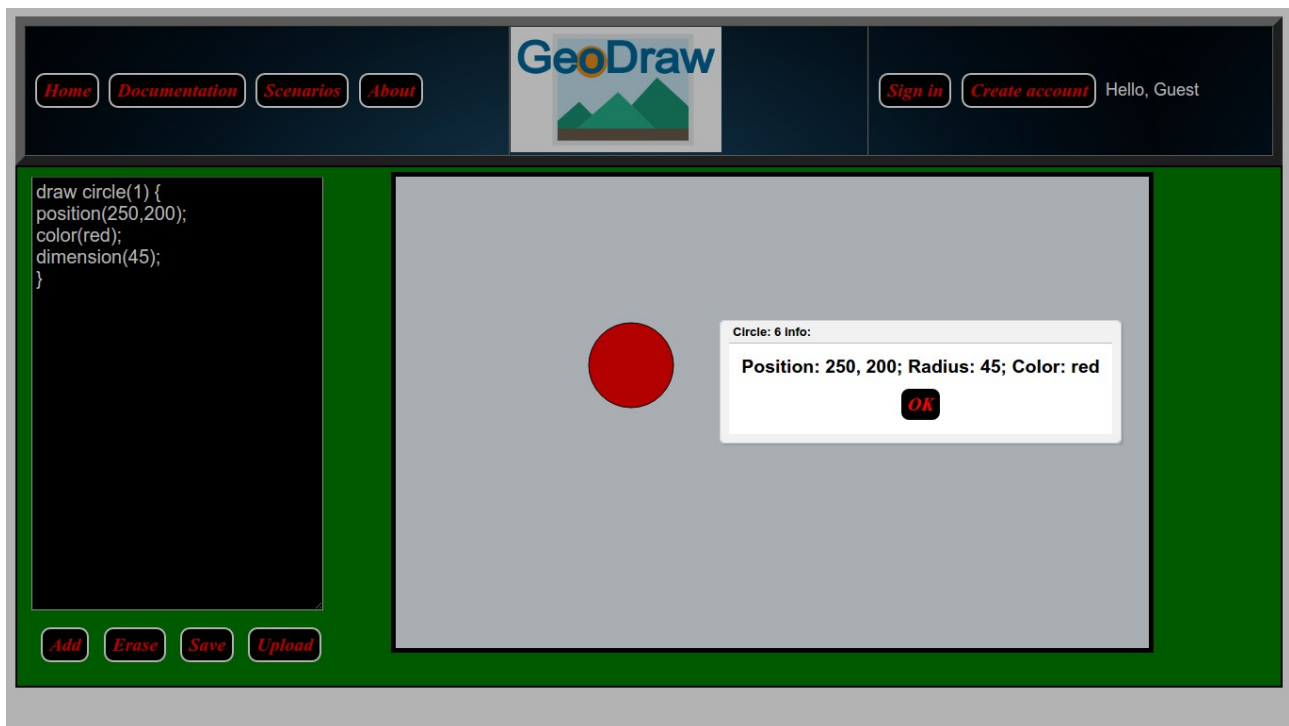
For drawing a simple circle, the user must enter the action “*draw*”, followed by an object that he wants “*circle*” and a parameters in brackets to specify how many shapes of that kind he wants to draw. By clicking a shape the user gets information about that.



## Draw a circle with all methods

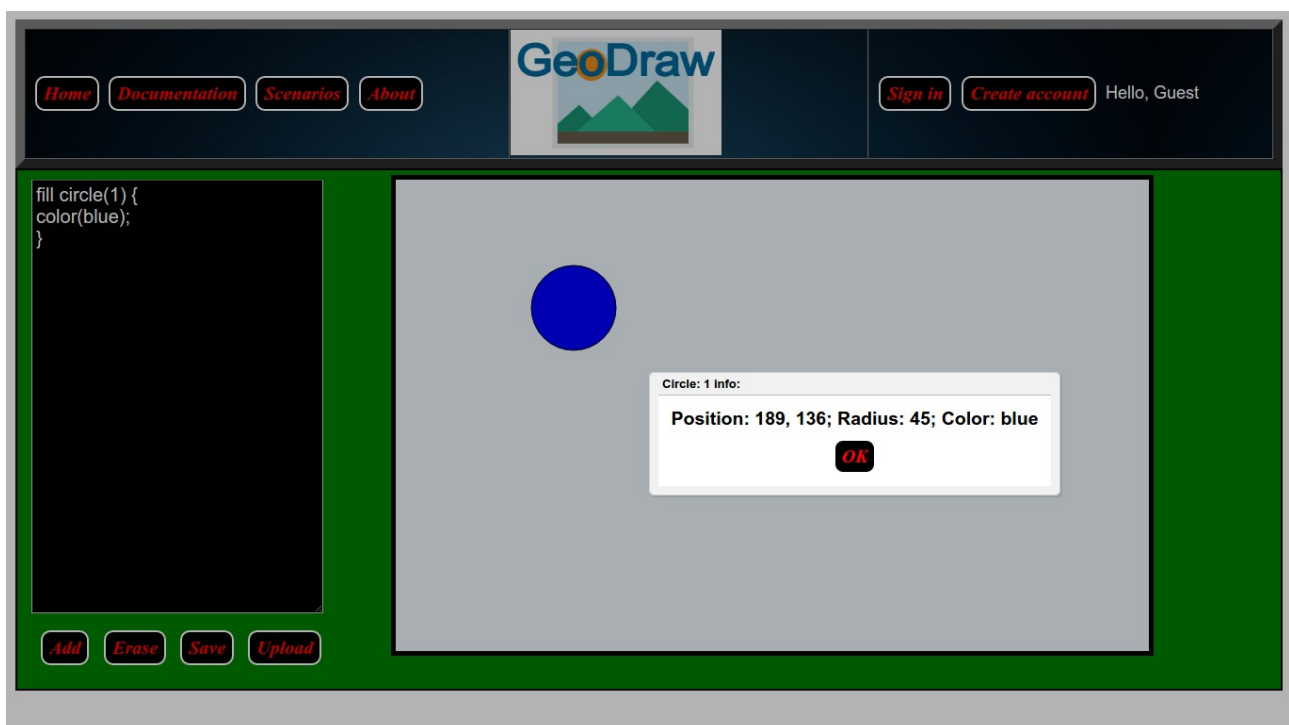
The user can also draw a complex shape by adding some methods:

- *position* method, which takes 2 arguments represent the coordinates (x,y) where the user wants to draw that shape;
- *color* method let the user to choose the color of that shape;
- *dimension* method represent the size of the shape. Depending of what shape the user want to draw, this method may have one, two or three arguments



## Fill circle with color

If the user want to fill the color on a empty shape, he can do that with the action “fill”, followed by the type of the shape which takes as argument the number of that shape, and the method *color* with the argument of the color he wants.



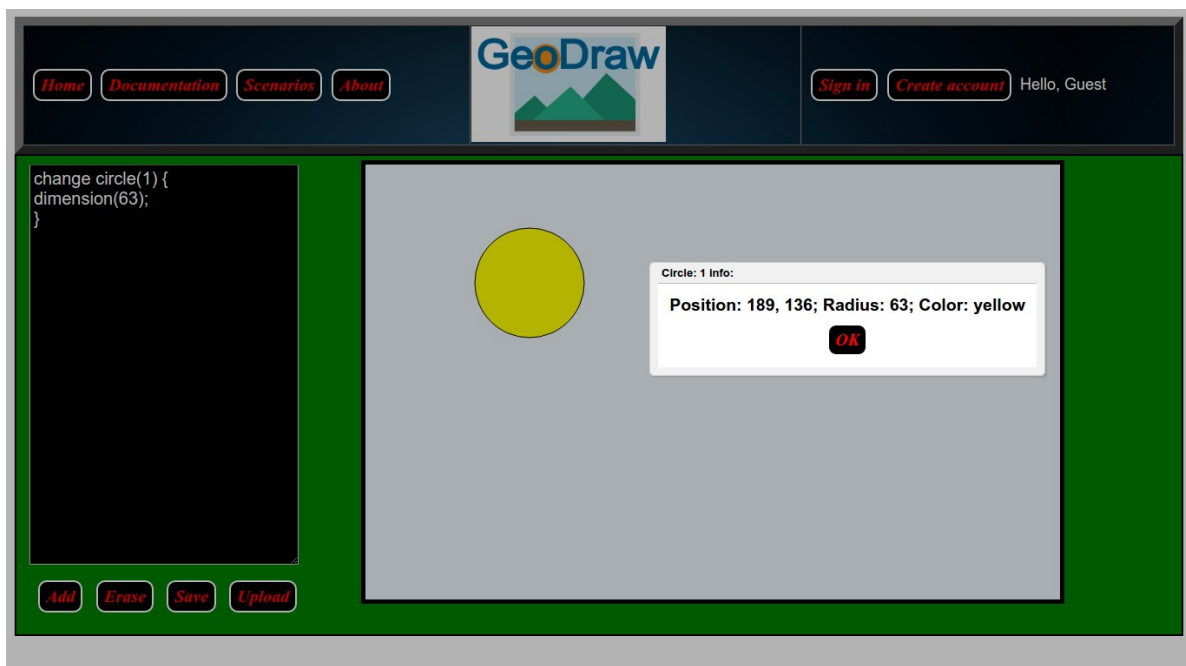
## Change circle's color

The difference between the fill and change method is that this method the user can use only when the shape has already a color. To use this method he just change the action name “change”.



## Change circle's dimension

The user can also modify any shape by using the action “change”, followed by the type of shape that he wants to change and the method dimension which takes as argument the parameter/s that he wants to change.



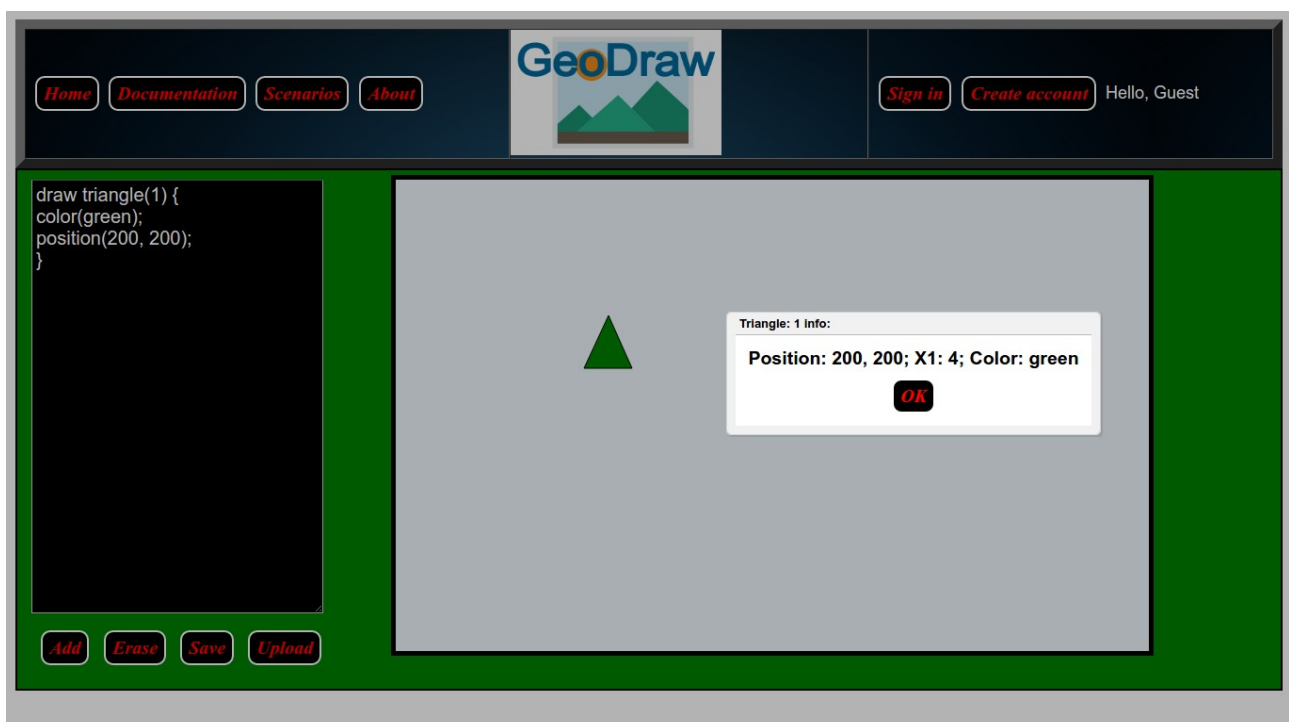
## Move circle right

Using the action “*move*”, user can move the shape on the canvas to right, left, up or down:



## Draw other shapes

Users can also draw: squares, rectangles, ellipses, triangles or lines:



## Example:



To make this draw the user can use the following commands:

1. draw circle(1) {color(yellow); position(60, 60); dimension(40);}
2. draw square(1) {color(aqua); position(300,320); dimension(180);}
3. draw triangle(1) {color(brown); dimension(180,180); position(300,319);}
4. draw square(1) {color(red); position(320,350); dimension(40);}
5. draw square(1) {color(red); position(420,350); dimension(40);}
6. draw rectangle(1) {color(grey); position(370,420); dimension(40,80);}
7. draw rectangle(1) {color(green); position(480,460); dimension(150,40);}
8. draw rectangle(1) {color(green); position(150,460); dimension(150,40);},

or he can draw random shapes, followed by other actions as: ***fill, change, move*** to put all the shapes together and create it.



## **Conclusions**

GeoDraw application development required the use of many practical and theoretical knowledge learned in two years of master, such as: communications protocols, object-oriented programming and design, data structures and databases. In addition, it was necessary to gain new theoretical knowledge, and also to learn the use of new technologies: Google Web Toolkit and ANTLR4.

## **Directions for the future:**

For the future there are several directions in which the application can be developed:

- Introducing complex geometric figures;
- Increase grammar of the application;
- Adding new methods such as rotating a shape to certain degrees, duplicating a shape;
- Visualize shapes in 3D ;
- Create scenarios.

## **Abbreviations**

**ANTLR** – *Another Tool for Language Recognition*

**API** – *Application Programming Interface*

**BDD** – *Behavior-driven development*

**CSS** – *Cascading Style Sheets*

**GWT** – *Google Web Toolkit*

**HTML** – *HyperText Markup Language*

**JRE** – *Java Runtime Environment*

**JSNI** – *JavaScript Native Interface*

**JSON** – *JavaScript Object Notation*

**RIA** – *Rich Internet Application*

**RPC** – *Remote Procedure Call*

**TDD** – *Test-driven development*

**UI** – *User Interface*

**XML** – *eXtensible Markup Language*

## Bibliography

1. Google Web Toolkit – tutorialspoint, [www.tutorialspoint.com](http://www.tutorialspoint.com)
2. Gwt in Practice – *Robert Cooper, Charles Collins*, Ed. Manning
3. GWT in Action – Easy AJAX with the Google Web Toolkit – *Robert Hanson, Adam Tacy*, Ed. Manning
4. Beginning Google Web Toolkit from Novice to Professional – *Bram Smeets, Uri Boness, Roald Bankras*
5. <http://www.gwtproject.org/>
6. <https://github.com/gwtproject/gwt>
7. <https://www.toptal.com/front-end/javascript-front-ends-in-java-with-gwt>
8. <http://www.vogella.com/tutorials/GWT/article.html>
9. The Definitive ANTLR4 Reference – *Terence Parr*
10. <http://wwwantlr.org/>
11. <https://github.com/antlr/antlr4>
12. <https://hackernoon.com/a-quick-intro-to-antlr4-5f4f35719823>
13. <http://progur.com/2016/09/how-to-create-language-using-antlr4.html>
14. [http://jakubdziworski.github.io/java/2016/04/01/antlr\\_visitor\\_vs\\_listener.html](http://jakubdziworski.github.io/java/2016/04/01/antlr_visitor_vs_listener.html)
15. <https://blog.knoldus.com/2016/04/29/testing-grammar-using-antlr4-testrig-grun/>
16. <http://www.theendian.com/blog/antlr-4-lexer-parser-and-listener-with-example-grammar/>