Graduation Outliner

A Senior Project

presented to

the Faculty of the Computer Science department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Jason Boyle

June, 2010

© 2010 Jason Boyle

Table of Contents

Abstract **3**

Problem Description and Motivation **3**

Overview 3

Scope 3

Manual Quarterly Schedule Planning Process Description 4

Previous Work **7**

Citations 7

Requirements **8**

Design **8**

Prototype 8

Representation of Requirements 9

Prerequisites and Corequisite 12

Manipulating Quarters 13

Representing Quarters 15

Editing Course Data 15

Representation of Course Names and Lists Thereof 18

Verifying Requirements 19

Adding Data Fields to Courses 21

Saving and Opening Files 21

Course Selection Defect 23

Completed Features **23**

Implementation **24**

Testing **24**

Future Work **25**

Conclusion **25**

Appendix **26**

Abstract

Planning a quarterly course schedule is a difficult and laborious process for university students which must be repeated every quarter. Students are expected to determine which courses to take based on a large and complex set of requirement criteria. Most academic scheduling research has focused on faculty availability and desires. This project attempts to address the needs of students for whom no quarterly course schedule planning tool is widely available. A Java Swing-based application is described that allows students to create a graphical quarter-by-quarter visualization of the courses they plan to take. The application also accepts a list of graduation requirements which may be validated against the planned course schedule in order to report potential errors. The final product is a working application which may help relieve students of some of the difficulties relating to planning and verifying a quarterly course schedule.

Problem Description and Motivation

Overview

Planning a quarterly schedule is a time-consuming and error-prone process for students which must be repeated each quarter. I planned out my quarterly schedule by hand for each of the sixteen quarters that I attended Cal Poly; a detailed description of this process is provided in the next section. The goal of this senior project was to create an application that facilitates this process in order to allow students to efficiently advance through their degree requirements and graduate in a timely manner. The application should also be useful to faculty administrators, who can provide a (incomplete) flowchart/list of requirements to students of a specific major.

Scope

The application can be broken down into three parts: Flowchart, Requirements, and Verification.

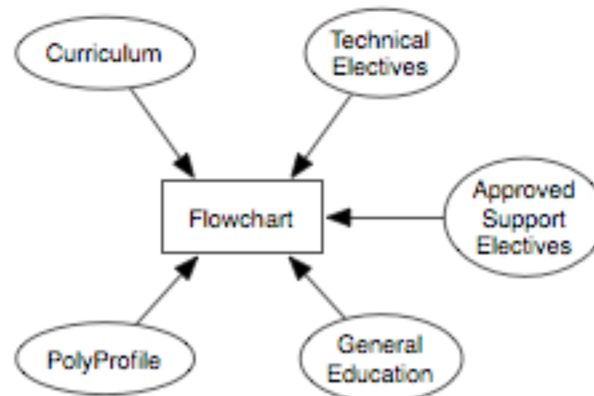
For the flowchart, the application allows a user to create a visual representation of the courses that he or she plans to take (or has taken) each quarter until graduation. Quarters are represented as columns and the courses they contain as labeled boxes. Courses may be rearranged between and within quarters via drag-and-drop. For requirements, the application allows a user to enter each requirement and the courses that he or she plans to take to fulfill that requirement. For verification, which ensures that courses fulfilling prerequisites, corequisites, and requirements exist and are appropriately arranged in the flowchart, the user is presented with a list of possible errors.

The application that I developed for this project does not solve all of the problems that will become apparent through the following description of a manual quarterly schedule planning process, since the scope was adjusted to account for the approximately six months of development time (not including the design and report phases). However, it addresses a fundamental need of Cal Poly students: the ability to create, save, and verify a quarterly schedule.

Manual Quarterly Schedule Planning Process Description

The following is a step-by-step description of the process I followed to plan my quarterly schedule every quarter at Cal Poly. It is included for the benefit of the reader in order that he or she may understand the potential benefits of the application that I developed for this project. In order to ensure that past mistakes would not affect my planning each quarter, I started from scratch each time. These directions assume that the reader is a Computer Science student following the 2005-2007 curriculum, but should be applicable to any major.

Figure 1: Required Resources for the Construction of a Flowchart



Print the Computer Science flowchart provided by the College of Engineering Advising Center (EADVISE) website (http://eadvise.calpoly.edu/dept/). Each box representing a completed requirement will be crossed out by the end of this process.

1. Refer to your PolyProfile, available through the Cal Poly Portal, which lists the courses that a you have completed to date. Cross out each box on the flowchart representing a concrete course that you have completed; for instance, CSC 101.

2. Refer to the Cal Poly General Education Program website (http://ge.calpoly.edu/studentsandadvisors/allgecourses.html) to determine which GE areas correspond to the courses you have completed. For each category (A, B, C, D), follow the link on the GE website. For each GE area (A1, A2, etc.), determine whether you have taken one of the listed courses (by referring to your PolyProfile). If so, cross off the box representing that GE area on the flowchart.

3. Refer to the Cal Poly Computer Science website (http://www.csc.calpoly.edu/csc-tech-

electives/) for technical elective requirements. Due to the complex and non-intuitive nature of these requirements, it may be necessary to spend some time with pen and paper to determine which categories you have completed (for instance, 4 out of 8 units in category 1a). Cross out one of boxes labeled "technical elective" for each 4-unit technical elective you have completed. Unfortunately, you will have to refer to your notes to determine which technical electives to take next quarter; this information is not depicted in the flowchart.

4. Refer to the EADVISE website once again (http://www.csc.calpoly.edu/csc-support-electives/) for approved support electives for Computer Science majors. Cross off any of the courses in this list which you have completed on the flowchart.

5. Refer to the Computer Science Curriculum (http://eadvise.calpoly.edu/policy/forms/csc_curclm0507.pdf) to determine which courses represent non-technical, non-support electives. These appear on the flowchart as: "math/stat elective," "science elective," and "physical science elective." (The last of these actually appears as an alternative between three specific PHYS or three specific CHEM courses.)

6. Cross out one box labeled "free elective" for each 4-unit course you have completed that does not count toward any of the previous requirements.

Requirements may not be "double-counted;" if you would be forced to cross out the same box a second time to fulfill a requirement, that requirement should be considered unfulfilled. This rule does not apply to the USCP requirement.

Once I completed this process, each box on the flowchart representing a course or (part of a) category (i.e., GE or elective courses), I was able to refer to the flowchart to determine which major and GE courses would be appropriate choices to register for next quarter. However, I still had to refer to my notes and the websites mentioned above to determine which technical and support electives to take. If I had not completed this process each quarter, I would have had no way of knowing which courses represented requirements that I had already completed.

Previous Work

All of the publications related to course scheduling that I found were focused on generating schedules based on faculty preferences. The process these systems are designed to replace is best described by Bloomfield and McSharry:

> Faculty members, having been informed of the courses they are scheduled to teach each quarter of the following year, are asked to state their preferences for the days of the week, times of the day, and type of classrooms desired for each quarter. A course scheduler then assembles the resulting preference lists for the 60 or so faculty members of the school, and over a two- to three-week period manually constructs an overall schedule that will satisfy as many of these preferences as possible

As is evident from this narrative, the focus is on satisfying the desires of faculty members. In contrast, this project aims to help students plan out which courses to take each quarter in order to avoid those representing requirements that they have already fulfilled.

Citations

Bloomfield, Stefan and Michael McSharry. "Preferential Course Scheduling." Interfaces 9.4 (1979). <http://www.jstor.org/stable/25059766>

Dyer, James and Mulvey, John. "An Integrated Optimization/Information System for Academic Departmental Planning." Management Science 22.12 (1976).

<http://www.jstor.org/stable/2630146>

Stallaert, Jan. "Automated Timetabling Improves Course Scheduling at UCLA." Interfaces 27.4 (1997). <http://www.jstor.org/stable/25062281>

Requirements

I assumed the role of both developer and client for this project. I wrote the requirements with my desire for a better way to plan my quarterly schedule as a Cal Poly student in mind.

I used Karl E. Wieger's Application Requirements Specification template which is freely available from the Process Impact website (http://www.processimpact.com). I chose this template because I am familiar with it from my application engineering project in CPE 308.

The requirements were verified through weekly discussions with Mr. Dalbey. After writing a Application Requirements Specification document, I wrote a User Manual including UI mock-ups. I used these documents to create a prototype in Java with the Swing API. The prototype was used as a basis for implementing the final project with the same language and toolkit.

Design

Prototype

I created a working prototype in Java with the Swing API during the last few weeks of CSC 491 (Senior Project I). I implemented the main course view using info.clearthought.layout.TableLayout (https://tablelayout.dev.java.net/). Courses are represented as turquoise squares in vertical columns representing quarters. Courses can be selected (their color becomes brighter to indicate selection) and the corresponding data may be edited in an inspector panel to the right of the main view. In addition, courses could be dragged between grid

squares.

I originally planned to continually enhance the working prototype until it became the final product. However, the non-standard TableLayout layout manager would have required an excessive amount of work to manage the model data and update the UI separately. Mr. Dalbey suggested that I use a standard JTable with a table model to manage its data instead. Another problem with the working prototype that I created was my use of the Eclipse IDE which does not have a built-in GUI editor. As a result, I wrote all of the layout code by hand, which was error-prone and time consuming. Mr. Dalbey suggested that I migrate to Sun's NetBeans IDE which has an excellent built-in GUI editor. I accepted Mr. Dalbey's advice and rewrote the application during the first two weeks of CSC 492: replacing the TableLayout with a JTable necessitated a complete rewrite of the application since code to manage the TableLayout was central to the implementation of the working prototype.

The short time that I had to implement the application (approximately two and a half months) made major design trade-offs necessary in order to complete the project on time.

Representation of Requirements

I initially envisioned that users would be provided with a "configuration" file specific to his or her major. This file would contain that major's list of requirements, each of which would include a list of courses which would fulfill it. With this information, users would not need to specify which course(s) they plan to use to fulfill a particular requirement. For instance, the GE Area A1 requirement would list ENGL 133, ENGL 134 as courses may be used to fulfill it. As a result, a user would not have to specify that he or she plans to take ENGL 134 (4 units) to fulfill

the 4-unit GE Area A1 requirement; this would be determined automatically based on the courses in his or her flowchart and the information in the configuration file. (This is accomplished in the Requirements window in the final design.)

Department-specific requirements could be migrated to a separate file, which would contain information about GE and other requirements applicable to all majors in that department.

I planned to use a relational database such as SQLite to store this configuration data. However, I quickly realized that a relational database would be too inflexible to store the various types of requirements. I believed that a requirement would be best represented by the following pseudo-EBNF:

requirement = alternative+
alternative = (units, course+)

For instance, the "Software Engineering" requirement is specified by the Computer Science curriculum document as follows:

CSC 307 Intro to Software Engineering (4 units) <u>or</u>
CSC 308/CSC 309 Software Engineering I and II (8 units)

In Python's tuple/list form (in which tuples of a fixed length are represented by parentheses and lists of variable length are represented by square braces) this would be stored as:

(Software Engineering, [(4, [CSC 307]), (8, [CSC 308, CSC 309])])

I spoke to Dr. Dekhtyar about how I might implement this schema in a relational database. He

advised me that a relational database would be a poor choice for this type of data due to its complexity and relatively small size. He suggested using XML instead, which would allow for more flexibility in specifying requirements.

Dr. Dekhtyar pointed out a fundamental flaw with this representation of requirements: the problem of determining which courses in the flowchart should be used to fulfill each requirement is NP-complete. He suggested that a logic programming language such as Prolog would be ideal for this problem, but would be impractical to learn over the short span of this project. He second suggestion was to use an open source Boolean Satisfiability Solver ("SAT solver") such as the Smodels library.

As a result, I decided that this representation of requirements was outside of the scope of this project. Instead, the user will manually enter the course(s) that he or she plans to use to fulfill each requirement. Consequently, no "guesswork" is necessary in the verification step.

I chose to represent requirements as a number of units and a list of courses which will be used to fulfill it. All of a requirement's listed courses must appear in the flowchart for it to be fulfilled. Therefore, a user would enter *one* of the following into his or her list of requirements. The format is (Name, Category, Units, Fulfillment):

(Software Engineering, Major, 4, [CSC 307]) <u>or</u>
(Software Engineering, Major, 8, [CSC 308, CSC 309])

The name and category fields are for organizational purposes and have no effect on validation. For instance, it would not be clear to which university requirement the example requirement

above would correspond if the name and category fields (containing "Software Engineering" and "Major", respectively) had been left blank; however, it would have no effect on how the requirement is treated in the verification process.

This decision required a major trade-off in favor of ease of implementation over user experience. The original design would have allowed a single file to validate *any* flowchart for a particular major. The revised design requires each user to input which course(s) he or she plans to use to fulfill each requirement.

Prerequisites and Corequisite

The application allows each course to have multiple prerequisites and one corequisite, since I was unable to find any courses in the Cal Poly catalog that required more than one corequisite.

I debated whether unidirectional corequisite requirements should be permitted. In other words, if A lists a corequisite of B, must B also list a corequisite of A? Since one implies the other, only one relationship is necessary. Therefore, I decided that it was reasonable to not enforce this identity in the flowchart.

I originally planned to represent prerequisites (an possibly corequisite) with arrows pointing from one to the other on the flowchart. I decided against this method because it would have cluttered up the flowchart; it would be difficult, if not impossible, to prevent arrows from crossing other courses. As a result, the UI would have been cluttered in an unacceptable way.

Mr. Dalbey and I discussed possible solutions to this problem. A viable alternative was to add

an option which would hide or show the arrows as desired. Another possibility was to add another view which would display courses in a graph, rather than grouped by quarter. I ultimately decided to change the color of a selected course's prerequisites and corequisite on the flowchart. The disadvantage of this method is that prerequisites and corequisite can only be viewed for one course at a time. The advantages include avoiding visual disorder, a simplified user interface, and ease of implementation.

Unfortunately, this method of representing prerequisites is not sufficiently versatile to cover all types of prerequisites in the Cal Poly catalog. For instance, the prerequisite for ENGL 149 is "Completion of GE Areas A1 and A2."

Manipulating Quarters

Deciding on an effective way of managing quarters was a difficult user interface design problem. Quarters must be ordered from earliest to latest (i.e., Fall 2010 comes before Winter 2011). The first problem I encountered was whether to allow nonconsecutive quarters in the flowchart, or "gaps" between quarters. Second, I had to determine whether to allow a user to rearrange quarters. Third, I had to decide on the user interface controls for adding and deleting quarters. Finally, I was unsure which quarters a new (blank) flowchart should contain, if any.

I eventually decided that every quarter that appears in the flowchart must be follow the previous quarter in time; only Winter 2011 may follow Fall 2010; there may be no nonconsecutive quarters or "gaps" between quarters. I initially planned add a preference to toggle display of Summer quarters, but decided against this as it would negate many of the benefits resulting from the decision to disallow nonconsecutive quarters.

I chose to make the controls for manipulating quarters as simple as possible. Only the most fundamental operations are supported via menu items under the "Quarter" menu. These include appending the next chronological quarter after the last quarter in the flowchart via the "New Quarter After Last" menu item. The "New Quarter…" menu item prompts the user for the season and year of a quarter to add to the flowchart. If the specified quarter already exists, the command is ignored; otherwise, the quarter is added to the flowchart before the first or after the last quarter and additional quarters are added in between if necessary. Lastly, the "Delete Empty Edge Quarters" deletes the leftmost quarter until a quarter containing a course is encountered, and repeats this process with the rightmost quarter. As a result, the leftmost and rightmost quarters will contain a course (assuming there is at least one course in the flowchart; if not, all quarters will be deleted).

Another user interface design problem I had to contend with was deciding which quarters, if any, should appear in a new document. My original idea was to prompt the user for the season and year of the first quarter to be added to the flowchart. Although this is not a bad idea, I decided that only the current quarter should appear in new documents. This makes prompting the user for the first quarter unnecessary.

It is easy, if not immediately obvious, to create a flowchart containing only the desired quarters. The process should be trivial for anyone who is familiar with the operation of the three menu items under the "Quarter" menu described above.

Representing Quarters

The Java SortedSet container naturally lends itself to the problem of storing Quarter objects. However, its lookup time is O(N). AbstractTableModel retrieves the quarter anew for each cell every time it is repainted! Therefore, painting the JTable becomes an O(R*S) operation, where R is the number of quarters and S is the number of cells in the JTable. Although in practice this may be a relatively small number, for various reasons the JTable may be repainted every time a key is typed in one of the course data fields. In addition, as a Computer Science major I am inclined to avoid operations with poor running time whenever possible.

My first attempt at resolving this issue was to subclass SortedSet and maintain a member ArrayList containing the same data. I overrode every method of SortedSet which could modify its data and modified the ArrayList appropriately. This was a reasonable (though far from ideal) approach since lookups are much more common than insertions or deletions in this application.

I eventually replaced this temporary class with a class I named SortedArraySet. Unlike the class it replaced, it does not duplicate the data it contains by storing it only in a member ArrayList. The restricted operations which are permitted ensure that the data retains its integrity.

Editing Course Data

Dealing with user-entered data was a major sticking point during the development of the application. When a course is selected, a user may modify its data in the inspector panel on the right side of the splitter in the main window. This initially included only the course name, number of units, prerequisite, and corequisite (both singular). The first few iterations of the application required users to click a “save” button at the bottom of this panel to update the course

data. Mr. Dalbey convinced me that removing this button would be vastly improve the user experience by preventing users from losing data when deselecting a course before saving. In addition, requiring users to manually click a "save" button after every change is non-intuitive and becomes irritating when creating a large number of courses. However, removing the button caused numerous problems which took many weeks to fully resolve.

I initially tried to save the modifications to a field as soon as it lost focus using a FocusListener. However, for reasons that I was unable to determine, fields were not saved in many cases. I believe this may be a bug in Swing. As I was unable to find a solution to the problem after two weeks, I concluded that I would have to use another method of listening for changes.

My next attempt at solving this problem was to prevent the user from deselecting the currently selected quarter as long as invalid data was present in any of the fields in the course inspector panel. At first, I was unable to figure out how to accomplish this, but I eventually figured out that this behavior could be effected by returning false in the stopCellEditing method of an AbstractCellEditor subclass. However, this brief success caused more problems than it solved. I found the inability to change the currently selected course frustrating and non-intuitive.

I accidentally stumbled onto what I believe is an ideal solution. Since I was unable to use a FocusListener to save the current value in each field, I tried to validate the field and conditionally update the data model after each character was typed with a KeyListener. In order to warn the user that a field contains invalid data that will not be saved, the background of that field becomes highlighted in red. A user may still select another course, but should have no

expectation that the data in those fields will be saved. A side effect of this method is that the instant a valid value is entered into a field it is saved (to the data model, not to a file).

When typing this report, I realized that I could have simply inspected the KeyEvent object to determine which key was typed; however, I was previously puzzled as to why the last character that was typed did not appear in the string read from the text field during a keyTyped event. Fortunately, I found the clever (if unnecessary) solution of associating a DocumentListener with each field's editor. A clever use of document properties allowed me to simplify the code somewhat.

Another difficulty with verifying user input was deciding what to accept for course names. Since requirements are verified by comparing course name Strings, it is important that course names have a standard formatting. I initially required that course names be input with a capitalized prefix and a space between the prefix and number. However, in able to make the application easy to use without referring to the manual, Mr. Dalbey convinced me to allow mixed case and to not require a space between the prefix and number. In order to ensure that course names would match even if they were entered differently in the flowchart and requirements list (i.e., "CSC 101" versus "csc101"), I chose to "correct" the input value to conform to a standard string representation; specifically, a capitalized prefix with a space between it and the number.

Comma-separated lists of courses may be entered in the prerequisites field as well as the requirements window. I used a similar technique to validate these lists. In addition to allowing variation in each course name, the space between each comma-separated course name is

optional. As a result, the string “csc101,CSC102, Csc 103” is a valid string which the application will record as “CSC 101, CSC 102, CSC 103”.

Representation of Course Names and Lists Thereof

Courses are represented as a 2-4 character prefix and a number in the data model. Although this representation is valid for the vast majority of Cal Poly courses, it cannot accommodate experimental courses such as “CSC X007”. I chose this representation for simplicity because experimental courses are relatively rare.

Comma-separated lists of courses may be entered in the prerequisite field for each course and the fulfillment field for each requirement. For simplicity, I chose to represent a requirement’s fulfillment (list of courses) as a comma-separated string of course names rather than a list of Course objects. I made this decision because a course is uniquely identified by its name (excepting prefix variations). In addition, it would be a waste of memory to create a list of Course objects which contain numerous fields that would go unused.

Prefix variations are a major pitfall of identifying courses by a name consisting of (a single) prefix and number. For instance, CSC 101 and CPE 101 are the same course. Unfortunately, the two prefixes are used interchangeably; one is no more correct than the other. I originally planned to include a list of equivalent prefixes in the XML “configuration” file (discussed in the “Representation of Requirements” section). This plan was discarded with that idea. As a result, users must use the same prefix for a particular course in the flowchart, prerequisites and corequisite fields, and requirements list.

Verifying Requirements

Writing the algorithms to perform the validations described in the *Completed Features* section was relatively straightforward. I did not encounter any serious problems during the implementation of this feature.

All validations are performed in the validateActionPerformed method of the ValidatorView class.

To determine whether all of the prerequisites for a course in the flowchart are satisfied, its prerequisites String is split into a list of course name Strings. Every course which appears in a previous quarter is compared to each of the course names in this list. When a match is found, the course name in question is removed from the list. The course names which remain in the list after this process is complete represent unsatisfied prerequisites. A validation error of this kind might read, “Prerequisite CSC 101 not satisfied for CSC 102”.

To determine whether the corequisite (if any) for a course in the flowchart is satisfied, the course name String in its corequisite field is compared to every other course in the same quarter. If a match is not found, the corequisite is unsatisfied. A validation error of this kind might read, “Corequisite CPE 169 not satisfied for CPE 169”.

To determine whether duplicate courses are present in the flowchart, a HashMap is used to associate a course name with the number of times it appears in the flowchart. After each course has been examined, a validation error is added for every course with a corresponding value

greater than 1. A validation error of this kind might read, “The course CSC 101 appears 2 times”.

For efficiency, the three aforementioned verifications are performed in the same loop over each course in the flowchart. The following verifications are performed in a separate loop over each requirement.

There are two criteria which must be satisfied for a requirement to be fulfilled. Multiple errors may be reported for a single requirement which fulfills neither of these criteria.

All of the courses listed in its fulfillment field must be present in the flowchart in order for a requirement to satisfy the first criterion. To determine whether this is the case for a particular requirement, its fulfillment String is split into a list of course name Strings. The flowchart is searched for each of these courses. The name of each course which is not found is appended to an “unfulfilled requirement” validation error. For instance: “Unfulfilled requirement Software Engineering: [CSC 308, CSC 309]”.

In order to satisfy the second criterion, the sum of the number of units of the courses listed in a requirement’s fulfillment field must be greater than or equal to the number of units in that requirement’s units field. To determine whether this is the case for a particular requirement, each time a course is successfully found in the flowchart via the process described above, the value of it units field is added to a running total. Afterwards, if this number is less than the number of units in the requirement’s units field, an “unfulfilled requirement” validation error is

reported. For instance: “Unfulfilled requirement Software Engineering: Need 8 units, have 0”.

Finally, in order to determine whether a single course is used to satisfy multiple requirements, another HashMap is used to associate course names with the number of times each is listed in a requirement’s fulfillment field. A validation error of this kind might read, “CSC 101 is used to fulfill 2 requirements”.

Adding Data Fields to Courses

After all of the major features had been implemented, Mr. Dalbey suggested adding a boolean “completed” field to each course that would cause a course to appear crossed out in the flowchart when true. This is an excellent analogue to the way I described crossing out completed courses in the in my description of my quarterly schedule planning process. Therefore, I realized that this was an essential feature of the application. Fortunately, the implementation was relatively painless.

I decided to add optional fields for a course’s title (i.e., “Software Engineering I”) in addition to its name (“CSC 308”), as well as the grade received. As with the completed field, these fields are not used in the requirements verification process.

Saving and Opening Files

I initially hoped to allow a user to have multiple flowcharts open at a time. However, I realized that this would be impractical for a number of reasons. First, Java does not have a built-in document management architecture (such as Document-Based Applications in Cocoa). JDesktopPane and JInternalFrame are not only ugly and only familiar to Windows users, but do not automatically manage documents or even their corresponding windows. Each document’s

separate flowchart, requirements, and verification windows would have complicated this task further. Therefore, Mr. Dalbey and I concluded that it would be reasonable to allow only one document to be open at a time.

I decided to use the Java serialization API because it automatically generates a binary representation of objects in memory. The only problem I encountered was the inability to serialize two data model classes which extend AbstractTableModel since the latter does not implement the Serializable interface. I resolved this issue by simply retrieving and serializing the data for which those classes served as wrappers.

Determining when a document is dirty was a seemingly complex problem with a straightforward solution. I searched for a solution which would automatically send a notification when a change to the data model occurred. Implementing the Change Listener API would have been a valid solution, although it is not automatic as it requires that the stateChanged function be called every time an object is changed. I ended up making the dirty bit a global static boolean variable in the Common singleton class which is set to true in every method of every data model class that might modify that class (i.e., setX, addX, and removeAll methods). I avoided using a global static variable for all other aspects of the application in order to make it as easy as possible to allow multiple documents to be open in future versions. The drawback to the way I implemented the dirty bit is that the application often thinks that the document has been modified even when it has not. However, this behavior can be found in many other applications.

As mentioned in the *Representation of Requirements* section, I originally planned to save files in XML rather than binary. However, implementing an XML parser would have been tedious and unnecessary as the Java serialization API provides binary read and write functionality for free. I determined that the JAXB library would require the development of a schema for the data and a mapping between XML schema components and Java components, which would have added significantly to the development time of the project.

Course Selection Defect

Occasionally the course data fields would not be enabled when a new course was created. I was unable to determine the cause of this defect until the last week of the quarter. It turns out that I was manually setting which courses in the flowchart were selected, but my code did not take into account many of the peculiarities of JTable selection behavior. For instance, clicking in a blank square would select it, and as a result my selection listener code (which enables the course data fields) would not be triggered when a new course was created in that square. This caused other less noticeable problems as well; right-clicking another course would select it, but its selection would not be reflected in the flowchart. I was able to fix this problem by changing the course renderer component, a subclass of DefaultTableCellRenderer, to only use the selected boolean value provided as the third argument to the getTableCellRendererComponent method.

Completed Features

- Basic document management including: new, open, and save operations. (Only one document may be open at a time.)
- Create and delete courses using main menu or pop-up menu.
- Create and delete quarters.
- Drag-and-drop courses between and within quarters.

- Input and update course data, including: name, title, units, prerequisites, corequisite, completed, and grade.
- Input and update a list of requirements consisting of a name, category, number of units, and fulfillment (list of courses).
- Validate the flowchart:
  - All prerequisites of a course appear in previous quarters.
  - The corequisite of a course appears in the same quarter.
  - All courses listed in the fulfillment field of a requirement appear in the flowchart.
  - The sum of the number of units of each of the courses listed in the fulfillment field of a particular requirement is equal to or greater than the number of units associated with that requirement.
  - A course is not used to fulfill multiple requirements.

Implementation

I used an iterative implementation process for this project. I met with Mr. Dalbey at the beginning of each week to demo the application. At each of these meetings, we discussed problems that were uncovered by actually using the software; these problems included defects and missing or poorly implemented features. I attempted to correct these problems in addition to implementing any new features before our next meeting and usually achieved this goal.

Testing

Mr. Dalbey and I performed ad-hoc testing on the application at each of our meetings by testing new and old features. We have analyzed the application running on Mac OS X and Linux. Unfortunately, I did not have time to write formal unit tests using JUnit or another testing

framework. Consequently, the application can only be considered a prototype.

Future Work

As described in the "Representation of Requirements" section, a future project would involve rethinking the way requirements are represented to allow for more flexibility: ideally, a single file would be able to represent any valid flowchart for a particular major. Improving the requirements data model would, in turn, allow the flowchart validation algorithm to be improved: ideally, a user would not have to specify which course(s) in the flowchart should be used to fulfill a particular requirement. This is a complex and interesting problem involving data modeling and algorithms which might make a good future senior project for another student.

The grade attribute of each course in the flowchart could be used to generate statistics and charts or to determine the necessary grades to achieve a certain GPA.

Conclusion

This project is the largest software project on which I have worked to date. It was particularly satisfying to build a finished application from scratch. My advisor, Mr. Dalbey, guided me through the software engineering process of requirements elicitation, design, and implementation. I believe the finished product will be useful to students, although there are many features which I did not have time to plan or implement that must be added before it can be considered production-quality software. The development of this application was an important part of my experience at Cal Poly.

Appendix

The following documents are attached to this report:

1. User Manual
2. Software Requirements Specification
3. User Interface Mockups
4. Sample Graduation Outliner Document