TEAM 2: WILL CODE FOR A'S
SYSTEM DESIGN
MIRROR APPLICATION

CSC-478B Fall 2015

TITLE:	SYSTEM DESIGN
AUTHORS:	GREG PALEN, ZACK BURCH, ASHLEY ROBERTSON
DATE:	DECEMBER 14, 2015
TYPE:	FINAL
VERSION:	1.0.0
DOC ID:	DES-01
REVIEW DATE:	DECEMBER 13, 2015
ACCEPTED DATE:	DECEMBER 13, 2015

1. Mirror's Operating Environment

Mirror operates in the general context of a typical Windows 7+ single-user system. In general, there are no special requirements of this environment other than a requirement that Java 8 be installed and operational on the system. Figure 1 below depicts the overall environment with an emphasis on the operating system elements that come into play in the application (NOTE: the Windows Scheduler is depicted in anticipation of future scheduled operation of Mirror; Version 1 of the application does not include scheduled operations).

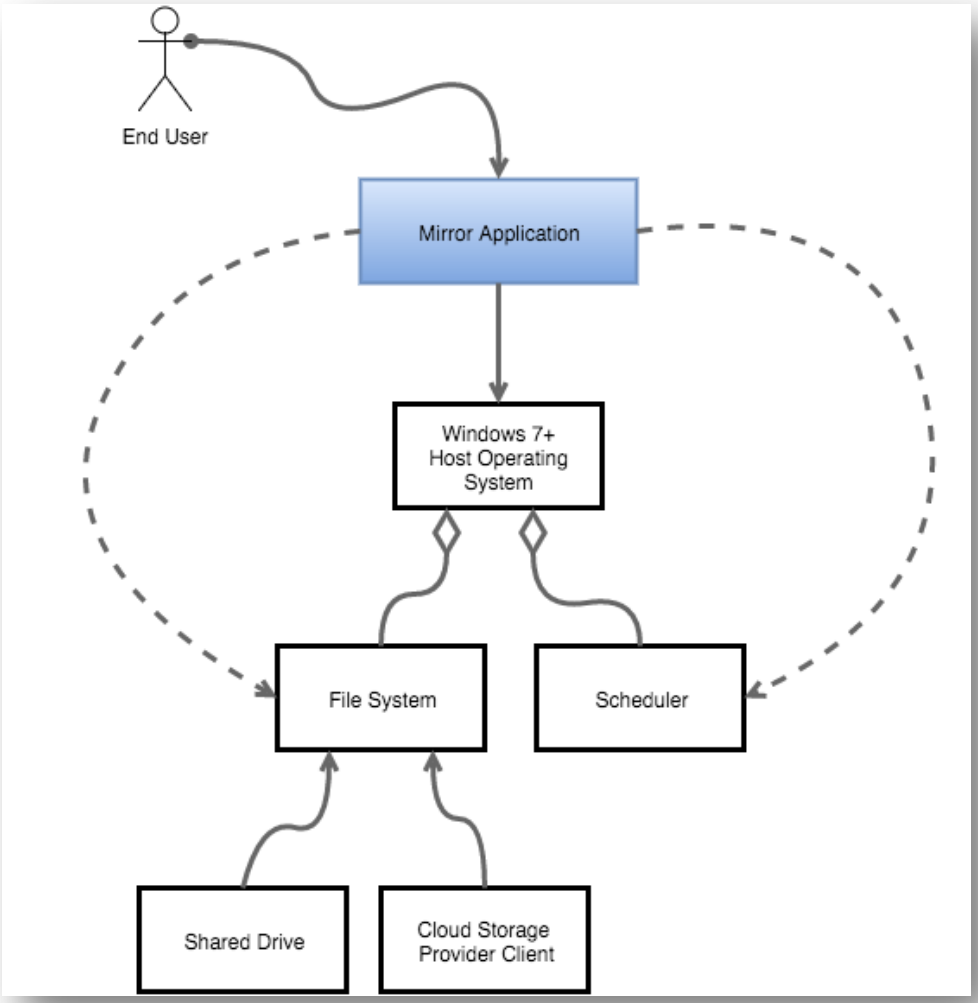

Figure 1 - Architectural Environment

2. Envisioned Use Cases

Mirror's design is a product of a fairly simplistic set of use cases envisioned for the application. While these cases are simplistic, the design anticipates new and enhanced use cases in the future and attempts to facilitate such enhancement through a combination of a layered approach to the overall design as well as a strong eye towards separation of concerns as evidenced both in the overall Model-View-Controller approach to the structure of the application and in the Pipe & Filter approach to the actual mirroring process. These design elements are discussed in expanded detail further below.

Figure 2 represents the initial set of use cases defined for Mirror. Note that not all of these uses cases are implemented in Version 1, but rather are spread out through the entire 8 version roadmap presented in the Project Plan.

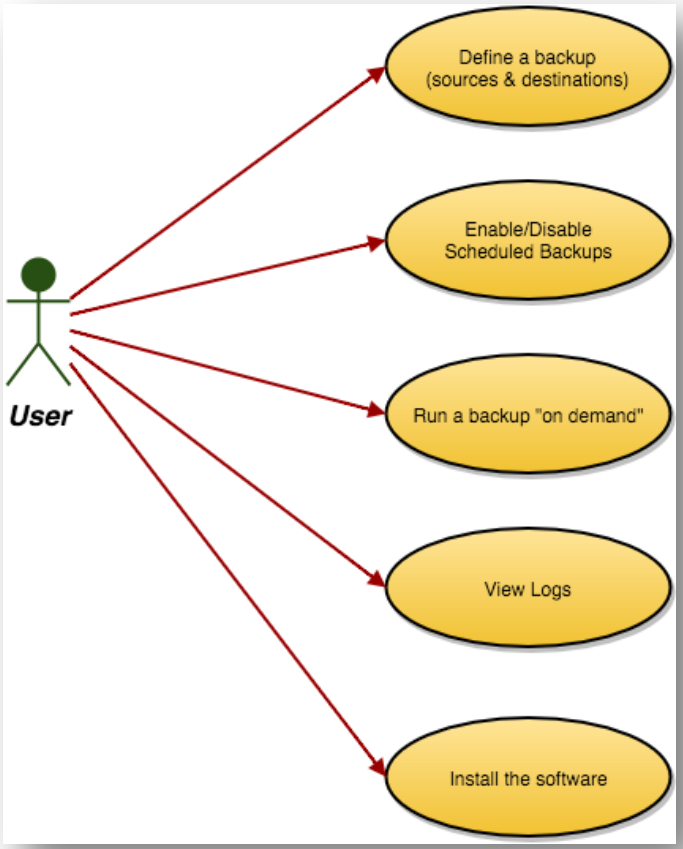

Figure 2 - Use Cases

3. Mirror's Layered Architecture

From an architectural point of view, Figure 3 depicts Mirror as a layered application (NOTE: some components are planned for future versions of Mirror). As can be seen from this diagram, the actual backup/mirroring process is a separate component from the user interface. This lends Mirror to enhancement and adaptation as will be discussed later in this document.

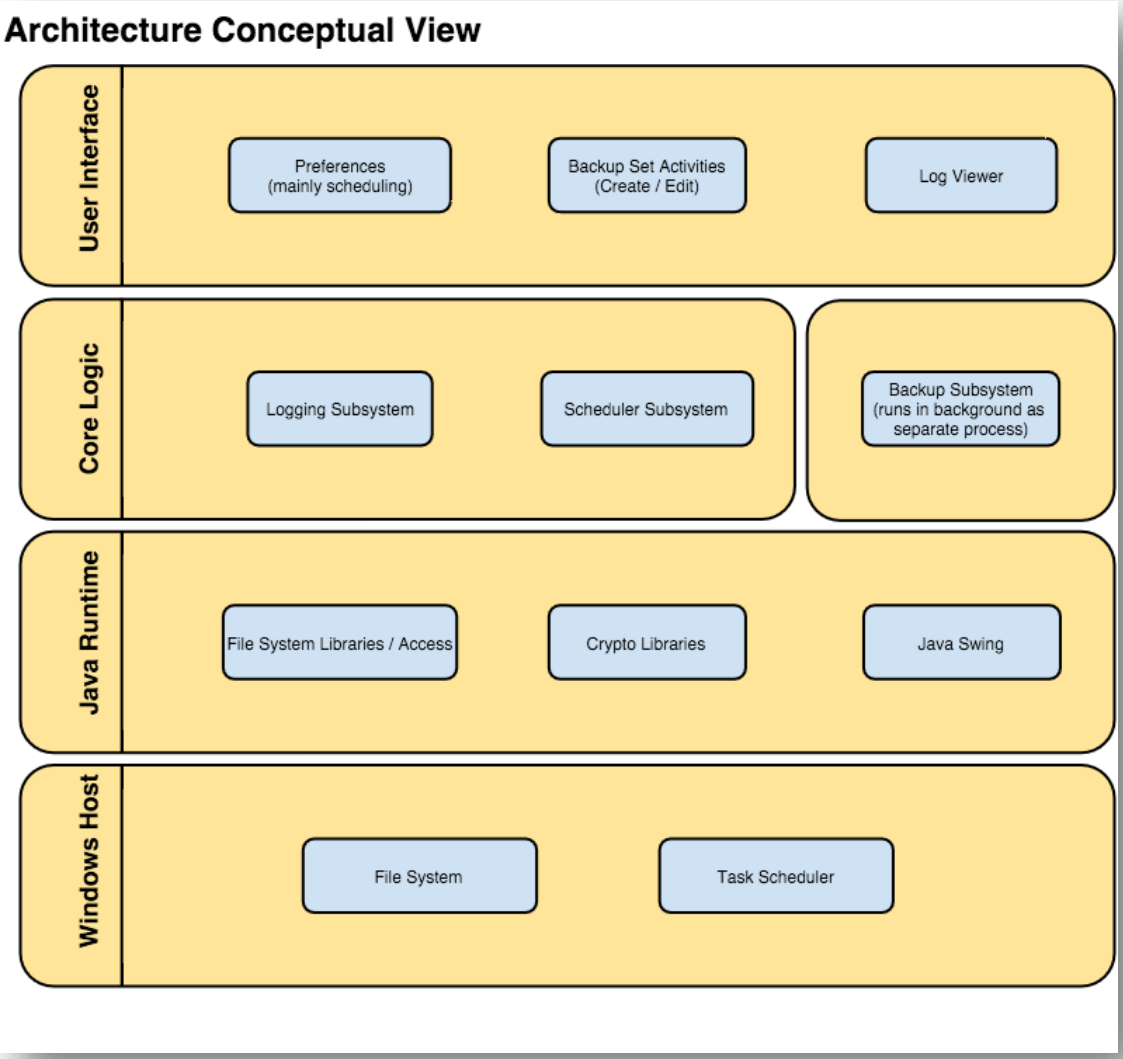

Figure 3 - Architecture Conceptual View

4. Key Classes

In an effort In order to better envision some of the key elements of Mirror's design as described at a high level in the prior section and in more detail in the sections that follow, Figure 4 below depicts the Key Classes involved in Mirror.

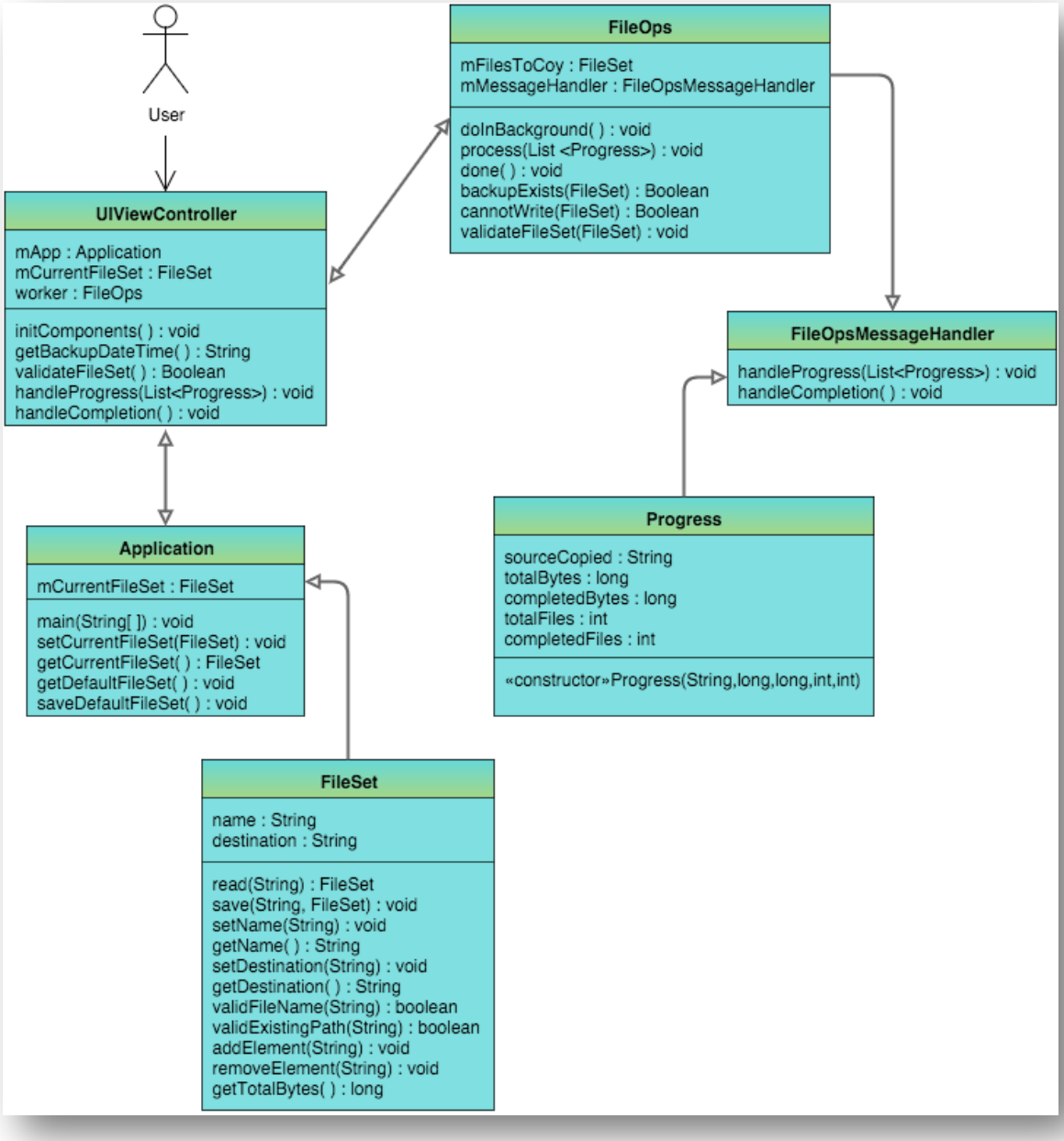

Figure 4 - Key Classes

5. Design Relative to Operation

In an effort to most easily explain the architecture of Mirror, it is helpful to examine the typical flow of operation by an end-user and relate those operations to the architecture and classes of the application. The following represents the typical flow of activities from launching the application through to the successful conclusion of a mirror operation (this list will be referenced later in the document as specific design elements are discussed and related to these operations):

1. The user launches the application.
2. The user adds files to the list of sources to include in the mirroring operation. Any files added to this list will be backed up when the user executes the mirroring process.
3. The user clicks the “Run Now” button to begin the mirroring process.
4. The mirroring process commences copying of the files defined in the source list.
5. As the operation proceeds, progress updates are displayed to the user.
6. When the mirroring process is complete, an appropriate status message is shown to the user.

From a design point of view, there are several Pattern-Oriented approaches utilized in the architecture to optimize the structure of the application.

Overall, the Model-View-Controller pattern describes the top-level design of the application. In Mirror, the “true” model class is the FileSet class as it describes the characteristics of a mirroring operation in terms of a list of source files to operate on, a destination to mirror into, and a name for the FileSet (which becomes the name of the directory that gets created within the destination to hold the mirrored files). In order to implement some default behavior for the application from the user's point of view, the Application class serves as a Proxy (or perhaps Façade) to the model (FileSet). Thus, Application is functioning as the Model from the point of view of the View and Controller in the MVC pattern (both of which are encapsulated in the UIViewController class). This approach allows Application to define default behavior such as a default destination (the user's "home" directory on the file system) and a default backup name (composed from the current date and time).

The actual mirroring operation is defined separately in the FileOps class and represents an instance of the Pipe & Filter pattern. The key design capability that Pipe & Filter offers to Mirror is the ability to easily extend the operation of Mirror by splicing the pipe and adding new filters. Figure 5, Pipe & Filter Sequence Diagram, illustrates two different ways to visualize an example of extending the core operation by adding an encryption step followed by a compression step (these steps are not implemented in Version 1, but the design facilitates introduction of such features). It is worth noting that in Version 1, Mirror consists only of the Data Source (a source file to mirror), a single pipe (implemented in the FileOps class), and the Data Sink (the destination for the mirror).

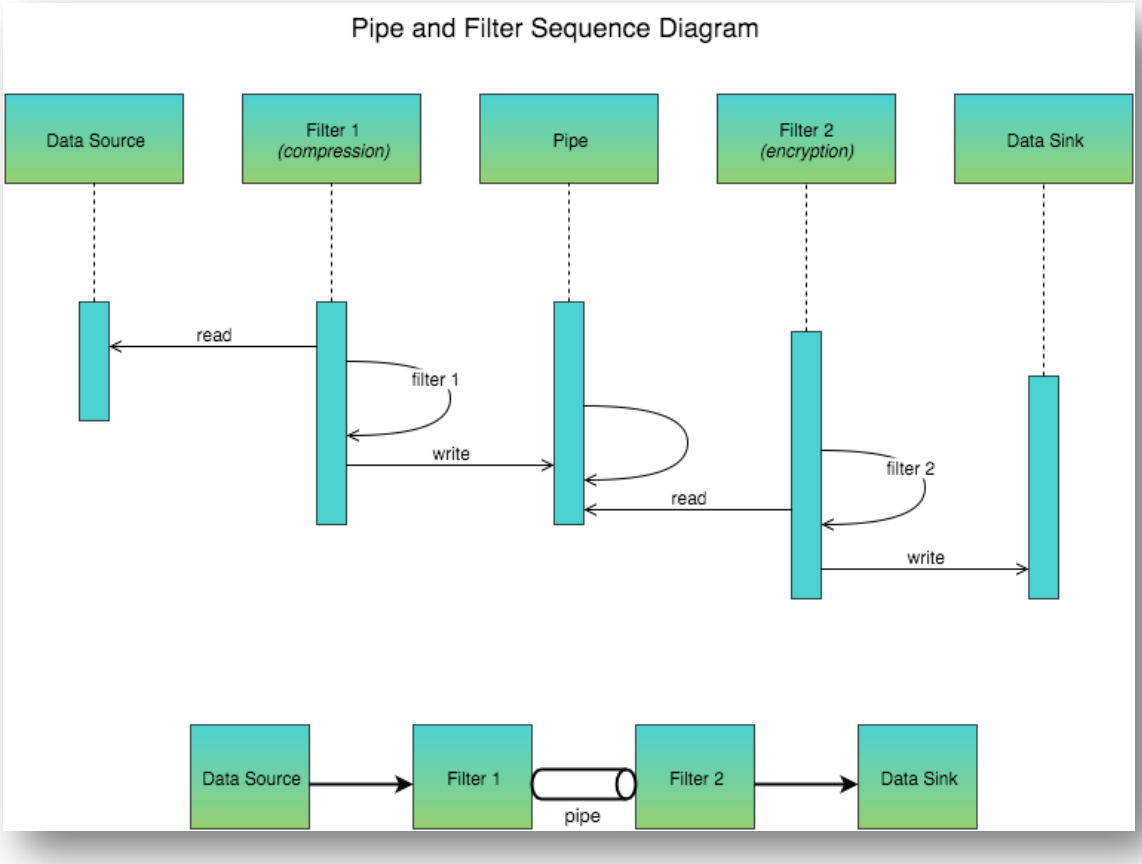

Figure 5 - Pipe & Filter Sequence Diagram

Establishing the Application's "Context" – Step 1

Upon launch, the constructor for Application class sets up the context for the application by attempting to locate a previously-saved default FileSet and loading it. If a previously-saved FileSet cannot be found, a new, empty FileSet is created. In either case, the Application serves as the interface to the FileSet for the UIViewController.

Once the constructor has finished execution, the Java Virtual Machine locates and executes the Application.main() method, whose sole purpose is to then display the user interface by instantiating a UIViewController object, passing itself (as the model in the MVC architecture) to this object to enable access to the model data and related methods.

User Interaction – Steps 2 & 3

Once the user interface is displayed, the UIViewController class handles all interaction with the end user (Step 2 of the typical interaction). Rather than strictly separating the View and the Controller into different classes, these two aspects of MVC were combined into the single class as the Controller and View were nearly indistinguishable from an operational point of view. As the user interacts with the UI, appropriate methods are called on the model via the Application class to do such things as adding new source files to the list, modifying the destination, or renaming the FileSet (the backup name).

Once the user configures the mirror operation through the user interface and is ready to start the mirroring operation itself (Step 3), the user clicks the Run Now button in the UI.

The Mirroring Operation – Steps 4 & 5

```
// Pseudocode for - The user shall have the capability of executing a backup on demand.
// (Requirement 1.1.4.1)
//
// Assumes user has clicked the "Run" button

display a progress bar and a status message to indicate the backup started in the GUI

method doInBackground()
        initialize parent destination Path object
        initialize destination directory Path object
        if (directory exists or directory is not writable)
                abort copy process
        create destination directories
        initialize array list of files to copy
        for (files left to copy - 1 < total files)
                initialize source Path object from current file
                if (source is not readable)
                        abort copy process
                add source to array list
        create first Progress object
        for (files left to copy - 1 < array list size)
                initialize source Path object from current file
                initialize destination directory Path object
                create destination directories
                initialize source File object
                initialize destination File object
                initialize input stream
                initialize output stream
                initialize array of bytes
                while (bytes left to read > 0)
                        write bytes
                        create new interim Progress object
                close input stream
                close output stream
                create last Project object with encapsulated source name
```

Figure 6 – FileOps pseudocode

The FileOps class is responsible for all of the actual mirroring functionality. This class is implemented as an extension to the SwingWorker class provided by the standard Java API to allow the mirroring operation to run in the background (in a separate thread). Not only is this required by Java (long-running operations should not occur on the main thread, which is where Java Swing user interface components operate), but it also facilitates a much more responsive user interface by allowing the UI to display progress indication. When the user clicks the Run Now button (Step 4), the FileOps.doInBackground() method is invoked where the actual Pipe & Filter pattern is implemented to copy all files defined in the source list of the FileSet to the destination identified in the FileSet.

In order to receive progress updates (identified by Step 5 above) from the FileOps class, it is necessary for the UIViewController to implement the FileOpsMessageHandler interface. The reader should recognize this design as an adapted version of the Publish-Subscribe pattern where UIViewController is subscribing to notifications of progress from the mirroring operation and FileOps is publishing these notifications periodically. The FileOpsMessageHandler interface defines a protocol for progress updates that includes periodic updates during the mirroring operation as well as a final notification of completion at the end. It is not required that UIViewController implement FileOpsMessageHandler, but it is the sole means of receiving progress information from FileOps. The Progress class is further defined as a means of specifying the status of the operation in more precise terms. One or more Progress objects are passed back to UIViewController periodically (the SwingWorker class is responsible for implementing this capability; refer to SwingWorker documentation in the Java 8 docs for more details). Each Progress object identifies overall progress (expressed both as the number of bytes copied as well as the number of files that have completed processing) as well as providing the name of the file currently being mirrored. UIViewController uses

this information to update a progress bar in the shape of a circle as well as to update a textual status in a Status box in the user interface.

Completion – Step 6

When the FileOps.doInBackground() method finishes processing, it automatically invokes the SwingWorker.done() method that it inherits to notify the calling object that the processing has finished. In this case, UIViewController (the caller) uses the completion notification to alert the user that the mirror operation has completed, thus fulfilling Step 6 of the typical flow.

6. Separation of Concerns – The Analogy of a Car

One may view an automobile as roughly two major systems if you focus only on the absolute minimum requirements: an Engine and a Body. Mirror's design approximates this analogy by separating concerns along the dividing line of processing (the engine – represented by the FileOps class) and user interface (the body – represented by the UIViewController class mainly).

Due to this design approach, the engine for Mirror can easily be re-used or adapted without needing to alter the body and likewise, the body (user interface) can easily be altered without impact to the engine. If the developer chooses to lift out Mirror's engine and wrap a new user interface around it, that's easily done. Similarly, if the developer wishes to alter the way the mirroring operation takes place, changes can be made to the FileOps class without need to modify the user interface.

The next section, Extending Mirror, elaborates on some of the modifications and reuse that is possible with Mirror.

7. Extending Mirror

Mirror was designed with extension and enhancement in mind from the beginning. There are a wide variety of adaptations that can applied to Mirror, but this section will focus on two adaptations that are anticipated to be the most common: altering the attributes of the “default” FileSet and extending the mirroring operation itself with new capabilities such as encryption and compression.

Changing Aspects of the Default FileSet

As was mentioned earlier in this document, the Application class essentially forces the user to deal with one and only one FileSet, the “default” FileSet. This behavior is implemented in the Application class' constructor which looks for a previously-saved FileSet in a very specific location and with a very specific name. One may wish to alter some of those attributes or even add the capability for multiple FileSets to be saved and loaded into the user interface. These changes would require minor modifications to the Application class to change how instantiation is handled and

potentially more extensive changes to the UIViewController class if multiple FileSets are to be handled. In neither case does the FileOps class or the FileSet class need to be extended or altered.

Leveraging Pipe & Filter to Extend Mirror's Copying Functionality

The marquee design element of Mirror is the choice of a Pipe & Filter design pattern for the mirroring operation itself. While Version 1 of Mirror does not take advantage of this pattern (specifically, it does not apply any intermediate filters to transform the stream of bytes from source to destination), the FileOps class lends itself easily to the introduction of filters that would be linked together to apply desired transformations.

Two examples of potential enhancements that are actually anticipated in the roadmap for Mirror are the ability to encrypt files as they are mirrored as well as the ability to compress them. These capabilities an easily be inserted into the FileOps.doInBackground() method to pass the source data through an unlimited number of filters, then writing the transformed bytes to the output. Notably, the Pipe & Filter design also facilitates the combining of filters such that the option could be created to both encrypt and compress the files.

Another key capability of the Pipe aspect of Pipe & Filter is the ability to fork the pipe. This feature would allow a developer to have multiple operations occur on the source byte stream simultaneously such as allowing for multiple destinations (in other words, multiple copies) or even allowing for an encrypted copy as well as an unencrypted copy. It is up to the resourceful developer to explore the breadth of capabilities this design pattern lends to the application.