JDuck: Building a Software Engineering Tool in Java as a CS2 Project

Michael Godfrey
University of Waterloo
email: migod@plg.uwaterloo.ca

Dan Grossman
Cornell University
email: danieljg@cs.cornell.edu

Abstract

This paper describes our experiences in having students build a software engineering tool as a course project in a CS2 course. The tool, which we called JDuck (Java Documenter of Code, oK), was modelled on the `javadoc` tool that is part of Sun Microsystem's standard Java Development Kit (JDK). That is, a working version of JDuck would be able to read in Java source code and generate HTML files that summarize the basic structure of the provided classes. We discuss how we set up the project, what we think the students learned, what they told us they learned, and what we would do differently next time.

1 Introduction

One of the challenges in teaching a CS2 course is designing assignments that the students will find both intellectually compelling and fun. Our CS2 course at Cornell University, CS211, has the additional feature that it is required for all engineering undergraduates. (Cornell University has two distinct CS2 courses: CS211, which is taught in Java, and CS212, which is taught in a functional style using a language such as Noodle or ML. This latter course is intended primarily for CS majors.) Consequently, the majority of the class members are not CS majors, and some of these students can be difficult to reach and to motivate. We very much wanted all of the students to come out of this course with both a solid grounding in the material as well as a real appreciation of computer science. We decided that the best approach would be to have the students complete a non-trivial course project that they would find interesting.

1.1 Goals

We had several goals in mind for the project; in particular, we felt that it should

- serve to re-enforce the lecture material of object-oriented programming and introductory data structures,
- be non-trivial in size and scope,
- require students to learn how to function within a team,
- be constrained enough to be doable in an obvious manner,
- employ a staged delivery approach with obvious milestones, so that students could receive partial credit if they failed to complete the project,
- be flexible enough for students to express some creativity, and
- accomplish a real and useful task.

2 JDuck: A Simple Software Engineering Tool

A simple software engineering tool seemed to be a good candidate for the project. We decided to ask the students to create a simplified version of the tool `javadoc` [4], which is part of Sun Microsystem's standard Java Development Kit (JDK) package [3]. `javadoc` examines Java source class definitions, extracts information about the classes and their interrelationships, and creates a set of HTML pages that a naive user may browse as a reference document. For example, given a set of classes written in Java, `javadoc` will create a web page for each class that summarizes:

- the variables, methods, and constructors defined by that class (including precise syntax and comments extracted from the source code), and
- which classes/interfaces/packages the class inherits from, implements, imports, or otherwise uses.

For ease of browsing, `javadoc` structures these web page summaries in a uniform manner, and create hypertext links between referenced entities that are defined in other classes.

We designed our proposed tool, which we called JDuck (Java Documenter of Code, oK), as a simplified version of `javadoc`. The basic idea would be the same, but we would provide some of the system, and

we would allow the students to make some simplifying assumptions about Java syntax.

3 Overview of the Project

3.1 Preparation

The infrastructure for the JDuck project required significant preparation. One teaching assistant (TA) created a special-purpose lexer for Java using the JLex tool from Princeton University [1]. Another TA created a simplified grammar for Java. We also created a candidate solution in advance, and began to generate interesting test scripts for use in evaluating the students' submissions later on.

We tried to prepare the students for the task during the course. For example, one of the early assignments required students to process a set of simple text commands, where each command might take different numbers and kinds of arguments. This gave the students some preliminary experience in parsing and the idea of how the current state of a computation can influence the next action taken.

We also had the TAs give several tutorials on aspects of the project: there was one tutorial on simple scanning and parsing, and another on on basic HTML, visual design, and writing simple Java programs that produced HTML fragments.

3.2 The Project Overview

Once the project was ready to be presented to the students, we distributed the prepared handout and discussed it in lecture. We described to the students an overall approach that we felt would lead to success. In essence, the desired approach used student-written parsing methods to generate an abstract syntax tree (AST) containing the appropriate information. These parsing methods use the provided lexer to retrieve tokens from the input. Nodes in the AST are student-defined objects which have methods to generate HTML fragments describing themselves. Thus, a solution would generate a single HTML file describing a (restricted) Java class by performing the following steps:

- Take as input the name of a Java class. Create a lexer object with the appropriate input file name. Create an appropriately-named file for output.
- Parse the input, creating an AST.
- Generate an HTML summary of the class by "walking" the AST, based on the output criteria set forth in the project handout.

We gave students a simple, object-oriented interface to the lexer: a lexer object is created by passing a file name to the constructor, and a lexer object has a `nextToken()` method which returns an object which has methods `getText()` and `getTokenType()`. We also held a tutorial session to teach students how to use the lexer and reason about input-processing. We wrote several sample programs, such as one to "return the text of the third identifier after the fourth semicolon, or the empty string if the file has fewer than four semicolons."

3.3 The Grammar

We provided students a pseudo-grammar for the input format specification. The entire grammar is given below (Fig. 1). CAPS denotes an identifier, $[a]$ denotes that an item is optional, $a|b$ denotes alternatives, and $a*$ denotes a Kleene closure. All other terminals are literals and have distinct token types. Method bodies are not described by the grammar, as we told students to ignore them. Several aspects of this Java subset and the lexer deserve notice:

- Some syntactically-awkward elements of Java, such as arrays, have been omitted.
- The lexer removes all comments, except for the special "`//Variables`" and "`//Methods`" comments.
- All variable declarations precede all method declarations. This simplifies disambiguation considerably.
- The order of declaration modifiers is fixed for the sake of simplicity.
- The grammar can be parsed without any "look-ahead," except for one place. When the first identifier in a method declaration is encountered, one cannot determine whether it is the return type of a regular method or the name of a constructor until the next token is read. Students had to distinguish between methods and constructors based on this next token.
- Method bodies are ignored. To skip a method body, we instructed students to "count braces" until the number of "{" equaled the number of "}." This was as easy as it sounds; we explicitly told the students that the lexer was intelligent enough to correctly handle braces that occurred in unusual contexts, such as within comments or explicit character strings.

The project handout explained in detail how to interpret the grammar. We spent part of one lecture teaching students the basic ideas of parsing the grammar. We taught them to "determine which tokens might come next, and for each possibility record the necessary information and determine which tokens might come after that, and so on." Although we hinted which control constructs correspond to grammar constructs (`while` for $*$, for example), we were deliberately vague and conceptual.

3.4 Output Specification

Most of the output specifications were straightforward; for example, we required that students provide a hyperlink to the JDuck page of a class's inheritance parent. A more interesting requirement was that class members (*i.e.,* variables and methods) should be grouped by *kind* in the output, *e.g.,* static variables should be grouped

```
File      =  [import LIBRARY;]*
             [package PACKAGE;]
             [public] [abstract] [final] class CLASS [extends SUPERCLASS]
             [implements INTERFACE [,INTERFACE]*]
             {
                 //Variables
                 Variables*
                 //Methods
                 Method*
             }

Variables =  [public | private | protected] [static] [final] TYPE VAR [, VAR]*;

Method    =  [public | private | protected] [static] [abstract] [final]
             [TYPE] METHOD ([TYPE PARAM [, TYPE PARAM]*]) {...}
```

Figure 1: Grammar of the Java subset the students used.

and listed separately from instance variables. This implicitly forced the students to generate an intermediate representation of the class and its members rather than allowing the members to be completely processed "on the fly" in the ordered encountered in the class definition. In so doing, the students gained experience in designing and using an interesting tree-like data structure, which was one of the main goals of the project.

The required order of presentation for class members in the output was

1. static variables
2. instance variables
3. constructors
4. static methods
5. instance methods

While our grammar specified that all variable definitions would precede all methods definitions within a class, there were no restrictions on the relative ordering of different kinds of variables and methods.

We did not specify the visual appearance of their generated HTML pages. Apart from hinting that they should probably use certain HTML structural elements, such as lists, we left the design up to them. We warned them that while we would be happy to see creative web designs, they should be wary not to spend too much time on the visual aspects of the project until they were satisfied that their main "engine" was working correctly.

3.5 Extra Credit

For extra credit, students were asked to parse as many inheritance ancestor (parent, grandparent, *etc.*) classes as existed in the same directory, and use this information to list inherited members. More precisely, each inherited member needed to be documented exactly once if it was not overridden in the child class, and not listed as inherited if it was overridden. The extra credit requirement has an elegant recursive solution since the inherited members of the child are essentially the union of the inherited members of the parent and the members defined in the parent, minus the members overridden in the child.

Students who attempted the extra credit were warned that they would have to address some additional problems. For example, constructors and private members are not inherited. Also, determining equality on method types is an interesting exercise; parameter types must match, but parameter names need not.

3.6 Staged Development

Finally, we were concerned that some students might have trouble building their projects without some guidance on what process to follow. For example, it is difficult to complete the parsing correctly without generating the AST. Therefore, we recommended that students used a staged development approach [5] to building their JDuck implementation. We told them to work in stages by first getting a basic infrastructure working, and then successively adding more and more elements of the grammar to their solution. Fortunately, the grammar partitions easily into class declaration, variable declarations, and method declarations. Furthermore, augmenting a completed project with the extra credit requires almost no change to existing code.

4 A Simple Example

Figure 2 shows a short class definition that conforms to the grammar given in Fig. 1. Figure 3 shows the corresponding HTML for one possible correct solution, and Figure 4 shows the output as viewed in Internet Explorer 4.0.

5 Evaluating the Solutions

We designed and distributed the source code for a simple GUI front-end that all students had to conform to. This proved to be very useful during the grading; only

```
class Duck extends Fowl {
 //Variables
 int age, birthday;
 static private int numDucks;
 public Duck mother;
 //Methods
 protected void newDay(int day) {
   if (day==birthday) age++;
 }
 public Duck(int bday) {
    birthday=bday; age=0;
 }
 public Duck() {
    birthday=age=0;
 }
}
```

Figure 2: Example Input

a few of the submissions needed to be hand-tweaked to get them to compile and run.

Students handed in their solution on a diskette, plus printouts of their source code and the results of a single test run of their own devising. The graders then compiled and ran each submission against a standardized set of test programs and reviewed the results against the expected answers. There were 145 submissions for the 270 students (most students worked as part of a team of two).

The test suites used to grade the submissions were not released to the students beforehand. We warned them that we would be thorough in our test designs, and encouraged them to freely exchange test cases of their own design among themselves. However, few teams actually participated in this.

The submissions were graded mostly on correctness and style, with only five per cent of the grade allocated to visual design of the web pages. The students were warned that they should not spend too much time designing the visual look-and-feel of the web pages unless the rest of their project was already complete.

Five of the best submissions were selected to receive the Golden Duck Award, which we implemented as a T-shirt. The criteria for selection included passing all of the correctness tests, good style, and a compelling visual appearance. The pages generated by these solutions can be found on the JDuck website; the URL is given below.

6 Conclusions: JDuck as a Learning Experience

We now summarize what we think that the students learned, what they told us they learned, and what we would do differently next time.

6.1 What the Students Learned

We feel that JDuck gave the students useful experience in several areas of computer science and software engineering:

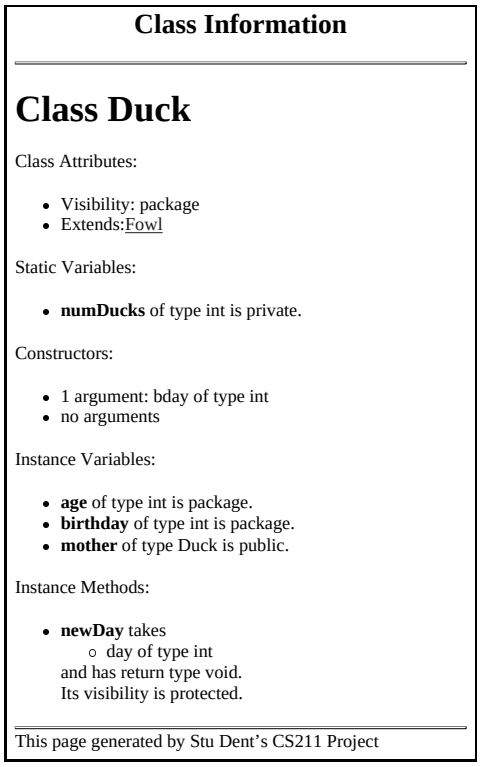
```
<HTML><BODY>
<H2><CENTER>Class Information </CENTER></H2>
<HR>
<h1> Class Duck</h1>
Class Attributes:
<UL>
 <LI> Visibility: package
 <LI>Extends:<a href=Fowl.html>Fowl</a>
</UL>
Static Variables:
<UL>
 <LI><strong>numDucks</strong> of type int is private.
</UL>
Constructors:
<UL>
 <LI>1 argument: bday of type int
 <LI>no arguments
</UL>
 Instance Variables:
<UL>
 <LI><strong>age</strong> of type int is package.
 <LI><strong>birthday</strong> of type int is package.
 <LI><strong>mother</strong> of type Duck is public.
</UL>
Instance Methods:
<UL>
 <LI><strong>newDay</strong> takes
  <UL>
    <LI>day of type int
  </UL>
and has return type void.<br>
Its visibility is protected.<br>
</UL>
<HR> This page generated by Stu Dent's CS211 Project
</BODY></HTML>
```

Figure 3: Example Output

- They designed and used non-trivial object-oriented data structures (*e.g.,* the abstract representation of a Java class and its members).
- They were exposed to a real software engineering tool (`javadoc`), and implemented a simplified version of it.
- They were introduced gently to some advanced CS topics, such as scanning and parsing. We feel this was especially beneficial to those students who may not take further courses in computer science.
- They gained practical experience in the use of a simple design pattern [2] (the iterator, which they had seen in lecture).
- They learned basic HTML.

Furthermore, the students learned how to write part of a larger system:

- Most students worked in pairs, and thus gained experience in working in a small team.
- The project was too large and complex to be implemented in an *ad hoc* manner, forcing the students to plan and think carefully.
- Students gained experience in using a staged development process to create a software system.



Class Information

Class Duck

Class Attributes:

- Visibility: package
- Extends:Fowl

Static Variables:

- **numDucks** of type int is private.

Constructors:

- 1 argument: bday of type int
- no arguments

Instance Variables:

- **age** of type int is package.
- **birthday** of type int is package.
- **mother** of type Duck is public.

Instance Methods:

- **newDay** takes
  - day of type int

  and has return type void.
  Its visibility is protected.

This page generated by Stu Dent's CS211 Project

Figure 4: Browser Output

- Students were not given access to the test suite ahead of time, forcing them to consider carefully the correctness of their projects, as well as how to design test suites on their own.

6.2 Feedback from the Students

The student feedback surprised us somewhat. A common remark was that the project appeared very daunting at first, but turned out to be easier than they had expected. We had been afraid that the project might be too difficult for some of the students, especially those who would not be taking another CS course in their degree. However, only a very few students failed to complete a significant portion of the project; we had pessimistically expected more students to "fall off the edge".

Many students said that they enjoyed being "held by the hand" through the development of an interesting system. One student said that he was interested to see how a real software development tool (*i.e.*, `javadoc`) could be constructed from basic concepts learned in a CS2 course.

Several other students remarked that they particularly enjoyed learning about HTML and web page construction. It was also evident that many students enjoyed the opportunity to express some creativity in their web page designs.

6.3 What We Would Do Differently Next Time

Overall, we feel that JDuck was very successful as a CS2 project. Given the opportunity, we will likely reuse the project; there are many ways of tweaking the project or adding new requirements to make it sufficiently different to be able to use it again. However, given the chance to do it all over again we would make a few changes:

- We would add a few more requirements to the assignment. We feel we slightly underestimated the abilities of the students.
- We would release one "nasty" test suite beforehand. This would give the students an indication of what kinds of cases we would be checking for without giving away too much. We would also encourage the students more strongly to exchange their own test suites with each other.

7 Available Materials

We have prepared a website that contains all of the pertinent information, including example output from the Golden Duck award winners, and the test suites we used: `http://plg.uwaterloo.ca/ migod/jduck/`. No solution is provided on the website, but one may be obtained by sending email to `migod@plg.uwaterloo.ca`.

8 Acknowledgments

We wish express our particular thanks to Max Khavin, who created the customized JDuck lexer using JLex, and also wrote the preliminary version of the assignment handout. We would also like to thank Linda Lee, who designed the Golden Duck Award, Kristen Summers, who created the tutorial on web page design, and the other TAs who were instrumental in the success of the project: Evan Gridley and Martin Handwerker.

References

[1] Elliot Berk, The JLex Home Page, `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Welsley, 1995.

[3] The Java Development Kit Home Page, Sun Microsystems, `http://java.sun.com/products/jdk/`

[4] The Javadoc Home Page, Sun Microsystems, `http://java.sun.com/products/jdk/javadoc/`

[5] Steve McConnell, *Rapid Development*, Microsoft Press, 1996.