MIT 6.170 Fall 2005 Final Project

Christopher Moh, Weijie Yuan, Wonsik Kim, Yongjin Kim

Fable (RSS Reader) v1.0 Design Document

Last updated: Monday, December 12, 2005

1 Requirements ... 3
1.1 Overview ... 3
1.2 Possible Usage Scenarios ... 3
1.3 Features ... 4
2 Design ... 5
2.1 Data Layer ... 5
2.2 Logic Layer ... 6
2.2 Presentation Layer ... 6
3 Logic and Implementation ... 7
3.1 Logic overview ... 7
3.2 Event Handling ... 8
3.3 Thread Management ... 8
3.4 Input and Output and Caching ... 10
3.5 Filtering and Censorship ... 10
3.6 Searching ... 12
3.6.1 Searching Within Articles ... 12
3.6.2 Searching For Articles ... 13
3.7 Exception Handling ... 13
3.8 The FableManager class ... 13
3.8.1 Starting Fable ... 13
3.8.2 Stopping Fable ... 14
3.8.3 Threading ... 14
3.8.4 Interactions with the GUI ... 16
3.8.5 Interactions with the ADT ... 16
3.8.6 Interactions with Input / Output modules and caching ... 17
3.8.1 Loading Feeds ... 17
3.8.2 Loading and Storing Articles / Caching ... 17
3.8.3 Creating Custom Feeds ... 18
3.8.4 Importing/Exporting a List of Feeds as OPML ... 18
3.8.5 Importing/Exporting settings as FBML ... 18
3.8.6 Exporting a series of Articles as PDF ... 19
3.8.7 Telling friends about Articles ... 19
3.8.8 Interactions with Logic modules and Censorship ... 19
3.9 Package Overview ... 20
3.10 Implementation of various layers ... 21
3.10.1 Data Layer ... 21
3.10.2 Logic Layer ... 24

3.10.3 Presentation Layer 25
4 Design Decisions........ 26
4.1 ADT and Data storage issues 26
4.1.1 Storing of HTML content of Articles in memory 26
4.1.2 Lazy Evaluation and Database........ 26
4.1.3 When to show Articles from the Cache 27
4.1.4 Handling of PseudoFeeds as a Class 27
4.2 Logic Issues........ 28
4.2.1 Exception Handling 28
4.2.2 Incremental Search 28
4.2.3 Threading 29
4.2.4 Implementation of Censorship........ 29
4.2.5 Event Handling and MessageHandler Observer Singleton 29
4.2.6 Separating Manager and Observer and the use of Façades........ 30
4.3 GUI and Usability Issues 30
4.3.1 SWT vs. Swing........ 30
4.3.2 Naming Conventions 30
4.3.3 Tabbing – the use of Multiple Tabs 31
4.3.4 Usability of the dialog used to Add New Feeds 31
4.3.5 Showing articles in a folder 31
4.3.6 Status Bar 31
5 Testing 32
5.1 Testing of Model (ADT)........ 32
5.2 Testing of Input / Output 32
5.3 Testing of Logic modules........ 32
5.4 Testing of GUI and Functional Features 32
6 Reflections 36
6.1 Software Design Process 36
6.2 Team Management........ 37
6.3 Conclusion........ 37
7 Appendix 37
7.1 External Libraries Used 37
7.2 Milestones and Date Achieved 38
7.3 Work Allocation........ 38
7.4 Module Dependency Diagram (MDD)........ 40
7.5 Problem Object Model (POM)........ 41
7.6 Code Object Model (COM) 42
7.7 Code Object Model (COM) – continued........ 43
7.8 Code Object Model (COM) – continued........ 44

1 Requirements

1.1 Overview

Fable is an RSS reader that allows the user to add RSS feeds in RSS 1.0, RSS 2.0, or Atom 0.3 format and browse these articles internally, using its own web browser. It allows for content censorship. Users are also able to highlight their favorite articles or feeds and export these feeds or articles in OPML format (for feeds) and RSS and PDF format (for articles).

Users are able to export own articles to their personal binders and edit these articles in their binders, as well as inform their friends of their favorite articles through email.

This document covers the design decisions made in the creation of Fable.

1.2 Possible Usage Scenarios

1. Subscribe to RSS Feeds (required) – User is able to subscribe to RSS feed:
    a. By providing URL
    b. For users who do not know the URL of a feed, they are able to obtain the feed by searching for the feed by keyword (utilizing feedster.com)
    c. Load local RSS/RDF/XML file
2. Read Articles from Feeds (required) – After selecting a feed, user is able to browse the articles contained in that RSS feed as well as read them in an embedded web browser.
3. Import/Export feeds to OPML file (required) – User is able to import and export many feeds into an OPML file. This facilitates the easy sharing of feeds between people.
4. Import/Export RSS reader settings (required) – User is able to save all information related to feed retrieval interval, organization structure, and other preferences into a file. Preferences can also be loaded. Thus, it would be easy for a user to quickly adapt his preferred feeds and settings on different machines.
5. Arrange and organize RSS Feeds (required) – User could organize feeds into folders/categories for easier organization. This is done via drag ‘n drop.
6. Cached Offline Viewing (required) – Feed and Article content is cached and available for viewing, even when user is offline.
7. Incremental (Keyword) Search of RSS Articles (requires) – User could search for words contained in articles, results are displayed as the user types. This feature supersedes the required Keyword Search feature.
8. Organizing articles into Binders (similar to a Favorites folder) – User can add articles from other feeds into his own binders. The user can also organize these binders and edit article content in these binders.
9. Search for Feeds – User is able to search for feeds by keyword.
10. Embedded web browser – When a link is clicked from an article, a new tab would open, displaying the selected page in it using the embedded web browser.

11. Changing Feed settings – User is able to change settings for caching of feeds and refresh interval, for both individual feeds and whole folders
12. Sorting and marking – User can mark articles as being preferred (starred) and he can sort articles by their date or author or their title.
13. Filter Articles/Feeds – User could filter displayed articles (by only showing those written after a particular date, or those that are starred, etc). The user is also able to set his own custom filter.
14. Content censorship (required – amendment) – User is able to censor certain URLs and articles that may be undesirable. Of course, he is able to change these censorship settings if needed to. Censorship is controlled by a master password.
15. View of Articles as PDF – User is able to view summaries of articles as PDF file. This makes it easy to peruse articles and also to print them
16. Email Articles – User is able to email summaries of articles to anyone he desires. Optionally, the PDF of these articles can also be sent as an attachment.

1.3 Features

Based on the usage scenarios we envisioned, we implemented the following features:

Num.	Feature Descriptions	Corresponding Scenario
1	Adding a new feed	1
2	Adding a new binder	8
3	Adding a new category	1, 8
4	Paneled viewing of article list and article	2
5	Articles can be marked in several ways: read/unread, starred/unstarred	12
6	Drag and drop reordering of feeds, binders, and categories	5
7	Menu, toolbar, and context menus to simplify user interaction	1-16
8	Articles are stored/cached locally according to user specified settings	6, 11
9	Online feed search using Feedster.com	1, 9
10	Incremental search of local articles	7
11	Filtered view of articles that allow only articles fitting certain criterion to be viewed, this allows censoring keywords from articles	13, 14
12	View articles as PDF	15
13	Send articles via email	16
14	Embedded webs browser	10
15	Export/Import OPML	3
16	Export/Import Fable settings as FBML file	4

Our philosophy in designing the Fable is to have multiple ways of allowing the user to accomplish tasks. Common usage tasks can be performed using toolbar buttons, menus and also context menus. We also pay attention to affordances, and allow common tasks to be performed in methods familiar to the user.

2 Design

Fable's design can be broken up into the following layers:

2.1 Data Layer

This layer handles the concrete data and consists of two main sub-layers:

Abstract Data Type (ADT). This layer stores the data present in feeds, folders (categories) and articles. This layer provides an interface to allow other classes to create new feeds/articles/categories and to access and modify content.

An abstract summary of the types used:

An **Article** contains information about a particular news event. It has a title, summary, author, date created, a html link, and other properties.

A **Feed** contains many articles. This is the usual data type that RSS Readers read and that providers (news sites, blogs, etc) supply in the form of RSS / Atom files.

A **Folder** (or category) contains many feeds and helps to groups the feeds into various user-defined categories. Folders may contain other folders.

In addition, Fable employs a fourth kind of data type, which is distinct from the other three:

A **PseudoFeed** is similar to a feed in that it contains articles. However, it may not belong to any folder, nor do the articles in it truly "belong" to it – it is simply a container of articles from many different feeds (these articles belong to their parent feeds, not the **PseudoFeed**).

To summarize, we may view their relationship abstractly as follows:

- **Folders** can contain other **Folders** or **Feeds**. These children take their enclosing **Folder** as their only parent.
- **Feeds** can contain **Articles**, and these **Articles** take the enclosing **Feed** as their only parent.
- **PseudoFeeds** can contain **Articles**. These **Articles** do NOT take the enclosing **PseudoFeed** as a parent.

This abstract definition does not preclude there being multiple top-level folders or feeds or articles that have no parents or enclosing feeds/folders.

In Fable, we employ a stricter version of the hierarchy:

- All **Articles** must have exactly one **Feed** as a parent
- All **Feeds** must have exactly one **Folder** as a parent
- All **Folders**, except for one, must have exactly one **Folder** as a parent. The last **Folder**, denoted as the *root* **Folder**, has exactly zero parents.
- **PseudoFeeds** do not belong in the hierarchy

This definition means that the *root* **Folder** is necessarily the ancestor of all other ADT objects. This *root* **Folder** is not defined by the user; it is specially created by Fable and is named "Fable."

Input/Output layer. This layer is a collection of different classes designed to provide an interface to allow storage and retrieval of data from the local file system and the World Wide Web. This layer interacts with the ADT layer to create new data types (when reading from file) as well as accessing fields (when writing to file).

2.2 Logic Layer

This layer serves two features: As an interface between the presentation and data layers, and as a series of utilities that perform filters and searches on the concrete data.

The logic layer consists of several classes that do searching and filtering; however, they do not provide a viable interface. Instead, the interface between the presentation and data layers consists entirely of a single **FableManager** (see below: **The FableManager Class**) class that interacts with all other classes. This use of a *façade* design pattern allows the GUI to perform logic operations such as adding and deleting feeds, refreshing articles, censorship, filtering, renaming of components etc. as well as ADT-changing operations such as adding/removing feeds etc. all through this class, thus removing all dependencies on other classes.

2.2 Presentation Layer

The presentation layer consists of a Graphical User Interface (GUI) which uses the native look-and-feel of the operating system to present a tree of feeds and categories, the contents of feeds and articles, as well as a browser view to show html contents of articles.

The presentation layer interacts with the logic and data layers through the **FableManager** class, which as mentioned earlier provides a façade. All logical (Filter, Sort, Search, Censorship), I/O (Read, Write) and mutator (add/remove feeds, change the structure of the feed tree etc) operations go through this interface, which minimizes all dependencies on the presentation layer on the data layer.

Fable uses a modified *Model-View-Controller* architecture/design pattern, where the View and Controller are integrated within the presentation layer and the Model is represented in the data layer. The logic layer serves as an intermediary between the presentation and data layers to process data to a form suitable for presentation. This interface is represented by the **FableManager** class. Within the presentation layer, the *Controller* processes send requests to **FableManager** either for information (either for raw data or data processed by the logic layer) or to request modifications to the data. The *View* processes interact indirectly with the ADT through the use of a global observer that provides a façade between events fired by the ADT and listeners for these events in the *View* processes (see **Event Handling** in **Logic Overview** below). To prevent confusion, for the rest of this document (except when explicitly discussing the presentation layer) we will refer to the presentation layer simply as the Graphical User Interface (GUI) or simply "presentation layer" instead of referring to the "*View* processes" and "*Controller* processes" of the presentation layer.

The general design of Fable allows for a top-down layered dependency structure with the logic layer providing an intermediary layer between the presentation and data layers. Although the presentation layer retains a dependency on the data layer, this is only on the ADT and the dependencies are extremely weak.

As far as possible, cyclic dependencies are reduced, or made extremely weak. The overall design allows for easy dividing of the workload between our team members so that implementation can be done bottom up.

Because of the façade design pattern used, it is extremely easy to modify existing features in Fable without having to change a whole bunch of classes – in most cases the only class needed to be modified would be **FableManager**. The design also allows for easy extension of existing capabilities both in terms of the presentation as well as in the logic layers.

The overall dependency structure of our design can be seen in the Module Dependency Diagram (see **MDD** under **Appendix** below). It can be seen that the overall design is top-down with an intermediate façade layer (comprising of the above **FableManager** class as well as the **MessageHandler** class described below (see **Event Handling**, below in **Logic Overview**)) which simplifies dependencies greatly and reduces coupling.

3 Logic and Implementation

3.1 Logic overview

This section discusses the design decisions and overall logic that goes on behind the management of Fable and the interactions of the various modules. This is **not** purely a description of the logic layer, although several modules included in it are mentioned.

3.2 Event Handling

The ADT fires off events when certain properties are modified. Event listeners can latch onto these events and perform appropriate actions when needed. We simplify our design so that only one listener is needed for the entire ADT, and in fact, the entire software.

There exists a special **MessageHandler** class that demonstrates the *Singleton* and *Observer* design patterns. This class is a façade between all events fired (regardless of whether they are in the GUI or ADT) and all listeners (mainly in the GUI). It is the ONLY class that directly registers itself as a listener with all modules that fire events. All other classes register itself with the singleton instance of this class, which acts as an *observer*.

When an event is fired, the single **MessageHandler** (it is a singleton) instance registers the event and proceeds to broadcast it to all modules registered to listen to that particular event.

This allows a great deal of modularity and improves code management a great deal. Most GUI objects e.g. the **TreeView**, needs to register itself with every single member of the ADT, so that **StructuralChangeEvents**, fired when new feeds / categories (folders) are added, will be automatically updated in this view. This also means that every new ADT object created will also need to register the **TreeView** class as an event listener. This might be manageable were it only the **TreeView** class that needed to listen to such events; however, with many such GUI classes all listening to events fired from the ADT, the code quickly grows to a large unmanageable size. This problem is compounded when some GUI objects need to listen to more than one particular kind of event e.g. **ArticleMarkChangeEvent(s)** as well.

The use of such a **MessageHandler** observer solves this problem. All ADT instances simply need to register itself with one class on initialization. Similarly, all GUI classes simply only need to register itself as an event listener with the **MessageHandler** class. The **MessageHandler** class will take care of broadcasting appropriate events to the appropriate classes. This also allows a great deal of flexibility in adding new events or modifying event handling.

MessageHandler also deals with internal GUI-fired events and their listeners.

To simplify event handling a great deal, all event objects have to inherit from the **FableEvent** class.

3.3 Thread Management

As Fable may need to do several things concurrently (Adding feeds, refreshing feeds) while interacting with the user and keeping the GUI smooth, threading is required.

The class dealing with threads is the **FableThread** class. This class represents a single thread. It is a daemon and will terminate upon closure of the program.

The **FableThread** class, in addition to the methods defined in **java.lang.Thread** (**FableThread** inherits from it), defines the following methods:

```
public void addTask (Runnable newTask);
```

Adds a task to the thread. This task will be scheduled to run after all previous tasks are completed.

```
public void stopThread();
```

Tells the thread to terminate **after** its current task is complete. All tasks after the current task will be discarded, however, the current task will complete.

Since tasks are added one by one sequentially to this thread, **FableThread** uses a **java.util.ArrayList** object to store the list of tasks. Each task should implement the **java.lang.Runnable** interface.

To avoid complicated thread management and deadlock handling, the Standard Widget Toolkit (SWT) which we used for GUI allows only one special thread – the SWT display thread – to perform tasks that may eventually modify the GUI. The problem with this approach is that if computationally-intensive tasks are passed to this thread, there will be noticeable lag to the user as these tasks are being processed. This is extremely bad usability-wise.

To overcome this problem, we define a special interface that simply wraps around **java.lang.Runnable**:

```
public interface FableNonGUITask extends Runnable
```

FableThread will delegate all tasks to the display thread, except for tasks that implement the **FableNonGUITask** interface, which it will run within the **FableThread**. The following criteria are required for a task to be classified as a **FableNonGUITask:**

(1) It has to be processor intensive
(2) It cannot result at any stage in an update of the GUI (for example, it cannot modify any ADT that is displayed in the GUI because GUI listeners will pick up the event broadcast through the **MessageHandler** event handler) – otherwise SWT will throw a Thread-related Exception

The reason why **FableThread** needs to deal with both normal **Runnables** as well as **FableNonGUITasks** is because these tasks need to be run in a particular order. We will cover details of this in **Threading** (under **The FableManager class**, below).

Note that **FableThread** does not guarantee that two threads will not access the same object at the same time, resulting in unforeseen circumstances. There are no object locks used within Fable – the advantages of this are that there will never be a situation of deadlock and there is no need to ensure that methods are synchronous. The overall Thread management, which prevents simultaneous access at the same time, is discussed in **Threading** (under **TheFableManager class**, below).

3.4 Input and Output and Caching

Fable comes with a package of I/O modules that can do the following:

(1) Read and write articles to and from RSS files (Writing is in RSS 1.0 format)
(2) Read and write a list of feeds to and from OPML files
(3) Export a feed as a summary in .pdf format
(4) Read and write an FBML (FaBle Markup Language) file

FBML is an extension to XML that allows the entire hierarchy of folders and feeds, together with censorship options and settings, to be exported as a single file.

If the user wishes to export his settings and feed hierarchy to a file that can be read by another copy of Fable on another machine, FBML is the way to go.

FBML is so useful that Fable even uses it to store data. There is no database interface or structure needed in Fable; FBML is sufficient.

The main FBML file that Fable uses to store data is read once, when Fable loads, and written once, when Fable closes. Note that the FBML file is relatively small because it does not contain html contents of Articles, only a link to a file that contains the actual html content.

This file that contains the html content will reside on the file system. Fable caches all html content for a certain period of time (to be determined by the user), or until the article is manually deleted. This cache is in a separate directory on the file system and is linked to by the FBML file, as stated earlier.

Articles, folders and feeds are organized in such a way that there exists only a single root folder (named “Fable”), as discussed previously (under **Organizational Structure**, above).

3.5 Filtering and Censorship

One important aspect of any RSS reader or email client is for the user to be able to view only articles that follow a certain criteria e.g. unread only, written by a particular author, updated in the last 7 days, etc. This adds an important aspect to usability.

Fable incorporates this by providing a **Filter** class that contains a single static method:

```
public static PseudoFeed filterArticles (PseudoFeed ct, FilterStrategy f);
```

This method filters articles in the **PseudoFeed** ct according to the **FilterStrategy** f provided and returns a **PseudoFeed** containing only articles that the provided **FilterStrategy** allows.

In incorporating filters, we utilize the *strategy* design pattern. **FilterStrategy** is an interface that filter classes will implement to provide the required filtering.

Although **FilterStrategy** instances may be created manually, the preferred way is to use a **FilterStrategyFactory**. This class, which is one of the ways we have used the *Factory* design pattern, contains static methods that generate **FilterStrategy** instances for various filters, including date, mark, and author filters. In addition, **FilterStrategyFactory** defines the following static method:

```
public static FilterStrategy acceptCombined (Collection<FilterStrategy> x);
```

This allows a **FilterStrategyFactory** to combine a collection of **FilterStrategy** instances into a single **FilterStrategy** instance.

The use of the *Factory* design pattern allows us to encapsulate and hide details of the underlying **FilterStrategy** classes, such as **DateFilter, MarkFilter, AuthorFilter, CombinedFilter** etc and provides a direct interface for the client, which saves us a lot in terms of code complexity, reusability and extension.

In addition, by interning common filters (such as filtering for unread documents), the **FilterStrategyFactory** saves a lot of processor time in terms of garbage collection as a single instance of common filters can be used. This can be done because the concrete classes that **FilterStrategyFactory** returns are *immutable*.

The **Filter** class is part of the logic layer. As mentioned previously (see **General Overview**, above), the interaction of the logic layers with other layers is controlled by the **FableManager** class. The **Filter** class is no different. It is never directly interfaced by the presentation layer; instead **FableManager** provides the interface for the presentation layer to indirectly interact with **Filter**.

Our use of the *strategy* design pattern in terms of filtering suggests a natural way to implement the censorship amendment. A censor simply provides an additional filter that is always activated when censorship is on and is never used when censorship is off.

A censor is defined by the **FableCensor** class, which contains a list of banned keywords, banned URLs, and password settings. Passwords are stored as a hashcode. To check if a password is valid, we simply hash the password string and compare the result against the stored hashcode. The hash function is a RSA-based encryption: Thus, it is extremely difficult, even knowing the hash function, to obtain the password

from the hashcode with current known techniques. Our design decision here provides a measure of security with regard to censorship settings.

We add an additional method to our **FilterStrategyFactory** factory class:

```
public static FilterStrategy acceptCensor (FableCensor x);
```

This allows us to easily generate the appropriate censorship **FilterStrategy** from the **FableCensor**. The advantage of this approach is twofold:

(1) There is no need to make a radical change to the existing design. We simply extend our existing filtering framework. In fact, this shows the ease of extensibility of our design.
(2) As the user changes banned keywords, banned URLs etc within the **FableCensor**, there is no need to re-scan all articles again to check for censorship, which can be an extremely slow and cumbersome process. Instead, we only need to change the inherent **FilterStrategy** used in censorship, which is smoothly integrated into the running of the program.

Further details of the integration of censorship with program's logic will be covered when we discuss the **FableManager** class (see **Interactions with Logic modules and Censorship**, in **The FableManager class**, below).

3.6 Searching

3.6.1 Searching Within Articles

Searching within Articles is done with the **RSSSearcher** class. An instance of an **RSSSearcher** class provides an interface to a *Lucene* searcher that stores the information in articles.

The **RSSSearcher** provides extremely fast searching of its articles. This allows incremental search to be performed simply by asking the **RSSSearcher** to search again every time the keyword is updated (not the computationally fastest method, but it simplifies the implementation and interface greatly).

RSSSearcher searches take in a keyword and returns a list of Articles that contain that keyword.

As with all other modules in the logic layer, interfacing with the **RSSSearcher** is done indirectly through the **FableManager**. For more details as to how this is done, refer to **Interactions with Logic modules and Censorship** (in **The FableManager class**, below).

3.6.2 Searching For Articles

One tool that a novice user would find extremely useful is to find feeds by searching for a particular keyword. For example, to find the CNN News feed, he could simply search for "CNN."

We provide this functionality through the use of the module **FeedsterSearcher**. This class provides an interface to Feedster search at http://feedfinder.feedster.com/ through which Fable integrates searching for feeds into its viewer.

The key method in **FeedsterSearcher** is:

```
public static List<FeedsterSearchData> FeedsterSearch(String keyString);
```

Through this method, clients can obtain a list of feeds that match the keyString to search for. **FeedsterSearchData** is an auxiliary data structure that stores the URL and title of the feeds that are matched.

Like every other module in the logic layer, **FeedsterSearcher** is interfaced through the **FableManager** class.

3.7 Exception Handling

As far as possible, exceptions are handled the moment they are caught, instead of propagating up to higher levels. This reduces code complexity at higher levels to prevent having to handle multiple kinds of exceptions.

Because exceptions are handled the moment they are caught, there is no need to have a user-defined Exception class that describes the kind of Exception.

3.8 The FableManager class

The **FableManager** class serves as the façade between the ADT and the GUI. In addition to handling direct ADT changes, it also handles sequences of logic and applies utilities such as filtering and searching to the data in the ADT. It also manages threads and is crucial to the overall smooth running of the program.

Because this is such an important class, we need to cover it in some detail.

3.8.1 Starting Fable

The main GUI initializes **FableManager** the moment it is loaded.

When Fable is started, the following things happen:

(1) The main FBML file is loaded – this loads both the root folder “Fable” as well as the censor. All descendents (direct and indirect) of this root folder, as well as the root folder, are activated so that they will fire events to the **MessageHandler** event handler.
(2) The root folder read is passed to the GUI through a request from the GUI after **FableManager** is loaded.
(3) The censor is used to create a **FilterStrategy** that will only allow censored documents to pass though.
(4) Threads are initialized and started.
(5) Censorship is turned on.

3.8.2 Stopping Fable

The main GUI will call **FableManager**’s close() method just before shutting down.

This causes the following to happen:

(1) All descendents of the root folder, as well as the root folder, are removed from the **MessageHandler** event handler so that they no longer have a link to the GUI.
(2) All threads are told to stop; however, they are allowed to process their final task before stopping.
(3) We only continue when we verify that no thread is still running a task
(4) We remove all **Articles** that have lasted longer than the cache time (set by the user) of their parent **Feed**
(5) We write the current settings, folder hierarchy and censorship options into the main FBML file

3.8.3 Threading

In addition to the main SWT GUI thread, which is implicitly started by the GUI classes, Fable runs only two other threads, both of which are instances of **FableThread**:

(1) Add thread – Adds a feed to a given Folder (Category)
(2) Refresh thread – Refreshes a particular feed

The **FableManager** class maintains a set of currentlyModifiedFeeds, which are the feeds that it is currently modifying or refreshing.

When Fable is told to add a feed, we add an empty feed to the folder it should belong to, add this feed to the currentlyModifiedFeeds list, and the add thread is given the following instructions, in sequence:

(1) **FableNonGUITask** – Load the contents of the feed into a temporary data object, by delegating to the appropriate I/O module. However, the data structure of the feed is not modified at all.

(2) For every single article in this feed, do the following:
   a. **FableNonGUITask** – Load its article contents into a temporary data structure
   b. **Runnable** – Add the article contents to the feed
(3) **Runnable** – Remove the feed from the set of currentlyModifiedFeeds

Slow steps (1) and (2a) have to be **FableNonGUITasks** to prevent decreased usability due to lag. This means that they cannot operate on objects (Feeds or Articles) that the GUI accesses. Hence they have to operate on temporary structures.

Step (2b) acts on objects that may be displayed within the GUI and hence this action can only be undertaken within the main SWT GUI thread. Step (3) is extremely fast and hence can be undertaken within the main SWT GUI thread as well.

Because it is important that these steps are executed in order, they are all executed sequentially in the Add **FableThread**, instead of delegating them separately to the Add thread and the SWT GUI thread.

Similarly, when told to refresh a Feed, if the Feed is in the set of currentlyModifiedFeeds, the **FableManager** does nothing (because the result of the refresh would be exactly the same as the result of the current add/refresh it is undergoing), otherwise the refresh thread is given a series of instructions similar to the add thread above.

Simultaneous access by one object on any two threads is avoided:

Main SWT Thread and Add Thread:

The Main SWT Thread only deals with objects that are linked to the GUI. The Add Thread will only deal with objects that are NOT linked to the GUI. Basically, the Add Thread will do computationally intensive tasks to load data from file or from the World Wide Web, and when this data finally needs to be loaded into the GUI (which is simply adding a reference to the created object), delegates the loading to the Main SWT Thread.

Main SWT Thread and Refresh Thread:

The explanation for this is similar to the above.

Add Thread and Refresh Thread:

The only possible way for these threads to act on the same object is for the same Feed to be added and refreshed at the same time (one way this could be done, for example, is the user adds a feed, and before the feed is fully loaded, selects “Refresh All”). However, the **FableManager** class prevents this from happening through its maintenance of a currentlyModifiedFeeds set. Feeds in this set will not be delegated to

either the Add or Refresh thread to be modified. Hence there will not be simultaneous access of the same object.

Because there are only three threads running at any one time, and no two of these threads can be accessing the same object simultaneously, simultaneous access is avoided even without object locks (which can lead to deadlock).

3.8.4 Interactions with the GUI

As **FableManager**'s main purpose is to serve as a façade between the GUI and the logic utilities and the ADT, its main interaction with the GUI is simply to take in requests for information.

FableManager is initialized by the GUI and is closed by the GUI. When it receives a request for information from the GUI, it delegates the task to the appropriate module and acts as an intermediary to return the results to the GUI.

FableManager may also take in requests to refresh feeds (whether via the GUI auto-refresh mechanism or by the user's manual refresh request) and add feeds / folders (categories), during which it creates tasks which delegate the I/O modules to read information from the feed and add it to the ADT. These tasks are then appropriated to the correct thread (see **Threading**, above).

FableManager never makes any request from the GUI. It can only take in requests from the GUI. This means that it has no dependence on the GUI and coupling is greatly reduced. **FableManager** does not tell the GUI when feeds / articles are successfully added. The GUI is able to auto-update itself through its use of listeners to events fired by the ADT via the **MessageHandler** observer (see **Event Handling**, above under **Logic Overview**).

3.8.5 Interactions with the ADT

FableManager interacts with the ADT in two main ways:

Requesting information

This happens when the presentation layer requests for some data. If the data needs to be processed, **FableManager** handles the required logic and delegates it to logic modules if needed. In either case, **FableManager** will use the ADT's accessor methods to obtain the required raw data.

Changing the information within the ADT

For requests that do not require I/O processing e.g. renaming, the request is directly delegated to the ADT mutator methods.

For requests that also require I/O processing and thread management, such as adding and refreshing of feeds, **FableManager** does the required logic management and delegates the required I/O and processing tasks to the appropriate threads (see **Threading**, above).

In general, the behavior with regard to adding/refreshing different ADT types is as follows:

> **Feeds:** Add an empty feed to the parent folder, while loading the articles of these feeds. These articles are then loaded individually into the enclosing feed.
>
> **Articles:** Articles are loaded to completion before they are added to their parent feeds.

3.8.6 Interactions with Input / Output modules and caching

3.8.1 Loading Feeds

FableManager interacts with the **RSSManager** class to read Feeds. These feeds may either be from the World Wide Web or on the local file system. There are two points to note:

(1) **RSSManager** takes in a **java.net.URL** address, not a string. Hence, local addresses on the file system are converted to **java.net.URL** objects using the "file" protocol
(2) **RSSManager** returns a **Feed** object that contains the descriptions of the Articles contained within it, but not the html content. Hence, the html content of these Articles has to be loaded separately.

3.8.2 Loading and Storing Articles / Caching

A **Feed** contains URLs to its Articles. Fable loads these Articles from the World Wide Web through **java.net.URL**'s getContent() method.

Fable caches Articles on the local file system. The management of articles is through the **ArticleManager** class, which handles the storage of Articles on the local file system as well as extracts their keywords (for censorship).

The **ArticleManager** class provides several static methods:

```
public static boolean setContents (Article a, String articleContent);
```

Sets the html content of an article by modifying the file on the local file system.

```
public static String getContents (Article a);
```

Gets the html content of an article from the local file system.

```
public static boolean removeArticle (Article a);
```

Removes the article from the local file system.

```
public static Set<String> generateKeywords (Article a);
```

Generates the set of keywords contained in this article.

With regard to caching, articles that are too old are removed only when the user exits Fable. This is part of **FableManager**'s job when it is closed (See **Stopping Fable**, above). The reason for this design decision was usability: Users should not have an Article suddenly deleted when they were browsing it.

3.8.3 Creating Custom Feeds

One feature in Fable is the ability for the user to create his own Custom Feeds (or Binders) and export Articles from other Feeds to them. In his own Custom Feeds, he can delete Articles or edit their descriptions and titles.

FableManager stores Custom Feeds on the local file system and uses **RSSManager** to write these Feeds to file.

In addition, **FableManager** handles the extra logic needed whenever Articles in Custom Feeds are edited or deleted by modifying the file on the local file system using methods provided in the **RSSManager** class.

3.8.4 Importing/Exporting a List of Feeds as OPML

Another feature in Fable is that a user is able to load and export a list of feeds in OPML format. **FableManager** interacts with the **OPMLManager** class to do the following:

(1) Read OPML: Provides an interface to obtain a list of Feed URLs and their titles from an OPML file
(2) Write OPML: Create an OPML file on the local file system from a list of **Feed**s.

3.8.5 Importing/Exporting settings as FBML

FBML import and export is done through **FableManager**'s interaction with **FBMLManager**.

There are two ways **FableManager** exports FBML:

(1) When the program saves FBML at the closing of **FableManager** (see **Stopping Fable**, above), it saves all the article information within the FBML file.

(2) When the user exports FBML, article information is not saved: Only the hierarchy of feeds and folders (no articles) as well as refresh/cache settings and censorship settings are saved.

Importing FBML of both types [(1) and (2) above] is universal. **FBMLManager** is able to read FBML of both types. When an FBML file is imported, the existing settings are completely removed and the new settings take precedence. When FBML is imported, **FableManager** takes the following steps:

(1) Use **FBMLManager** to load the FBML file into a temporary *root* Folder
(2) Remove all children of the current *root* Folder, including removing all connections they have to the presentation layer
(3) Add all children of the temporary *root* Folder to the current *root* Folder and attach them to the presentation layer via the **MessageHandler**.

Notice that the *root* Folder itself is never removed.

3.8.6 Exporting a series of Articles as PDF

One useful feature that a user might want is to be able to export a set of Article summaries in PDF format, so he can get at a cursory glance important Articles he is reading. This can also be passed to friends via email.

FableManager interacts with the **ExportDocumentManager** class in order to produce such PDF summaries.

3.8.7 Telling friends about Articles

Another useful feature we have is that the user might see some interesting articles and would want to tell his friends about them.

We have an easy way to do that, through email. **FableManager** interacts with the **MailSender** class in order to do so. We also provide the option for sending PDF summaries with these emails as attachment, and this is done via the interaction of **FableManager** with **ExportDocumentManager**.

3.8.8 Interactions with Logic modules and Censorship

For searching of feeds on the World Wide Web through a keyword, **FableManager** simply delegates the searching to **FeedsterSearcher** directly without any additional logic.

Searching of articles is done directly through **RSSSearcher** as well with the special case that if the list of Articles to be searched changes, a new **RSSSearcher** is created, otherwise the search is done with the existing **RSSSearcher** instance.

Filtering is integrated with Censorship. If Censorship is turned on, we use **FilterStrategyFactory** to combine the filter derived from the current **FableCensor** together with the current filter selected by the user (the default filter is to View "All Articles". Other filters available include "View unread only," "View last 7 days," etc). If Censorship is turned off, we use only the current filter selected by the user. This resultant **FilterStrategy** obtained is then passed to the **Filter** class's filterArticles method which will then filter Articles. **FableManager** will pass the filtered list to the presentation layer which will then display only the Articles that are accepted by the filter.

In this way, the presentation layer is completely unaware of the Censorship settings. All logic to do with Censorship is completely handled by the **FableManager** class. This decoupling of censorship issues from the presentation layer caused by the integration of filtering with censorship allows for a far more modular approach, as well as the easy extension of existing features to allow for censorship.

To turn Censorship on and off, as well as add/remove banned keywords and URLs, or to view censored Feeds or Articles, the user needs to enter the master account password (If the user wants to change the master password, he will also need to enter it). However, for the user to keep having to enter the password is horribly inconvenient and is bad in terms of usability. Equally bad from a security perspective would be to have the user only need to enter the password once throughout the entire Fable session.

We achieve a happy medium by authenticating the account every time the user enters the master password for ten minutes. **FableManager** achieves this by storing the last time the password was authenticated.

The user might also want to force authentication to occur. **FableManager** provides an interface for this to occur as well.

3.9 Package Overview

This is an overview of the various packages:

fable.adt	Contains all classes related to the Abstract Data Type(s) used as well as auxiliary data types such as the **FableCensor** class.
fable.adt.event	Contains all classes dealing with events fired by ADTs when the underlying representation is mutated such as **StructuralChangeEvent** and **ArticleMarkChangeEvent** as well as the listeners dealing with these events.
fable.gui	Contains all classes related to the layout of main components of the GUI. Also contains factory classes: ImageFactory and FontFactory which creates and interns all images and fonts respectively for the GUI.

fable.gui.dialog	Contains all dialog boxes used in the by the GUI. Also contains all the data objects used by these dialog boxes to pass data back to the GUI component which called it. There is also a useful DialogUtility factory class that helps to provide buttons, treeviews etc for these dialog boxes.
fable.gui.event	Contains all the listener classes that are used by the MessageHandler classes to facilitate interaction between different GUI components.
fable.io	Contains classes which facilitates the export and import of OPML and FBML files. Also contains classes which handle the parsing of XML and RSS files.
fable.test	Contains JUnit test suites that are used to verify the correctness of the ADT and model classes.
fable.thread	Contains classes which manage the threading behavior of the GUI.
fable.utility	Contains classes which manage logic handling as well as utility procedures such as searching for feeds on the Internet and sending of mail
fable.utility.strategy	Contains classes which deal with strategies for filtering (Classes that implement **FilterStrategy**) as well as the **FilterStrategyFactory** factory class that creates these strategies.

The package organization allowed us to logically split the workload in implementation.

3.10 Implementation of various layers

In this section, we go through the design of the various layers that were introduced in the first section individually.

3.10.1 Data Layer

Abstract Data Type

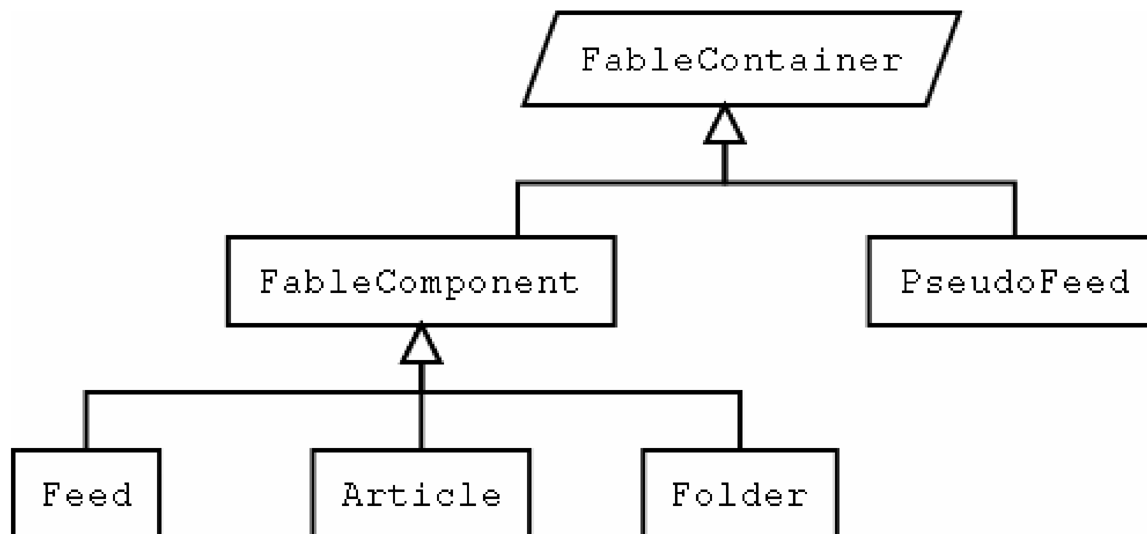
There are four main abstract classes to store data:

Feed, Folder, Article, PseudoFeed.

To implement their abstract relationship (see **Data Layer**, under **General Overview** above) concretely, we utilize the *Composite* design pattern.

Every single one of these concrete classes is a subclass of the **FableContainer** class, which is an abstract class that stores instances of the **FableComponent** class as children.

The inheritance hierarchy is as follows:



The **FableComponent** class, which is the abstract base class for **Feeds, Folders** and **Articles** may contain other **FableComponent(s)** as children, and in this way employs the *Composite* design pattern. It also defines several basic fields used in all of its children: Date created and title.

The **FableComponent** class also defines two methods as follows:

```
public abstract void accept( FableVisitor v );
public abstract void acceptRecursive( FableVisitor v );
```

This is the utilization of the *visitor* design pattern that allows a single visitor to traverse all the children and grandchildren etc. of the **FableComponent**. The difference between accept and acceptRecursive is that accept does not pass the visitor to its children (meaning no traversal), while acceptRecursive does. **FableVisitor** is the visitor class that visits and performs actions on the **FableComponents**. This v*isitor* design pattern is used extensively in **FableManager** to perform operations on various **FableComponents**.

The **Folder** class is simply a wrapper around a **FableComponent**.

The **Feed** class has the following additional fields: URL, RefreshRate, and CacheTime, which denotes the feed URL, the refresh rate and the time to cache its articles respectively.

The **Article** class has a few additional fields: In addition to defining the URL, it also defines the author, summary, and content fields. It contains the set of words stored in its summary and html content so that this can be used for censorship options. It also has a set of *Marks* – used to mark the **Article** as read/unread or starred/unstarred (starred/unstarred referring to favorite articles).

The **PseudoFeed** class is simply a container for storing **FableComponents**, but it is mainly used to store **Articles**.

The overall advantage of using a heavyweight *Composite* design pattern is that it allows various operations to be easily done on all its children components using the *Visitor* design pattern. This makes the code far simpler and shorter and the design more modular. It also makes it very extensible: As we wish to add features, we have to add operations that work on every single ADT object, and this is easily done through the use of the *Visitor* design pattern.

Input/Output Layer

The Input/Output Layer consists of the following main modules:

ArticleManager, ExportDocumentManager, FBMLManager, OPMLManager, RSSManager

Refer to the section above on **Interactions with Input / Output and Caching** (under **The FableManager class**, above)

CustomName

This module allows us to check the file system and look for an available name to save Articles / Custom Feeds / Summaries to. For example, supposing we have a Custom Feed named "Slashdot" and we wish to save it as "Slashdot." However, if a file named "Slashdot" already exists we are unable to create this new file. This module checks the local file system and generates a suitable name for a new file.

Parser

This module consists of a single static method:

```
public static String makeLegalXML(String s);
```

This module is a helper module to convert a String to legal XML. This is used in **RSSManager**, **OPMLManager** and **FBMLManager** to write titles, summaries etc. as legal XML.

Censor

The **FableCensor** class contains methods to add and remove banned keywords and URL, as well as to check if a password is acceptable and to change the password.

Of interest are three methods:

```
public static String hash (String password);
public boolean isOkArticleContent (Article a);
public boolean isOkURL (String url);
```

The first method is a static method to hash the password. As mentioned previously, we use a version of RSA for a one-way hash. What is interesting about this method is that it can be replaced by any deterministic non-random hash function and **FableCensor** will still work exactly the way it should be. This means it is highly flexible: If we wish to improve the strength of how **FableCensor** protects its password, we simply only need to change this one method without modifying any other methods.

The second two methods provide an interface so that **FableCensor** instances can verify if Article(s) or URLs are valid under their Censor. This provides an abstraction so that clients need not know how the banned Keywords and URLs are stored. To check if an Article or a URL is banned, they only need to use these methods. These methods are also what is used by **FilterStrategyFactory**'s acceptCensor() method, which generates a **FilterStrategy** from a **FableCensor**. This allows for flexibility so that the underlying concrete data structure can be modified easily without changing the effective functionality of a **FableCensor**.

3.10.2 Logic Layer

The Logic layer mainly consists of the **Filter**, **RSSSearcher**, **MailSender, FeedsterSearcher**, and **FableManager** classes, as well as the **FilterStrategy** interface and **FilterStrategyFactory** factory class that have already been described in previous sections.

The following classes implement the **FilterStrategy** interface:

AuthorFilter	Filters Articles according to their author
DateFilter	Filters Articles according to their Date
MarkFilter	Filters Articles according to their Mark (read/unread or starred/unstarred)
CombinedFilter	Combines various other **FilterStrategy** filters

The **FilterStrategyFactory** class, as mentioned before, provides a far more intuitive way of creating these filters. This is another advantage of using the *factory* design pattern.

For more information on the Logic layer, refer to **Logic Overview** and **The FableManager Class** above.

3.10.3 Presentation Layer

Fable uses a modified *Model-View-Controller* architecture/design pattern, where the View and Controller are integrated within the presentation layer and the Model is represented in the data layer. The logic layer serves as an intermediary between the presentation and data layers to process data to a form suitable for presentation. This interface is represented by the **FableManager** class. Within the presentation layer, the *Controller* processes send requests to **FableManager** either for information (either for raw data or data processed by the logic layer) or to request modifications to the data. The *View* processes interact indirectly with the ADT through the use of the **MessageHandler** global observer that provides a façade between events fired by the ADT and listeners for these events in the *View* processes (see **Event Handling** in **Logic Overview** above). This decouples the model from the controller and the view and thus reduces dependencies greatly.

The Graphical User Interface is divided into several components. **FableGUI** is the main class, which contains the menu, and toolbar. It instantiates and lays out all other component classes in the GUI, and also the **FableManager** class that handles all changes to the GUI.

The main components of the GUI that are initialized by **FableGUI** include: **FeedView** (the panel showing the list of articles in the current feed and a summary of the article currently selected), **BrowserView** (the embedded browser), **TabView** (manages the tabbed behavior, manages the open/close behavior of **FeedView** and **BrowserView**), **FeedListTree** (the hierarchical feed/binder/category tree in the left panel), and **StatusBar** (the status bar of the main GUI).

Each of the above components inherits from an Abstract class of the same name. We made this design choice, in order to logically divide up the interaction of the GUI. Interactions of controls within a particular component all reside in the Abstract classes, which interaction between these components are handled in the concrete classes. For example, in the **FeedView**, when a user clicks one article in the list, and the summary of that article is shown, this is handled by the Abstract class, because it is an interaction within the component. On the other hand, when a feed is selected in the **FeedListTree**, the corresponding list of articles contained in this feed should be displayed in **FeedView**. This interaction is handled by the concrete class, since it is an interaction between two components. This allows a clear logical separation of local interaction and inter-componential interaction, and allows the code to be more easily managed.

Another effort we made in modularizing our GUI is to use the *factory* design pattern: We created two factory classes called **ImageFactory** and **FontFactory**. These two classes handled all images and fonts (respectively) in the GUI. In this way, each image or font used is only initialized once, and can be disposed easily when the program terminates (SWT requires manual disposal/garbage collecting of all user created Image and Font objects). These factory classes allow us to use memory effectively and create

a predictable routine for image and font usage. It is makes it extremely flexible. For example, if we wished to change the image for “Feed,” we would not need to change the code in everywhere that needed that image; we would only need to change the code in **ImageFactory**.

4 Design Decisions

Here we will elaborate on the many choices we had to make in the design process: the rationale for our choices, and why we rejected the alternatives.

4.1 ADT and Data storage issues

4.1.1 Storing of HTML content of Articles in memory

Since we had to cache HTML content of Articles, the natural question to ask was if we should store the HTML content of Articles in memory during the running of Fable. It was decided that because the HTML content of articles could be exceedingly large (100kb in some articles), we would not store it in memory.

Instead, the HTML content would be stored in the file system. Although reading from file would be time consuming, reading this file would only be needed when the user wished to browse the article, which was comparatively rare and in any case would be much faster than loading it straight from the World Wide Web.

With regard to Censorship, the only thing needed in an Article was the list of keywords that it had. We could store this in the ADT as a set of Keywords. This would be far smaller than the HTML because of two reasons: Firstly, many extraneous HTML tags, especially comment and script tags would not be present. Secondly, words that were duplicated many times, such as common English words and technical jargon in technical sites, would only be included once.

As such, it was extremely economical to store only the set of Keywords while keeping the actual HTML content stored in the local file system.

4.1.2 Lazy Evaluation and Database

Another issue was Lazy Evaluation: Should we load every single article at the start, or load them as they are needed? Initially, we were thinking of using lazy evaluation, but because we decided that since searching would require us to access every single article, it would be more prudent to load every single article at the start.

The amendment, in effect, forced us to load every article at the start, including their HTML content, because we had to censor every article’s HTML content. Because of this, lazy evaluation was totally discarded as an alternative.

Yet another question was the issue of using a database. How would the HTML content be stored on the local file system? Should we use a database, or should we keep separate files for each Article?

Initially we used a Derby database to store html content. However, we realized that the advantages provided were minimal and there were more disadvantages using a database. Hence we switched to keeping separate files for each Article.

The main advantage of a database was that reading information was faster than reading from a file from the hard disk. However, we had already stored most of the relevant information in memory and the only information not stored was the Article HTML content, which would only be accessed relatively rarely. The rest of the information stored was relatively low in size, and hence this did not present a problem of scalability.

Another advantage of a database was that it allowed lazy evaluation and "read only when necessary." However, since this was made much harder by the amendment, we felt that this advantage was no use to us.

In contrast, a database had two disadvantages that we felt were important to us. The first was the memory footprint of the database, which was relatively large enough to discount any advantage it might have of allowing us to store less information (we would have to store a lot of information in memory anyway, so this advantage is not great). The second was that database access was far slower than memory access. Since we had already stored most of the relevant information in memory, there would be no need for database access.

We felt that because of these reasons, using a database would slow us down unnecessarily while taking up more memory, so we chose the route of storing each article's HTML content in its own individual file.

4.1.3 When to show Articles from the Cache

There were two options here: Either to show articles always from the cache, or to show articles from the World Wide Web whenever possible and only show from the cache when there was no internet access. We chose the latter.

This enabled the user to always obtain an up-to-date version of the article when necessary. We updated the cache whenever the user obtained this up-to-date version, ensuring that the cache was always updated as well. When there was no internet access, however, the cached page was shown, allowing the user to view the latest version of the Article he had ever read.

4.1.4 Handling of PseudoFeeds as a Class

Where should the **PseudoFeed** object be in the overall hierarchy? We felt that because the main goal of a **PseudoFeed** was to store **Articles**, a kind of **FableComponent**, it

should be a **FableContainer**. In addition, because of the similarity of **PseudoFeeds** and **Feeds** in terms of them being a container of **Articles**, they were substitutable for each other as a **FableContainer**. This allowed us to create methods that could take in **PseudoFeeds** and **Feeds** as parameters instead of having to overload every single one of them.

4.2 Logic Issues

4.2.1 Exception Handling

Should exceptions be handled when they are caught, or should they be propagated up to higher levels where they can be handled in the overall situational context?

We decided to handle exceptions when they were caught as far as possible. There were two reasons for this. Firstly, we felt that the occasions that propagation of exceptions would lead to a better result was rare. Exceptions usually happened atomically and thus were rarely part of a larger picture. There would be thus little advantage in propagating exceptions. Secondly, propagating exceptions upwards would result in extremely complicated code at top-level modules where exceptions of many different types would have to be caught and handled. In addition, propagation of exceptions upwards would generally have a detrimental effect on the running speed of the program, which could be bad to usability if the user noticed lag. For code simplicity and general ease of logic, we thus decided to handle exceptions when they were caught.

Because exceptions are not propagated upwards, there is no need for us to wrap exceptions within our own user-defined exception to provide more detail about the exception to higher-level modules.

4.2.2 Incremental Search

We decided to include incremental search as a feature because it was useful and welcome feature to users.

In addition, the strength of the Lucene search package is that it allows you to perform multiple searches on the same set of objects in relatively quick time. The bottleneck in searching lay in initializing the set of objects for the first search. This meant that performing incremental search by simply repeatedly searching the same set of objects for increasingly long (or short) key strings was extremely efficient and this incremental search was not noticeably slow even for a large number of articles.

Since we could provide such a useful feature at a low time cost, we implemented incremental search as the Searching method instead of simply doing standard Keyword Searching.

4.2.3 Threading

Instead of implementing multiple threads, each updating a single feed, we only had two threads, one to add feeds and one to refresh feeds. This simplified threading issues greatly.

Because we only had two threads, we could afford to not provide object locks on feeds and instead handle manual locking through the **FableManager** class, as we could analyze the behavior of the two threads separately instead of having to analyze many different threads. This prevented deadlocks from happening as well as simplified the **FableThread** class.

The use of threads allowed program running to be efficient without noticeable lag to the user. This improved usability greatly, especially when the user refreshed a great number of feeds (e.g. by clicking on "Refresh All") or added a great number of feeds (e.g. by adding many feeds obtained from a Feedster Search).

4.2.4 Implementation of Censorship

One possible way of implementing censorship would be for every Article and Feed to keep an additional flag to check if it was censored and update this flag when the **FableCensor** was modified.

However, this had two problems: Firstly, we would have to look through every single article every time the **FableCensor** was modified. With potentially hundreds of articles, and disk load time relatively slow, this would cause a massive delay every time the **FableCensor** was modified. Secondly, this would mean that we would need a non-minor code revamp.

Since we had already incorporated filters into our code, censorship was simply a logical extension of this filter, and this extension would be relatively easy to code up due to the flexibility of our design. We would only need to add an additional field to the **Article** class – a set of keywords that it contained. As elaborated before, this set would likely be small and would not be detrimental to the scalability of the program.

In addition, incorporating a censorship filter had the additional advantage that since it was an extension of an existing feature, it kept the code relatively neat and easy to maintain as well as remained flexible for possible modifications and changes to the censorship feature.

4.2.5 Event Handling and MessageHandler Observer Singleton

Instead of having every single GUI object register itself as a listener to the relevant ADT object(s), we use a **MessageHandler** class as an observer. We only need for every GUI object to register itself as an event listener. ADT objects will fire events to this **MessageHandler** class that also acts as a façade between the ADT and the GUI. This class will take every event it receives and dispatch it to the correct event listener.

To keep the design and the code simple, this **MessageHandler** class also uses the *Singleton* design pattern so that it is the only global observer.

The use of an *observer* design pattern here reduces coupling between the GUI and the ADT by the use of a façade, and keeps code and logic management simple, as well as provides extensibility and flexibility.

4.2.6 Separating Manager and Observer and the use of Façades

The **MessageHandler** class mentioned in the previous design decision was part of the façade between the GUI and the ADT. The façade is completed through the **FableManager** class.

The decision to separate the logic management and the event handling in the façade was because we felt that we could decouple the two easily without adding extra complexity. This meant that the code complexity would be reduced and the dependencies on the individual façade classes would be weaker than if they were combined, leading to improved code maintainability.

The use of a façade design pattern allowed us to modify and add features in an extremely systematic way, without having to check every single class when modifications and additions occur.

4.3 GUI and Usability Issues

4.3.1 SWT vs. Swing

We chose to use SWT/JFace to build our Graphical User Interface, because of its speed and visual appearance. Because SWT leverages as much of the Operating System's native API as possible in drawing its widgets, it is much faster than Swing. Because of this reason, SWT also matches the appearance of the native OS, and has a more consistent look-and-feel overall. Swing in contrast, in slower and does not have this visual consistency. However, SWT was not familiar to us, so a great of deal of time was invested in learning to use it.

4.3.2 Naming Conventions

One important issue in usability is to not use technical jargon and instead employ vocabulary that is familiar and intuitive to the user. To this end, we spent quite some time thinking of good names for our toolbar buttons, menus, and context menus. We chose to retain the name of "feeds", because it is intuitive. "Folder" is a name that is a bit misleading, because usually one associates folder as a container of files, so we renamed it as "category". "Custom Feed" was renamed as "binder" because it is really a custom container of articles, and the word "binder" conveys this meaning.

4.3.3 Tabbing – the use of Multiple Tabs

Tabbed viewing has become a popular feature of many recent software applications. We wanted to leverage its power in a way which retains the most flexibility while staying consistent to the expected behaviors that most users are used to. To this end, we decided to have only a single tab that shows the Feed/Article view. This would be consistent with the behavior of standard email and news clients. However, we also decided to allow multiple tabs for browser. This consideration was based on the fact that most browsers today support this feature. It also makes sense from a common usability point of view. Users should have the ability to open several articles at once and switch between them. This allows the user to easily compare and contrast the content of different full sources.

4.3.4 Usability of the dialog used to Add New Feeds

There are several ways to add new feeds. One is to type in the URL, another is to load local RSS file, and the third way is to use our search feature to find feed by keyword. We decided to integrate all three methods into the Add New Feed dialog box. The user can select via radio buttons, which of the three methods he wants to use to add a feed. Giving the user several options allows flexibility and open ended usage. Our design is also careful to not confuse the user, by making the choices clear.

4.3.5 Showing articles in a folder

One somewhat unusual decision we made is to show articles indirectly contained within a folder. Folders cannot directly contain articles, but when a user clicks on a folder, all articles contained within the feeds and folders inside the parent folder is displayed in the Feed/Article view. In other words, Folders and Feeds are like transparent containers of article which allow their parent in the hierarchy to look into them.

This decision is different from common behavior in a mail/news client, but we think it makes a lot of sense. Users may well want to see all articles within a given folder. The other possible behavior, of showing a blank view when a folder is clicked, offers no useful information.

4.3.6 Status Bar

A status bar gives the user some indication of the current state of the program. Although it takes up some space in the screen real estate, it provides valuable information to the user. On the status bar, we have several status indications. First, there is an indication of whether feeds are currently being updated or not. Also, there is a description of the last folder/feed/binder/article that has been selected or deleted. Finally, there are a couple of icons that show whether censorship has been enabled, and whether the user is logged in or out (if a password is entered, for a period of 10 minutes the user is considered logged in).

5 Testing

5.1 Testing of Model (ADT)

The Abstract Data Type classes were tested using JUnit.

This was tested extremely thoroughly because it was the base of the entire software.

Our strategy was to test each method according to its specification. We made sure that every possible logic flow was tested. Finally we tested the tree structure operations fairly extensively -- tried to make variety of invalid tree structure such as cyclic tree, children having more than one parent, setting parent for illegal position, etc.

5.2 Testing of Input / Output

The OPML, RSS, and FBML reading and writing were tested using JUnit.

We tested two things:

1. Reading: Read a known file and check if it is read correctly.
2. Writing: Write known data to a file. Re-read this file to check if the output written is correct.

5.3 Testing of Logic modules

We tested the searching and filtering modules using JUnit.

For searching within articles, we tested two things: Ensuring that keywords that did occur in the article were matched, and ensuring that no false positives (articles returned that did not contain the keyword) happened.

For searching for feeds on the World Wide Web, we checked that the searches for the keywords “Slashdot” and “CNN” returned the correct sites.

For searching for filters, we tested that **FilterStrategyFactory** did indeed return the correct type of filters and that these filters were able to filter out articles correctly.

5.4 Testing of GUI and Functional Features

In this section, we list a series of tests that we did to test the efficiency and working of the GUI and features.

Adding valid Feed (Names):

1. Click on “New Feed” icon in toolbar. Or select File | New | New Feed in the menu.
2. Type in the Name and URL, and select parent folder
3. Click on the OK button
4. The Feed appears under the specified location, and has a parenthesis around it indicating the number of unread messages.

Adding invalid Feed
1. Apply same procedure as above, except enter an invalid URL.
2. An error message will come up.

Viewing Article
1. Click on a Feed/Binder/Category.
2. The list of articles contained within is displayed in the top right panel.
3. Click on an article in the right top panel shows the information and content of the article in the right bottom panel.

Delete Content
1. Select a Feed/Binder/Category.
2. Click on the Delete button in the toolbar, or select File | Delete in the menu, or use the “Delete” key on the keyboard.
3. A confirmation dialog pops up, click “Yes”, or “No”
4. If “Yes”, the selected item should be deleted. If “No” it should be preserved.

Renaming
1. Select a Feed/Binder/Category
2. Right clicking and selecting context menu “Rename”
3. Change the name. Make sure that spaces do not cause an error
4. When “Enter” is pressed or the focus is moved away from the selected Feed/Binder/Category, the new name should be shown on the Tree and the Article Unread count should be restored.

Article Unread Count
1. Unread count decreases by one every time an unread article is read either by clicking it or by selecting it by the arrow keys

Marking Articles
1. Clicking on an article should make it appear as unread (in terms of view, it should be un-bolded)
2. Right clicking and selecting context menu “Mark as unread” should make article(s) unread again (in terms of view, it should be bolded).
3. Clicking on the star icon in the left most column should change an the icon either from unstarred to starred, or vice versa.
4. Right clicking and selecting in context menu “Mark as starred” or “Mark as unstarred” should mark the article(s) as either starred or unstarred respectively (visible by the icon in the left most column).

5. Closing the program, and restarting it, all the markings should be the same as before. This tests that the markings are actually applied to the ADT.

Censorship

1. Enter a password the first time Fable is started.
2. The next time the password prompt dialog comes up, entering the correct password should be accepted and entering the incorrect password should result in some error.
3. The password prompt dialog should come up when clicking on Censorship Off button.
4. The password prompt dialog should come up when clicking on Edit | Change Censorship Setting in the menu
5. After entering the password correctly, no password prompt dialog should come up, in the next 10 minutes.
6. After 10 minutes, when you try to change censorship setting or turn it off, the password prompt should come up again.
7. Select Force Authenticate button should reset the 10 minute grace period to 0. So when clicking on Change Censorship Setting or turn Censorship Off the next time, the password prompt dialog should come up again.

Filter View

1. Select “All Articles” from the filter view should allow all (uncensored) articles to be visible in the panel.
2. Select “Last 7 days” from the filter view should only display (uncensored) articles whose date is within the last 7 days.
3. Select “Unread only” from the filter view should only display (uncensored) articles which are marked unread.
4. Select “Starred only” from the filter view should only display (uncensored) articles which are marked as starred.
5. Select “Custom”, and a dialog box will come up, allowing different categories to be filtered. After this, the correct results should appear.

Incremental Search

1. Start typing a word in the search box, after a brief delay, all articles matching the query should appear.
2. As more words are entered, or the query is refined, the search result should change instantaneously.

Feed Search

1. Click on Search button in the toolbar.
2. Type a keyword in the dialog that appears and click search.
3. A list of feeds matching the keyword will be returned.
4. Select (by checkbox) several of these, and click on the category under which they are to be added.
5. Click OK. These feeds should be added to the appropriate category.

Import/Export OPML

1. Click on File | Export | to OPML file
2. Select the location where the file is to be exported, and select which feeds are to be exported.
3. Click OK
4. Click File | Import | from OPML file
5. Select the category into which feeds are to be imported.
6. The imported feeds should be the same as the exported ones.

Import/Export FBML (settings)

1. Using a similar procedure as above.
2. First export the FBML file, and then import it and compare results.

Custom Binders

1. Create a binder by selecting “New Binder” on the toolbar
2. Add articles to this binder by either dragging and dropping or right clicking and using the context menu
3. Edit this custom binder by deleting articles or editing their contents and ensure that this changes immediately in the article viewpane.

Export PDF file

1. Select several articles
2. Open the context menu by right-clicking, and select “View as PDF”
3. A browser window should open with this PDF. This PDF can be saved by clicking on the “Save” button within the browser.

Sending Email

1. Select several articles
2. Open the context menu by right-clicking, and select “Email Article(s)”
3. Try sending it both as text/html, and with or without the PDF attachment
4. You should be able to receive the mail using your mail client.

Drag and Drop

1. Dragging feeds into categories should be allowed, and behave as expected.
2. Dragging feeds into binders or binders into feeds should not be allowed.
3. Dragging categories into feeds or binders should not be allowed.
4. Dragging articles into binders should be allowed. But dragging articles into feeds or categories should not be allowed.

Status Bar

1. Click RefreshAll, and see if the feed status indicator is displaying the correct behavior in the status bar
2. Click Turn Censorship On/Off, and see if the correct icon is displayed in the status bar
3. Click Logout and see if the correct status is displayed in the status bar. Login again and see if the correct status is displayed in the status bar.

Stress Testing

1. Adding more than 20 feeds, and test performance.
2. Accumulating more than 500 articles, and test performance.

6 Reflections

6.1 Software Design Process

Due to the short time allocation of this project, our team chose to use the incremental approach to design. At the outset, we set specific milestones, with features incrementally being added at each stage, and tested to ensure stability and conformation to the specifications. This allowed us to periodically evaluate our current build, to determine if we had strayed from our initial specifications and if any changes needed to be made due to unforeseen issues. During the initial planning stages, we also discussed the list of features we wanted to implement beyond the basic requirements. Specifically, we thought about what features users would desire, in order to make their usage experience more pleasant.

As our design got underway, the ADT and the parsing module were quickly written and tested with little problem. The GUI, however, proved to be a big challenge. We realized that SWT was a different beast than Swing, and much more time needed to be devoted to learning it. As the preliminary release drew near, we were forced to scramble to complete the GUI. All four of us ended up doing some components of the GUI, because of the complexity of SWT.

The amendment proved to be a bit unexpected, but not a huge challenge for us. Because of our modularized design, the content censorship was achieved by applying a view filter over all items that are displayed. This was relatively easy to achieve due to the fact that we had already implemented a view filter feature. However, we were forced to revise another part of our design, since HTML pages must be loaded and scanned for conformity to the censorship standards. Initially we considered only caching HTML files after the user views it. Now it became necessary to cache all HTML files associated with articles automatically.

About 75% of the time, we were able to meet the milestones we set. And on occasions when we did not, the slip was no more than one or two days. However, when the due date was extended by one week, we had to make some adjusts to our schedule. With seven more days in the development cycle, we could take more time to fine tune usability and to implement one or two additional features. The lesson that we learned here is that milestones sometimes cannot be always met. The timeline must sometimes be adjusted to meet changing specifications and issues that come up.

6.2 Team Management

Our team consists of four members, with differing backgrounds in programming and working. We were not necessarily familiar with each other's work habits. At the start of the project, we divided up the tasks to be done: two of us would be devoted to the GUI, one member would work on the ADT, and the other on input/output processes. Our strategy at first was to work individually, and meet several times per week to exchange ideas and to integrate our code.

This strategy worked rather well in the beginning, as each person's tasks were quite separate from the others'. But by the second week of the design process, we felt the need to begin integrating our work. So a decision was made to work together for a period of three to four hours, in order to work toward that. This turned out to be a major success, as our efficiency was greatly increased as we were able to bounce ideas off one another, and to ask for help when stuck. We were also able to discuss any issues that arose and resolve them quickly. After this experience, we decided to work together as often as possible, as it allowed us to greatly expedite our work.

Because of our frequent meetings, we were very much synchronized as to the tasks to be done, and decisions to be made. This led to rather efficient design cycles, with little repetition of tasks or miscommunications. Also, because of our small team, decisions were made in a democratic fashion, with no significant disagreements.

6.3 Conclusion

As a team, we have learned a great deal from this project. We gained experience in programming in a team environment, having to deal with the reality of making decisions together and integrating the code of different people into a cohesive unit. In terms of the graphical user interface, we truly came to understand the time and effort involved in making an interface usable and elegant. Many small features that we often take for granted (such as sorting columns in a table) proved to be harder to implement than can be expected at first glance.

7 Appendix

7.1 External Libraries Used

SWT	Standard Widget Toolkit, used in all GUI classes to provide native look-and-feel
JFace	Provides some higher level functions built on top of SWT. Used in conjunction with SWT to create the GUI.
Xerces	XML Parsing library, used in **OPMLManager** and **FBMLManager** to parse XML.
NekoHTML	HTML Parsing library, used in **FeedsterSearcher** to parse the

	HTML output from Feedster
JDom	Needed for both **Xerces** and **NekoHTML**
Rome	Used to parse and write RSS feeds in RSS 0.9, RSS 1.0, RSS 2.0, and Atom 0.3 format, utilized in **RSSManager**.
Lucene	Library with searching utilities, used in **RSSSearcher**.
iText	Document library, used to export PDF and HTML files in **ExportDocumentManager**.
mailApi	Sun's JavaMail API, used for sending email

7.2 Milestones and Date Achieved

Stage 1: Leading up to Preliminary Release

- Specifications (for ADT, Database and File Operations): 11/9. Completed 11/8.
- Paper prototypes for GUI (for usability testing): 11/9. Completed 11/10.
- ADT + Unit tests: 11/12. Completed 11/12.
- * Database Operations + Unit tests: 11/12. Completed 11/10.
- File Operations: 11/15. Completed 11/14.
- Manager module: 11/15. Completed 11/15.
- Observer modules: 11/15. Completed 11/19.
- GUI Prototype (and running) code: 11/18. Completed 11/20
- Functional tests: 11/20. Completed 11/21

Stage 2: Leading up to final release

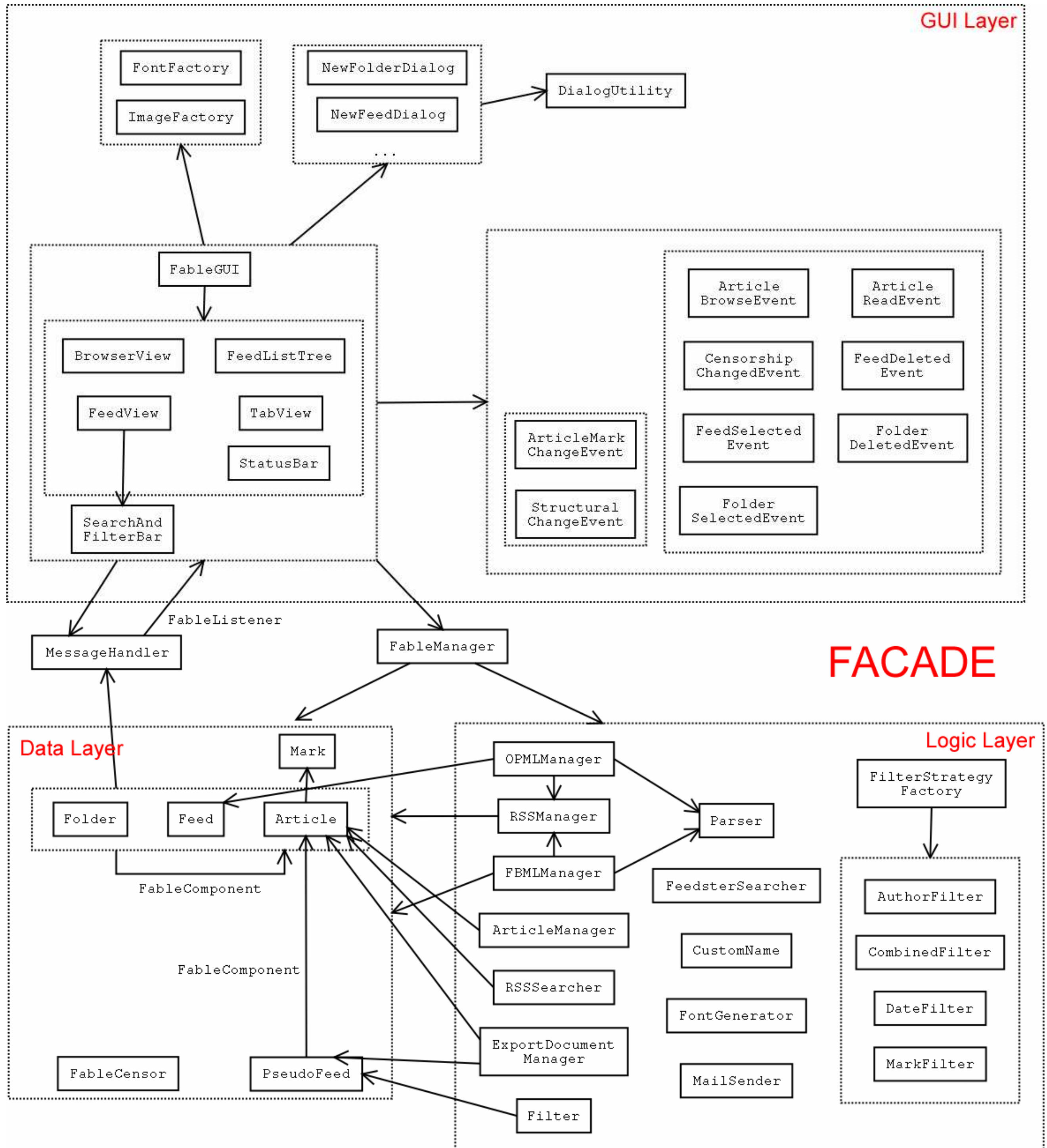
- GUI Feature Complete: 11/30. Completed 12/2
- Project Amendment: 11/30. Completed 11/27
- GUI Usability Beta Testing: 11/30. Completed 11/30.
- Advanced Features: 12/8. Completed 12/10
- GUI Usability Complete: 12/8. Completed 12/10
- Documentation and User Manual: 12/8. Completed 12/9
- Stress Testing: 12/8. Completed 12/8
- JAWS Packaging: 12/9. Completed 12/10
- Debugging: Up to 12/10. Completed 12/10

7.3 Work Allocation

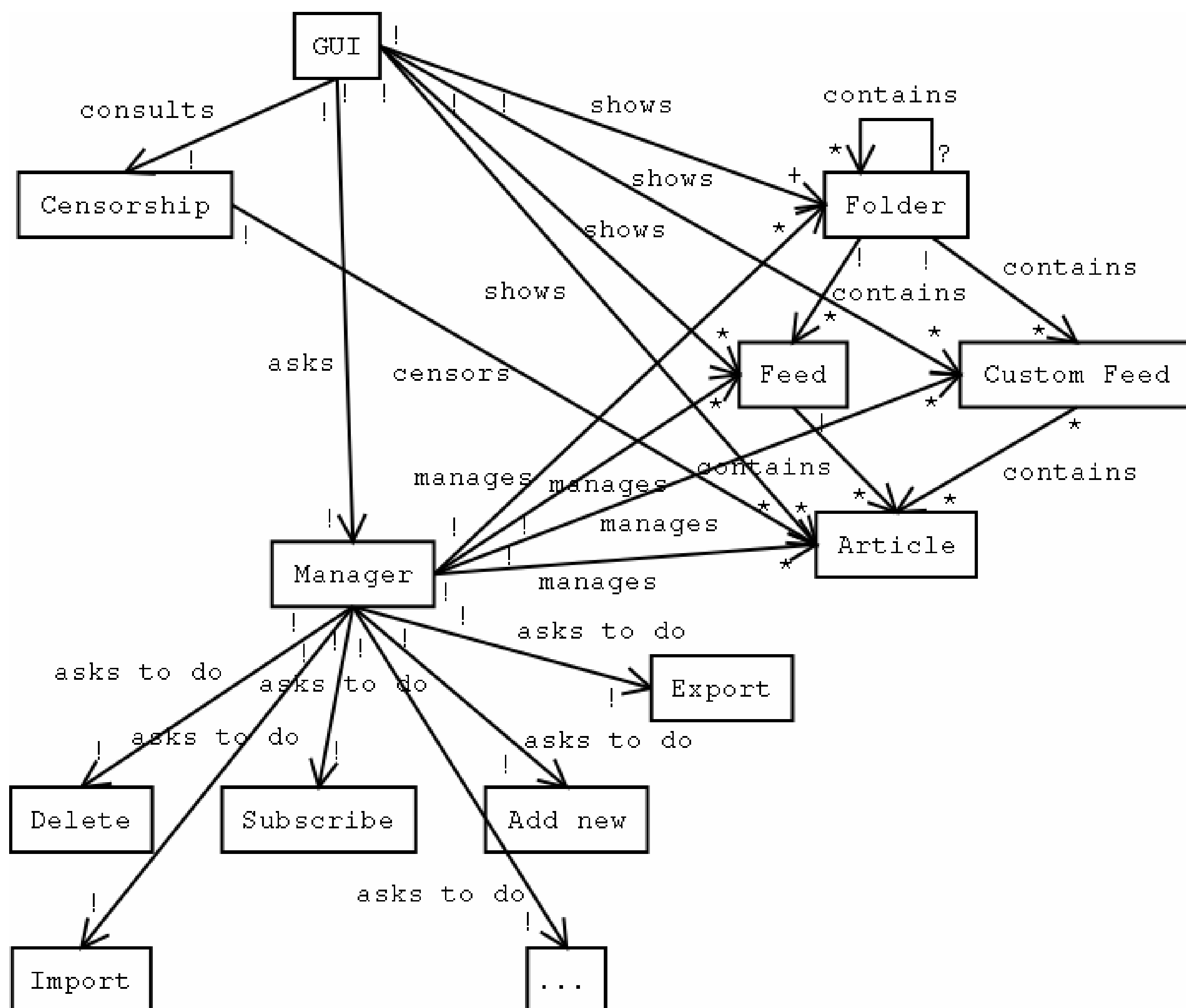
Team Member	**Primary Responsibilities**
Christopher Moh	• File Operations • Logic Layer and Threading • Design Documentation
Weijie Yuan	• Main GUI Components • User Manual • Design Documentation

<table>
<tr><td>Wonsik Kim</td><td>ADT
Message Handling
GUI Dialogs</td></tr>
<tr><td>Yongjin Kim</td><td>GUI Functionality
Icons and Art
Usability Critique and Testing</td></tr>
</table>

7.4 Module Dependency Diagram (MDD)

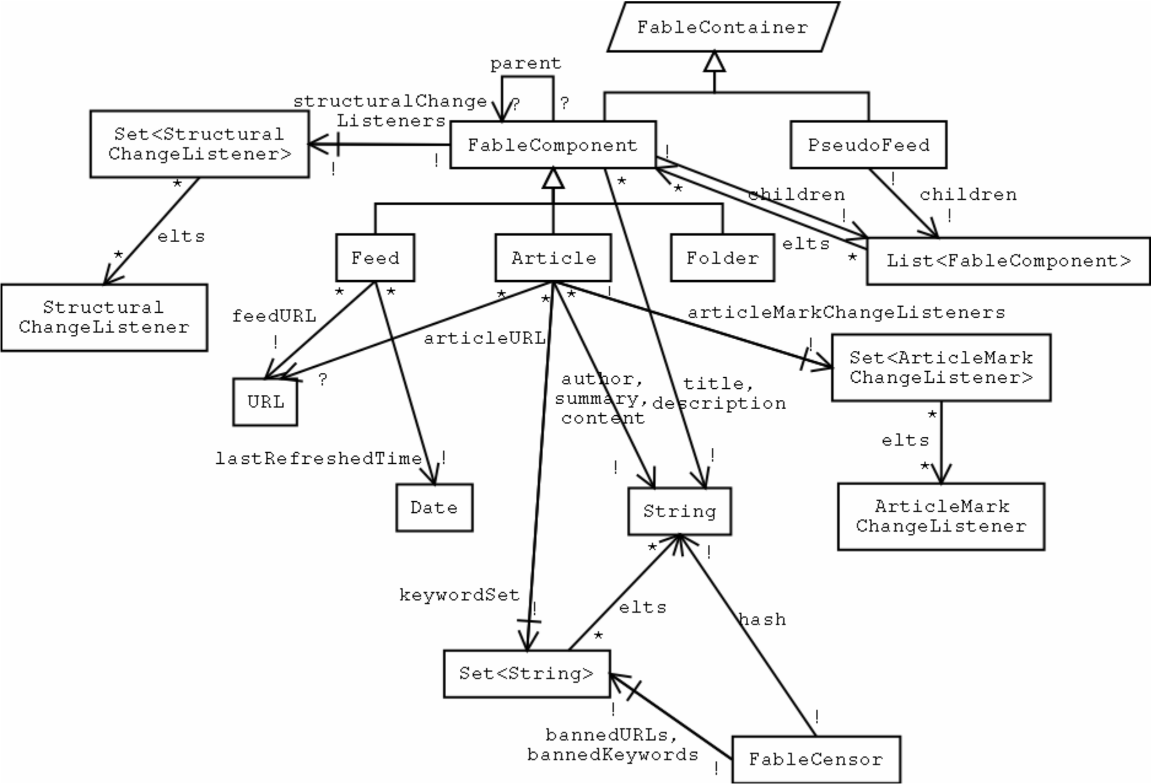


7.5 Problem Object Model (POM)

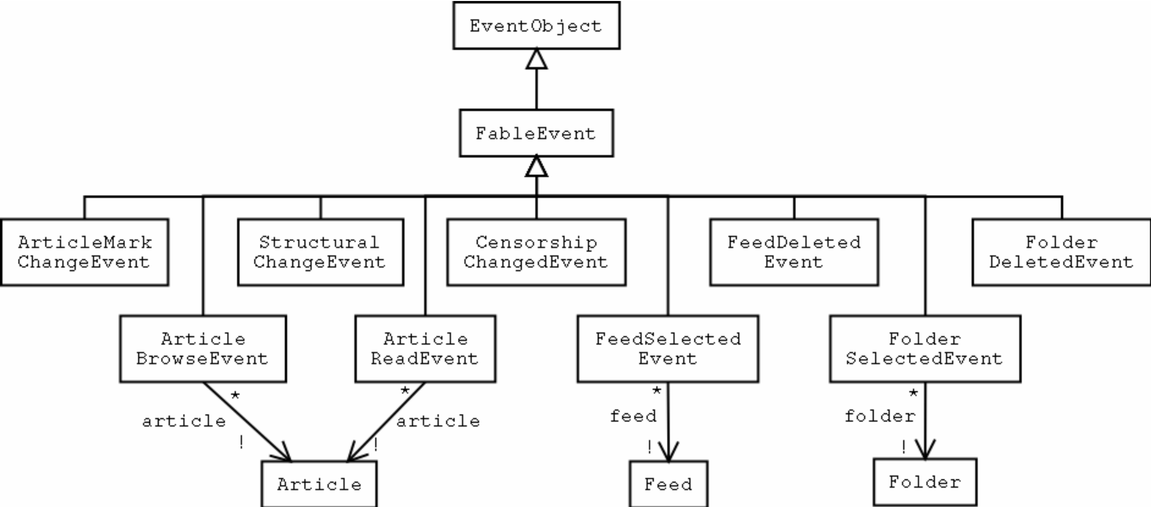


7.6 Code Object Model (COM)

Abstract Data Type (ADT)

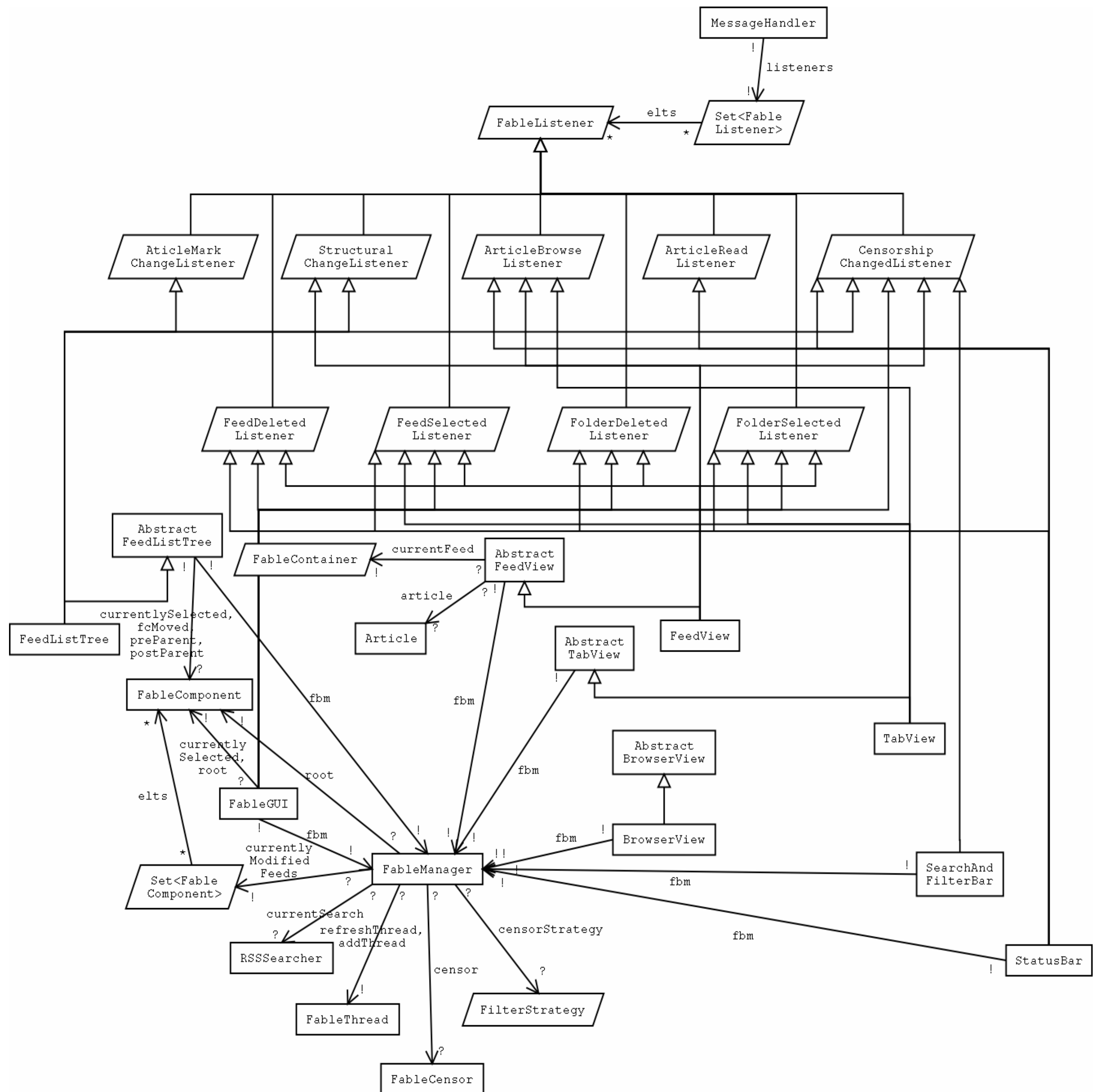


Event Hierarchy



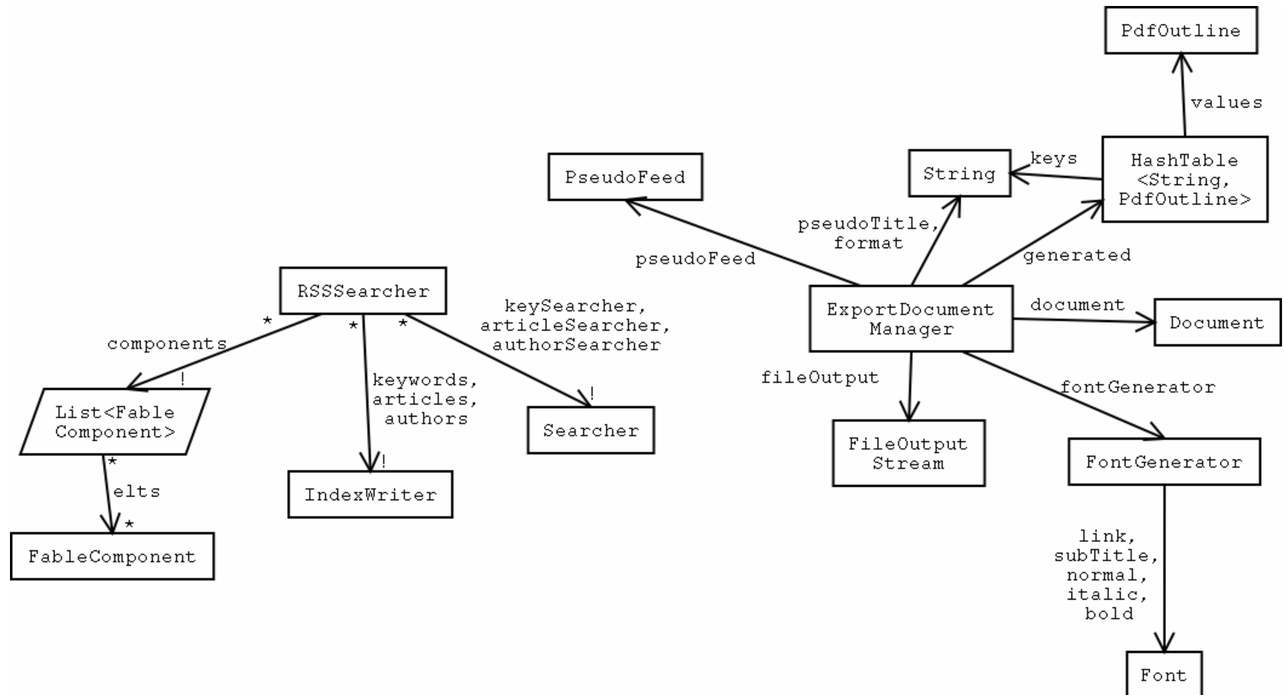
7.7 Code Object Model (COM) – continued

Graphical User Interface (GUI) and the FableManager class



7.8 Code Object Model (COM) – continued

Utility (Logic) and I/O classes



Filtering Strategies