# RSS Reader Design Document

by

Team 14

Paul Groudas, Michael McGraw-Herdeg, and Zachary Ozer

6.170 – Laboratory in Software Engineering

TA Lucy Mendel

Professor Daniel Jackson

Professor Martin Rinard

## Design

Our RSS reader adheres to the MVC architecture. We have a core set of *model* interactions, including both data types and a database handler which takes care of low-level interactions involving those data. Our *controller* class acts as a layer of abstraction between other classes and our model. Finally, we have a *view* class that draws images on the screen and talks to the controller.

### *Model*

#### DatabaseHandler

This class acts as a front-end for our particular database implementation. Because we storeall state information in a database, and constructing holder objects for the data when necessary, it's important that our database handler work quickly and without errors. This class has methods that take relevant arguments from the problem domain (from the controller) and translate them into the appropriate format required by our database. This class also translates the result of database queries to an appropriate model object, or throws an expected exception.

#### Derby

This third-party system is the core of our database; it implements a lightweight SQL-like language. We're using “embedded Derby”, which requires only that our source's build path include the jar (no other configuration is needed). SQL queries ensure fast, reliable data storage, and we get competitive performance with search operations.

#### Folder

This class represents folders in our system. A folder may contain other folders; it may also contain feeds. It may not contain articles.

#### Feed

A feed class represents a collection of articles. The feed itself may have additional information: title, link, description, last updated date. It may also have a uniquely identifying id field (provided by the database) and may know the unique id of its parent.

#### Article

An article represents a single item in an RSS feed. It contains a title, an author, a summary, a date, a link to the article, metadata describing whether the article has been read, is in the trash, or is in the outbox, plus additional metadata reserved for future expansion (rating, and article read time). Articles may have an id field (supplied by the database) and a parent's id.

### *Control*

#### Controller

The controller object acts as the controller in the MVC architecture. The controller carries much of the burden of the system; it handles the GUI events, interprets them correctly, and makes appropriate calls to the DBHandler or the Parser. The controller is capable of two-way communication with the view; it alters the model on the view's request, and it can update the view if any model operations result in exceptions. The controller is the ultimate exception handler in our system.

The controller provides a powerful API to the view that calls it; it will provide on request all information about the model elements, will make modifications to those elements, will search the database, will create and choose among users, and can reset the database on request.

## View Interface

### MailBoxView

This class represents the traditional mail client view for an RSS Reader. For viewing articles, it implements the a standard preview pane that acts as a Web browser; folders and feeds are displayed in a tree, and each feed's articles are displayed in a columnar list.

### NewsPaperView

This class represents the newspaper view. Unread articles are displayed in a *Wall Street Journal*-style lineup, with up to five long articles and up to fifteen shorter article summaries given. Articles can be clicked on,

### FightView

This class represents the “fight” view. This view matches up two words that a user types in and graphically displays the number of times that each word appears among the user's articles. The word with the higher number wins the RSS Fight!

### MinimalView

This class, used for testing purposes and to demonstrate how a view is added, is very minimal indeed: it displays nothing.

### TimelineView

This class represents the timeline view. After typing in a keyword, the user can scroll from the first occurrence of that word among their articles to the most recent occurrence (matching articles are sorted by date of publication). The article itself is loaded and rendered (not a summary).

### RandomView

This class, useful when a user has a large number of feeds, randomly picks an article from among the user's feeds and displays it.

## View API

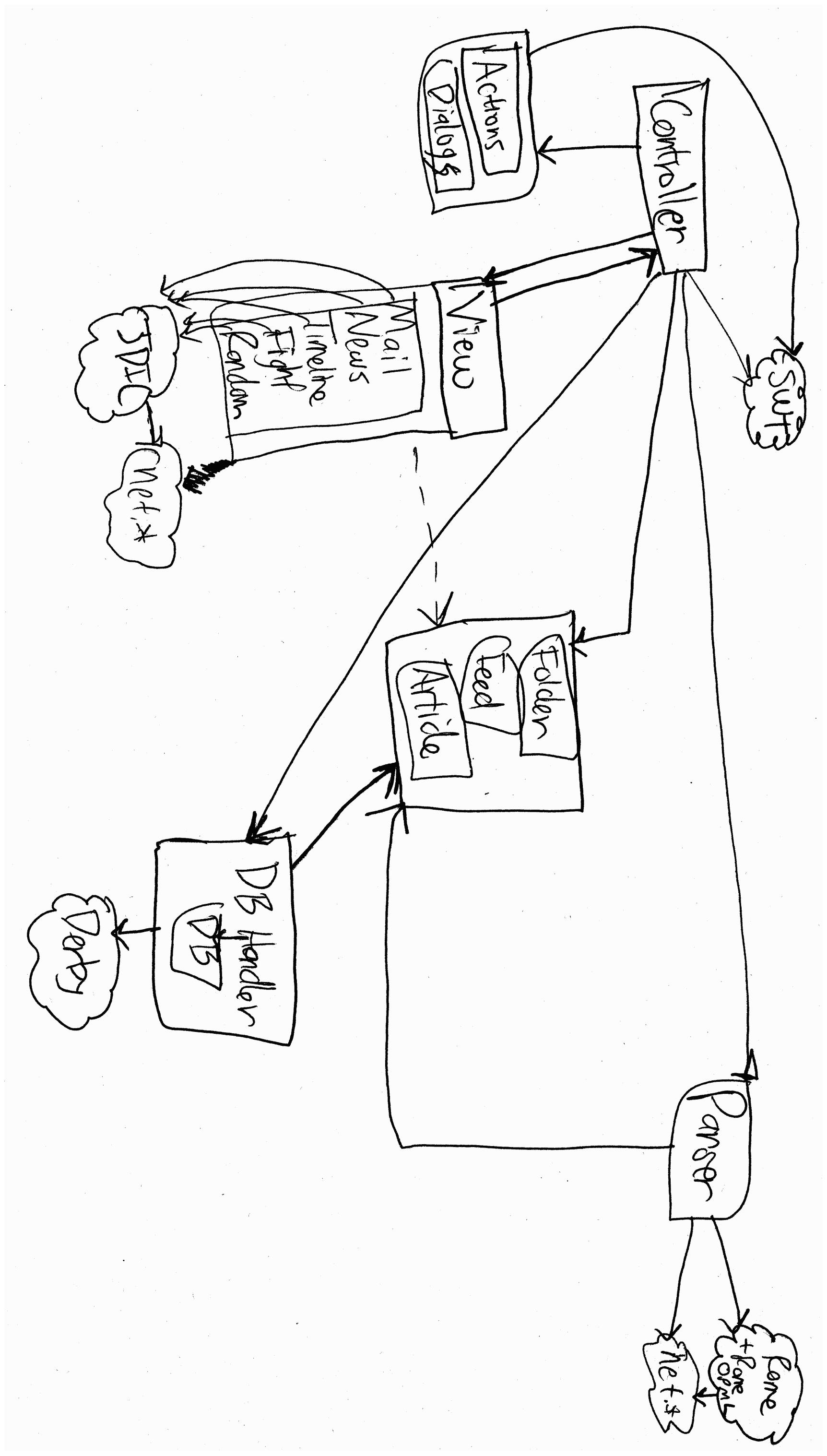
All views must implement a minimum set of methods that affect controller communication; they must have methods to get the current selection (if any), to update, and to set the current browser URL (if the view has one). They may return garbage or null data if they like, but they must implement these methods. All views additionally have access to a controller object, so they can perform any of the myriad operations the controller exposes. Adding a view requires making trivial, well-documented modifications to a single Controller method.

## *Parser*

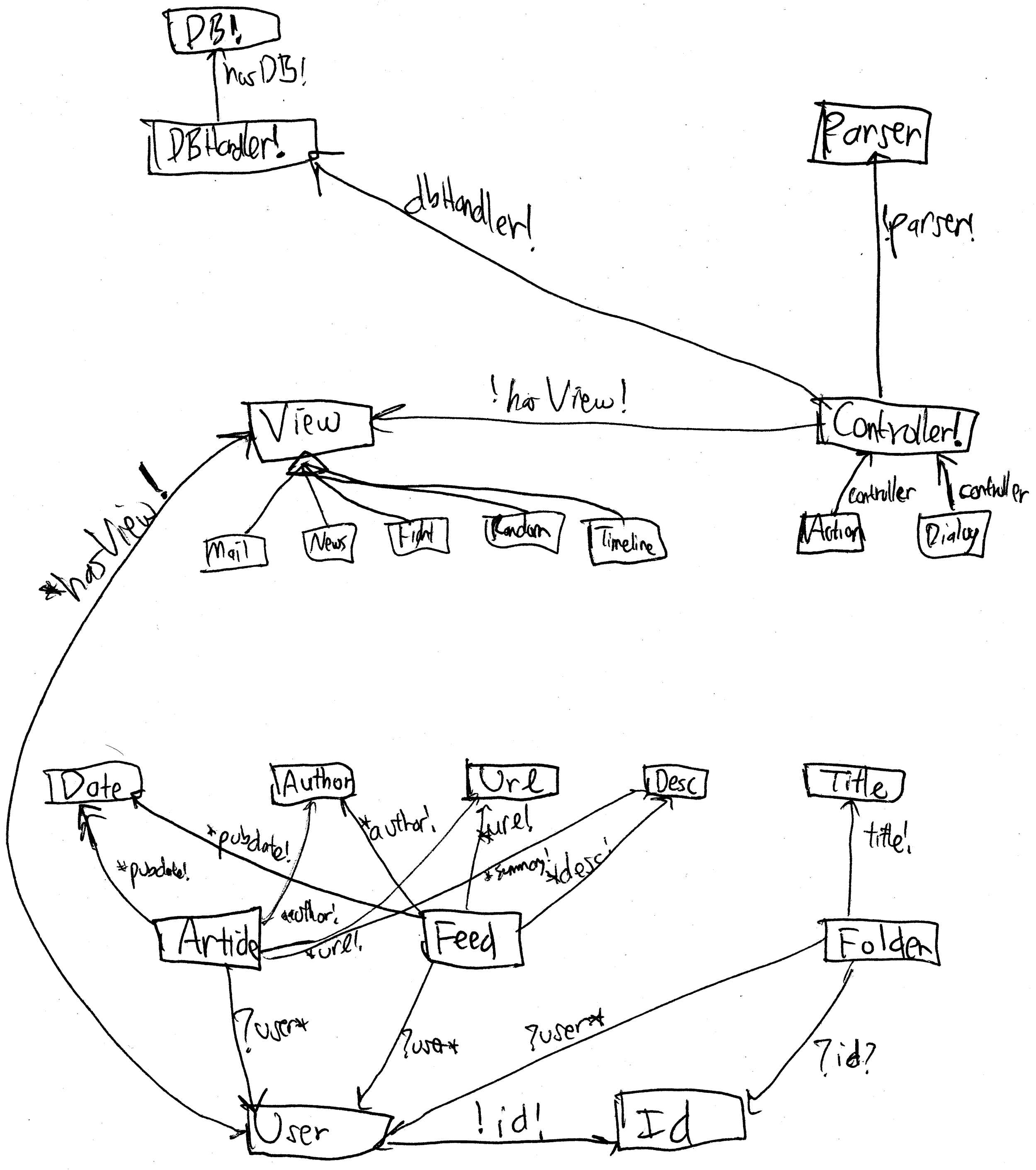
A series of static methods embodied in the Parser class use Rome to take in RSS and OPML feeds and create feed objects with article objects and lists of feed objects, respectively. The parser methods support local and Web-based files. The parser is also capable of writing a list of articles to an RSS 2.0 feed (the user's outbox) and writing a list of feeds to an OPML feed (the user's subscribed feeds list). Parser methods are called by the controller and give it objects from the model domain.

## Diagrams

### *Module Dependency Diagram*



***Object Model***



## Database Design

### *Feeds Table*

A table containing feeds and folders. The table has the following fields:

int feedid (distinct for all feeds)

varchar feedtitle

varchar feedurl

varchar feeddesc (feed description)

date feedpubdate (when feed was published)

date feedlastupdate (when feed was last updated)

date feedlastupdateuser (when feed was last updated by user)

bibo feedicon (reserved for future use)

int feedreadtime (how long feed has been read, reserved for future use)

int feedfolder (another feedid; where the feed / folder is contained)

int feedpreviousfolder (another feedid, usable for undo, reserved for future use)

int feedupdate (how often to update the feed)

smallint isFolder (whether the item is a feed or a folder)

smallint isDeleted

smallint isActive

int userID (id number of the user whose feed this is)

Note that folders are stored in the feeds table as though they were feeds. Since the number of folders is usually small compared to the number of feeds, this adds little overhead to feed accounting; but the distinction is otherwise arbitrary. Future incarnations could create a separate table for folders, though the code would be less simple.

### *Articles Table*

A table containing articles. The table has the following fields:

int articleid (distinct for all articles) varchar articletitle

varchar articleauthor

varchar articleurl

varchar articlecontent

date articlelastupdate

date articlepubdate

int articlereadtime (reserved for future use)

int articlefolder (represents the ID number of the article's parent)

int articleprevfolder (ID number of the article's previous parent, usable for undo)

int articlerating (reserved for future use)

smallint articlevisited (whether the article has been visited, reserved for future use)

smallint isRead (whether the article has been read)

smallint isDeleted (whether the article has been trashed)

smallint isActive (whether the article displays at all, even in the trash)

int userID (id number associated with the user who owns this article)

### *Users Table*

A small but essential table, this tracks user names and internal id numbers.

int userid (unique id)

varchar username (the user's name)

varchar userpassword (reserved for future use)

int userview (integer representing the user's current view)

smallint isActive

## Rationale

## *Data Storage: How and What*

### Databases vs. XML

Our database stores a large amount of information; it must at a minimum support tens of thousands of articles and hundreds of feeds. We considered storing this information in a flat XML file that could be quickly parsed into memory, but rejected this approach because it would be prohibitively expensive in terms of memory and parsing time. Search and access is much faster in our Java Derby database, running in an embedded mode so that it is transparent to the user.

In order to keep all data in synchronization, our model objects don't really exist. All permanent changes to model objects are made through the database (by way of the controller). New, updated data is given to the view when necessary. So all objects created by the system reflect data in the database, and we avoid synchronization issues. As a result, views should always respect the information received from the database; any temporary data stored by the view is secondary.

### Feeds

How much of the data that may be available about a feed is relevant? How much more information should we add? We choose to capture a feed's title, link, and description, plus the date it was last published.

Our system leaves open the possibility of tracking additional information about a feed: future systems may track how often the user has accessed it or how recently the user updated it, but our system doesn't use this information.

### Articles

The various syndication specifications provide an extraordinary wealth of information about an item; not all of it is useful, though the Rome feed parser captures most of it.

We retain an article's author, title, summary, publication date, plus a link to the full item content (the link supplied in the RSS feed). The database imposes size constraints on all these fields, though we've encountered no problems in practice. One possibly problematic size limit is that an article's summary may be no more than 32,762 characters long; in practice, we tried about ten thousand articles and found that the longest was well below this limit.

### Trash

We decided that users might want to get rid of articles they didn't like seeing in a feed. As a result, users can throw away articles they don't like by placing them in the trash. These articles remain visible in the “Trash” feed (in the mailbox view) until the trash is emptied. The drawback to this approach is that it takes two clicks to completely delete an article; we considered this an acceptable sacrifice.

Deleted articles are not removed from the database; this way, updates (see below) don't reintroduce deleted articles. To get back deleted articles, a user must unsubscribe from the feed and resubscribe to it. (We decided an 'untrash' option would be unnecessarily confusing.)

There is no trash for feeds, from which a user “unsubscribes”, or for folders. Unsubscribing from feeds and deleting folders are more serious actions.

### Feed Updates

By default, feeds are updated every 30 minutes. The user can choose the feed automatic update interval. Additionally, the user can trigger a manual update of a feed or of all feeds by pressing the appropriate GUI button. Updating a feed adds only items which have never been deleted. By letting the user configure update times, we allow a significant but not overwhelming amount of choice.

### "State" Information: A User's Active View

Users may have preferences and settings that aren't directly related to the outbox. Currently, we keep track of only one user setting: the active view. All other information, from articles to feeds to update intervals, is stored somewhere besides in the database's users table. (Future versions might include password protection on user accounts; the current database and system are set up to support this.)

We chose not to retain user settings in an outside configuration file because we wanted to minimize the number of files needed to run the application. Currently, only the application itself and the database folder automatically generated by Derby are essential to the program's functionality.

## *Parsing*

### Parsing Different Kinds of Inputs: The Parser Interface

We want to be able to handle many kinds of input files. Luckily, Rome is fairly versatile; it handles all valid feeds of the required Atom 0.3, RSS 1.0, and RSS 2.0 format, and many invalid feeds and feeds of other formats like Atom 1.0 and OPML. This library lets us parse many different kinds of input XML and produce standard output of the form of model objects. (Using the parser in reverse, we can generate valid RSS as well.)

### Feed Validation: How and When

We considered doing feed validation in addition to that done by Rome, which parses fairly forgivingly but which can be configured to report invalid feeds. The specification required only that we parse valid input files; it was silent on the topic of invalid input, but we decided that to be most useful to the user, we would attempt to parse as much as possible and fail quietly on any invalid feeds. (For instance, feedvalidator.org rejects the New York Times front page RSS feed, but it's useful for us to be able to handle this.)

We considered but rejected notifying the user explicitly of an invalid, properly-parsed feed, but decided that this would present no benefit to most users.

On a related note, we face the corner case of “empty” articles, those with no usable content. While they represent valid articles, we consider them to have no value and to be of no interest, and thus, they are simply deleted.

## *Users*

Our system supports switching among multiple users simply by having a table of user ids and names and by attaching a user id to each article, feed, and folder in the database. Users have some metadata (current view, plus a supported-but-not-implemented password). Though this approach introduces a little inefficiency – suppose two users are subscribed to the same feed – it is ultimately necessary, since it's rare for two users to be subscribed to exactly the same feed (with no elements deleted and with updates performed at the same times). This approach yields fast user performance.

When the application is first started, the user is prompted for a name; this will become the first user name in the system. We decided to make it easy to switch among users; since views are preserved between users, there's no reason to make it difficult, and since the application runs locally, there's little reason for extensive security.

## *Exporting*

What exactly is an outbox? It's a special feed, to which articles can be copied, and in which articles can be created. In this sense it's a subclass of "feed" with the special property that you can't unsubscribe from "outbox" and the property that you can export your outbox as a feed in RSS 2.0 format. (Nothing prevents us from letting the user choose the format of the feed, but this would help no one.)

## *Folders*

### Folder Hierarchy

In our model, a feed is simply a collection of articles, and folders are simply a collection of feeds and folders. Thus, it makes sense to think of a folder as simply a feed of feeds, or simply a larger collection of articles. As a result, there must be a root – in our case, a fictional entity which is a list of one folder (“subscribed feeds”) and two feeds (“outbox” and “trash”).

### Moving and Removing Articles

Articles can be moved from their default feed to an outbox or to the trash. Moving an article to the outbox is done by creating a new copy of the article object and placing it in the outbox feed. This way, it's safe to mark the article as deleted in the original feed while it remains in the outbox feed.

The trash is represented as a feed that articles are moved to, but where they are not copied beforehand. Rather, the only copy of the article appears to be moved to the trash, though it can always be restored to its original feed. (In fact, the trash is not a “real” feed – but that's something that the reprsentation does not expose.)

## *Views*

We want to be able to load additional views “on the fly”. In order to preserve modularity – and to properly keep track of a user's current view – we chose to have a basic view interface that each of our views implements. Developers can create a view that implements this interface and takes advantage of our Controller API, and they can trivially insert their view into our list of available views. This allows more views to be added quickly without significant alterations to the base code.

## *Testing*

In choosing appropriate tests, we prioritized first ensuring the internal consistency of our classes, second making sure that interactions between various parts of the system work as expected, and third that external interactions (with the GUI and the operating system) behave correctly.

To this end, we wrote a series of JUnit tests to ensure that the feed, article, and folder data types were consistent with their specifications; testing revealed a few problems with null data which we fixed. We then exhaustively tested the database handler and parser's interactions with the controller, also using JUnit; we fixed the failures we exposed.

We mostly tested GUI behavior through actual use; hands-on testing helped reveal when methods took too long (because of threading issues), what exceptions were incorrectly thrown and passed (based on how the controller catches them), and whether graphical elements looked acceptable. Frequent testing ensured that there were no surprises in GUI behavior on the Linux and Windows platforms we used for development; we didn't heavily develop on OS X, and so we were pleasantly surprised that our code converted easily and ran just as well.

Because we didn't do primary development under OS X, we didn't notice until far too late that SWT exhibits some erratic behavior on the OS X platform. In particular, heavy abuse of the code (loading many feeds at once and rapidly switching among views) yielded occasional display artifacts and once an inexplicable crash. Earlier testing might have helped us resolve these errors, or it might have just provided more proof for the common wisdom that SWT on OS X is still immature.

## Post Mortem

Our database-centric approach paid off; we were able to trivially extend our system to support the project amendment, and in general our system is designed so that data-based features can be added quickly with a minimum of additional code. One possibly dubious decision was putting together folders and feeds in the same table; in our experience so far, the benefits of drastically simpler SQL queries still outweigh the drawbacks of representing two kinds of data in one table.

The controller became slightly more powerful than we had originally intended; because it is the main method of the system, it is where the basic graphical elements like toolbars and the status bar are instantiated. This design decision resulted in a slightly larger controller than we might have otherwise had, but saved the cost of having an additional view class and made it easier for views to communicate messages to status bars. We had to be careful to split up the controller's methods as much as possible in order.

Our view interface is less pluggable than we had hoped. Originally, we had hoped it would be possible to load an external file containing a plugin and to directly implement that view; instead, developers must add a few lines in one, well-documented place in order to add a new view. In order to make view plugin development easy, we were careful to document the controller's public methods heavily and to make sure we only exposed methods we wanted views to be able to call. (View development is pretty easy; we went on to develop several more.)

Parsing turned out to be much easier than expected; instead of having to implement any kind of interface, and have separate classes for parsers that took different kinds of data, we needed to implement only one important parser method. Rome parses efficiently but not very elegantly, throwing occasional unexplained exceptions and using a poor HTTP useragent by default; and its OPML handling is quite young. We had to make up for some of Rome's shortcomings incode; this effort cost time but was successful. We gave only cursory looks at other RSS parsing libraries, and we might've saved some time and effort had we found a better-documented, more fully-featured library. Still, by not doing any XML parsing ourselves, we saved quite a headache.

We worked closely on many aspects of this project, so integrating the different parts of the code base was never a problem. Indeed, our biggest frustrations came at the end of the project – when, in the course of packaging, we learned that Eclipse cheats by providing boatloads of native libraries to SWT applications. Figuring out how to supply these compiled libraries under each platform was a challenge, and we would've been better suited to face it had we tried ahead of time. (Spoilers: under Windows, a properly compiled jar needs only one DLL in the same directory; under OS X, a handful of .jnilib files are needed in the same directory; under Linux using GTK, we had to manually set both MOZILLA_FIVE_HOME and LD_LIBRARY_PATH, with the latter containing Mozilla & SWT .so files.) The latter problem was directly related to our choice of JDIC, which proved to be an excellent browser tool but which, in retrospect and in light of ads and popups seen in Internet Explorer, might not have been perfect.

We were lucky to be well ahead of our own schedule throughout the project; we were unlucky to have an extra day to turn files in – twice – as this led us to do a little bit of work we would've otherwise left to future updates. Adhering to schedules occasionally meant the sacrifice of a small amount of other classes' work and some quantity of social life but not hygiene.