FBMS

Milestone 4: Requirements and preliminary design

- Mike Hoffert - mlh374
- Syed Ahsan Rizvi - sar457
- Hattan Alsharif - haa775
- Da Tao - dat293
- Michael Butler - mdb815

1 - Summary use cases

Change backup folder (Hoffert)

Level

Summary

Actors

User

Goal

To allow the user to select a new location to store their backed up files.

Activities

The user chooses an option from a menu in the front end GUI. This opens a prompt to select a new directory location for the backup directory. Choosing a new location asks for confirmation. If the user confirms the action, the file that specifies the location of the backup directory is changed. The contents of the previous backup folder are then copied to the new location. The user is then prompted if they wish to delete the old backup location.

Quality

This aspect of the program is of very low priority, due to the fact that the program does not require this functionality to work. However, due to the fact that bugs here could cause a loss of all backed up data, it is a requirement that this functionality be stable. Therefore, it will likely be put off until last, and implemented only if time allows.

Version

11 October 2013

First run (Hoffert)

Level

Summary

Actors

User

Goal

To setup the live and backup directories on the first time the program is run

Activities

The program determines if it is the first run by checking for the existence of the file which specifies the backup directory. If this file is not found or the backup directory is invalid, it is considered to be the first run. A GUI wizard is shown, and the user is asked if they wish to specify a new backup or if they wish to import an old backup. To create a new backup, the user is prompted for paths for a live and backup directory (which cannot be children of one or the other). To import an old backup, the user is prompted to provide a path to the backup folder (which contains a database file). If the user enters invalid paths, they will be prompted to enter a new one. The wizard also allows the user to go back and change previously

made choices. On the final screen of the wizard, the program initializes the backup and live directories and creates the database connection.

Quality

This aspect of the program is top priority, as it is mandatory for the program to be used at all. Therefore, it must be completely stable and well designed. In fact, this portion of the program is already complete in our prototype, as will be demonstrated in section 9.

Version

11 October 2013

Create file (Tao)

Level

Summary

Actors

System

Goal

Add the file created by user to the system and backup it.

Activities

The system receives a file created notice from the watcher module.

A file with the same name and the same location has been existed in the system: Since there has been the "same" file existing, the system will first add a record, stating this special situation. The system will keep the previous backups, but do not make a diff file in this case. The system will copy the newly created file into the backup folder.

No file with the same name and the same location has been existed in the system: The system just simply copies the file into the backup folder. Then a new record will be added in database stating a new file added.

Quality

This is a main use case in the system and must work to a very high standard.

Version

13 October 2013

Rename file (Tao)

Level

Summary

Actors

System

Goal

Reflect that user renamed a file.

Activities

The system receives a file renamed notice from the watcher module. First the system will add a record to the database of this; then the system will rename all backups and records of this file.

Quality

This is a main use case in the system and must work to a very high standard.

Version

13 October 2013

Rename folder (Alsharif)

Level

Summary

Actors

User

Goal

To rename a backup folder to a new name.

Activities

The user specifies whatever folder he/she wants to rename it and then give it a new name. The system checks if the new name is already exist, then the user will be notified to choose another one, otherwise the folder name will be changed.

Quality

This is a main use case in the system and must work to a very high standard.

Version

14 October 2013

View revision (Alsharif)

Level

Summary

Actors

User

Goal

To show or display a selected revision of a file.

Activities

The user select a whatever file he/she wants to view a specific revision for that file. With a selected file, the user is presented a list of revisions and options, which include view revision.

Quality

This is a main use case in the system and must work to a very high standard.

Version

14 October 2013

Delete file (Rizvi)

Level

summary

Actor

System

Goal

Allows the user to delete a specified file

Activities

User scrolls over live directory and finds the file to be deleted. JNotify notices the file change and calls the watcher. The watcher notices a file has been deleted from the live directory and add to the list. The control reads the list and removes any other instances of the file in the other list.

Quality

This is a main use case in the system and must work to a very high standard.

Version

17 October 2013

Create folder (Rizvi)

Level

Summary

Actor

System

Goal

Allows the user to create a file

Activities

When a file is created the watcher notices the change. Control checks if the folder is present. If the folder is already present in the list there is no change.

Quality

This is a main use case in the system and must work to a very high standard.

Version

17 October 2013

Revert file to revision (Butler)

Level

Summary

Actor

User

Goal

Ability to revert a file to a prior version.

Activities

The file selects a revision using the GUI. Backend then uses FileHistory? and FileOp? to revert the file in such a way as not to interfere with versioning.

Quality

This part of the program is high priority, it is not critical to the backup functionality of the program but is necessary for the user to take advantage of the versioning information being created. To that end it is also a primary component to why a user would use our program.

Version

October 20th, 2013

Change live directory (Butler)

Level

Summary

Actor

User

Goal

Gives the program the ability to change where the current live data is being held. Changing the saved setting for the location and also moving all necessary content to the new location.

Activities

User specifies a new live folder location using the GUI. The GUI/frontend then calls control, which handles the rename, makes any necessary changes internally and moves necessary components to the new folder. Then cleans up any backup or temporary files.

Quality

Low priority part of the program. The system functions as specified without this being implemented. It however is a convenient for the user, and requires testing to avoid data loss.

Version

October 20th, 2013

2 - Fully dressed use cases

First run (Hoffert)

Scope

- Program is run for the first time

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Setup the program with the directories it will act on.

Preconditions

- It is the program's first run or the program cannot startup normally.

Success Guarantee

- The backup and live directories are configured to valid paths.

Main Success scenario

- Program determines that it is the first run because the file pointing to the backup location is missing, or the backup location is otherwise invalid (non-existent folder or missing necessary database file inside this folder).
- A GUI wizard is displayed, with buttons for navigating and exiting the wizard.
- The wizard prompts the user as to whether they wish to create a new backup or import an existing backup.
    - If a new backup is chosen, the user is prompted to provide paths for both the live and backup directories (via file choosers).
    - If the user chooses to import an existing backup, they are prompted to provide a path to this backup directory.
- The wizard finalizes, telling the user that setup was completed.
- The program establishes a connection with the database, setting it up in the specified backup directory by creating the file that points to the backup directory.

Extensions

- The user cannot specify live and backup directories that are children or parents of each other.
    - This prevents infinite recursion. If the backup folder was inside the live directory, every backup would be identified as a change to the live directory (and thus need to be backed up as well). If the live directory was inside the backup directory, then we run the risk of backed up files overwriting files in the live directory.
    - The program will issue an error dialog if this occurs, and will not allow the user to continue until they specify valid directories.
- When importing an existing backup, the chosen directory must contain a database file which indicates that the directory has been used for backup before.
    - This file contains the path of the live directory, which is necessary for the program to import an existing backup.
    - The program will issue an error dialog if this occurs, and will not allow the user to continue until they specify a valid directory.

- If the user closes the dialog window, it will do nothing unless they are on the final panel of the wizard, which indicates a success.

Special Requirements

- The program directory and the chosen live and backup directories *must* have write access.

Technology and Data Variations List

- The program flow changes based on whether the user chose to import an existing backup or start a new one. If they chose to import an existing backup, the live directory is set when the program initializes the database. If they started a new backup, the wizard sets the live directory.

Frequency of Occurrence

- Exactly once when the program starts for the first time
- May also occur if an error is encountered when the program starts up (such as if the backup directory is invalid)

Delete file **(Rizvi)**

Scope

- FBMS

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Wants to delete a file present in the live directory.

Preconditions

- None

Success Guarantee

- Nothing has changed

Main Success scenario

- The user deletes a file outside the system.
- JNotify notices the file change and calls the watcher.
- The watcher notices a file has been deleted from the live directory and add to the list.
- The control reads the list and removes any other instances of the file in the other lists.

Extensions

- None

Special Requirements

- None

Technology and Data Variations List

- None

Frequency of Occurrence

- Every time a file is deleted from the live directory.

Create file **(Tao)**

Scope

- FBMS

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Want this new created file is automatically backed up.

Preconditions

- The Watcher module is running normally.
- There is enough space in backup folder.
- System has read access to the newly created file.
- System has write access to backup folder.

Success Guarantee

- The newly created file is backed up.
- Records are added in database.
- The history of this file is accessible from GUI.

Main Success scenario

- Program determines whether a file with the same name and location has been existed in the system.
    - If it exists, system adds a special record indicating a file with the same name and location is created. Then the system renames the previous file backup, to prevent overwriting.
    - If not, system goes to next step straightly.
- System copies the newly created file into backup folder.
- System establishes a connection with the database, inserting a new record into the database.

Extensions

- The file will be locked when being copied to backup folder.

Special Requirements

- Privilege is possible required.

Technology and Data Variations List

- The program flow changes based on whether a file with the same name and location has been existed in the system. If it exists, system will do additional steps to ensure no overwriting occurs. If not, system will do it in regular way.

Frequency of Occurrence

- Every time a new file is created in live folder

Rename file **(Tao)**

Scope

- FBMS

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Want this renamed file is automatically renamed in system.

Preconditions

- The Watcher module is running normally.
- The reversion is accessible.
- System has read access to the newly created file.
- System has write access to backup folder.

Success Guarantee

- The backups of file renamed are renamed.
- Records are renamed in database.

Main Success scenario

- System is notified for the renaming operation.
- System renames all the backups of the file.
- System add a rename record into the database.
- System changes all the records of the file.

Extensions

- The backups of file will be locked when being renamed.

Special Requirements

- None.

Technology and Data Variations List

- None.

Frequency of Occurrence

- Every time a file is renamed in live folder

Rename Folder **(Alsharif)**

Scope

- FBMS

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Want this folder to be renamed.

Preconditions

- The Watcher module is running normally. There is no similar folder name.

Success Guarantee

- The folder has renamed to a new name. The new folder history is added in database.

Main Success scenario

- The user wants to rename a specific folder.
- Program determines whether the folder's name has been changed in the system or the new name exists.
- If it is exist, ask the user for another name
- If not, system goes to next step straightly.

Extensions

- The folder will be renamed, and it's files will remain with no changes.

Special Requirements

- No same folder's name is exist.

Technology and Data Variations List

- The system determines whether the folder with the new name and location has been existed in the system or not to ensure no overwriting occurs.

Frequency of Occurrence

- Each time a folder's name is changed

Revert file to revision **(Butler)**

Scope

- FBMS

Level

- User goal

Primary Actor

- User

Stakeholders and interests

- User: Revert the selected file to the selected version.

Preconditions

- System must be in a properly setup and running state.
- A file must be selected by the user.
- The file must be within the folder listing that is currently being backed up.
- A version must be selected by the user.
- A version history must exist for the file.
- System has proper read/write access to the backup and the system file.

Success Guarantee

- The selected file is reverted to the proper revision.
- The previous version that was overwritten still remains in backup as a version.

Main Success scenario

- User prompts the program via the GUI that they wish to make a file revision.
- The user selects the file in question and then selects the revision they desire from the selection.
- The system then goes through file history to find the proper diff file referenced by the database.
- Temporary files are made where appropriate.
- The diff file is used to create a reverted version of the specified file.
- The file specified is now reverted and in the proper location.

Extensions

- Only files within the specified backup environment will have versions stored to revert to.
- Nothing will happen if the user selects the file and revision and exits the dialogue or otherwise does not hit ‘Ok’.
- If the function fails due to system level issues, like permission conflicts, the function ends and the user is notified.

Special Requirements

- Require write/read access to the file being modified.

Technology and Data Variations List

- None

Frequency of Occurrence

- Whenever specified by the user.

3 - Supplementary specification

Interfaces (Rizvi)

The interface consists of the GUI windowing system along with a minimizer tray icon. Closing the GUI minimizes the application to an icon in the system tray. This means that the application does not end simply by pressing the “x” button on the GUI but needs to be shut down from the system tray icon which has an “exit” option for closing the application entirely. We have tried to make the main GUI as simple as possible so that an average user could be able to handle the software. It comprises of a few simple areas namely a table, a toolbar, and a menubar.

Table: This is used to display the contents of the backup directory. This directory is specified by the user and shows detailed entries of the names of the files or the folder, the sizes, the modified and the created dates as well as the last date it was accessed on and finally the revision size and the number of current revisions. Interactive features of the table include a scrollbar, which the user can use to scroll up and down the list and selection, allowing the user to select a single file at a time.

When a folder is selected the table refreshes the contents of the list by generating the contents of the folder. Selecting a file opens up a dialog box with information of the file revisions. Errors are handled by the means of dialog boxes.

Toolbar: This is located above the table and contains two buttons and a text field. One of the button is an up button enabling the user to navigate out of the current folder. However the button is disabled if the user is already in the backup directory. The second button is the refresh button which refreshes the directory to the latest version. The text field is set to contain the current path of the directory and is also disabled if the user is already in the back up directory.

Menu bar: At the very top is a menu bar which consists of a file and a help item. The file item expands to options of:

- View revisions: It works in the same was a double clicking a file (eg, opens the revision dialog box; if no files are selected, do nothing).
- Copy-to: Creates a dialog for selecting the folder to copy the selected file to. Like review revisions, this cannot be done if no file is selected.
- Restore all: Prompts the user (with a dialog box) if they're sure they want to continue. Upon approval, the directory is restored to the previous revision.
- Exit: Closes the GUI window. However it can be brought back by clicking on the icon in the system tray.

The help menu has a single option to display the online help. This opens the website for the software in the users default browser.

Vision (Butler)

FBMS is a multiplatform, self-contained automated backup and revision program. It does not require cloud services or any online access to function. It uses local resources and ideally secondary drive locations to backup critical data.

We chose this project as it fills a gap between cloud backup platforms and weighty software suites requiring considerable infrastructure and resources. Our program takes advantage of external drives or

mapped network drives and consolidates folders specified by the user to a secure location. Joined with a redundant hardware solution like RAID, our software is a suitable replacement to cloud services.

Our design choices have lead to a cross platform utility that should be usable on any operating system supported by the Java run time. This frees it from the majority of dependencies that weigh down other software.

Using the data we backup we are able to run and operate a versioning system. This allows a user to revert unwanted file changes, restore corrupted files, or replace files accidentally deleted. It takes a step further then previous versions in windows, or other similar systems like svn. This gives more control over where versions are kept and is more straightforward then other versioning systems.

Third party components **(Hoffert)**

The project uses third party components for several areas, allowing the project to be completed in a reasonable span of time, as well as assisting in functionality beyond our current skills. Currently, we use:

- **JNotify**: This library is used to detect changes in the live directory. It is capable of running event handlers when file changes occur. This allows us to efficiently find the changes in the directory. JNotify binds with the operating system (in which it supports all the major operating systems), allowing it to function similar to an interrupt-style listener. The program will then use a polling system to perform file operations on changes that occurred since the last poll. This makes spotting changes fast while minimizing file operations (which often occur in multiples even if the operation appeared atomic to the user).
- **Log4j**: The well known logging library is used to making logging efficient and configurable. Most importantly, it allows us to set levels to our logging. When the logging is set to DEBUG, the logging system is very verbose. This is useful when we need to know exactly what the program is doing, but for general usage, this slows the program down and floods the logs with irrelevant information. Normally, the user will have the logging set to either WARN or ERROR, which will limit the logging to things that are of actual use to the user.
- **Java-diff-utils**: This library is used to generate diff files, which is what our revisioning system is based on. The library is also used to apply the files, restoring the originals. To restore files several revisions old, the diff files are chained together, restoring revisions until we get to the one we want.
- **SQLite**: We use the SQLite JDBC driver to allow our databases to be well organized into an efficient, minimal file.

These components integrate with our program and were chosen with efficiency in mind.

Supportability **(Tao)**

OS Compatibility

- Different OS organizes files in different ways. For Unix-like OS (Mac OS X and Linux), files and folders are in a single tree root; for Windows, first there are many roots(drives), then the roots have their own trees.
- Since that, it is necessary to declare the OS type in configuration.
- Besides, the separator differs. Unix use "/", while Windows use "\". Fortunately, Java provides a method to retrieve the separator used. Windows also works with the forward slash.

Network Drives

- Supported provided the location of the drive is consistent (eg, same mount location or drive letter).
- No reliable means of creating links when the location of a drive is inconsistent.

Configuration

- The configuration will be stored as plain text, in the folder of program.
- The locations of live and backup folder could be changed.
    - A change in live folder will cause a factory reset.
- Importing is easy. Just select the previous backup folder, and the system will take care of the rest.
    - A backup of database will be stored in backup folder for a backup.

Reliability **(AIsharif)**

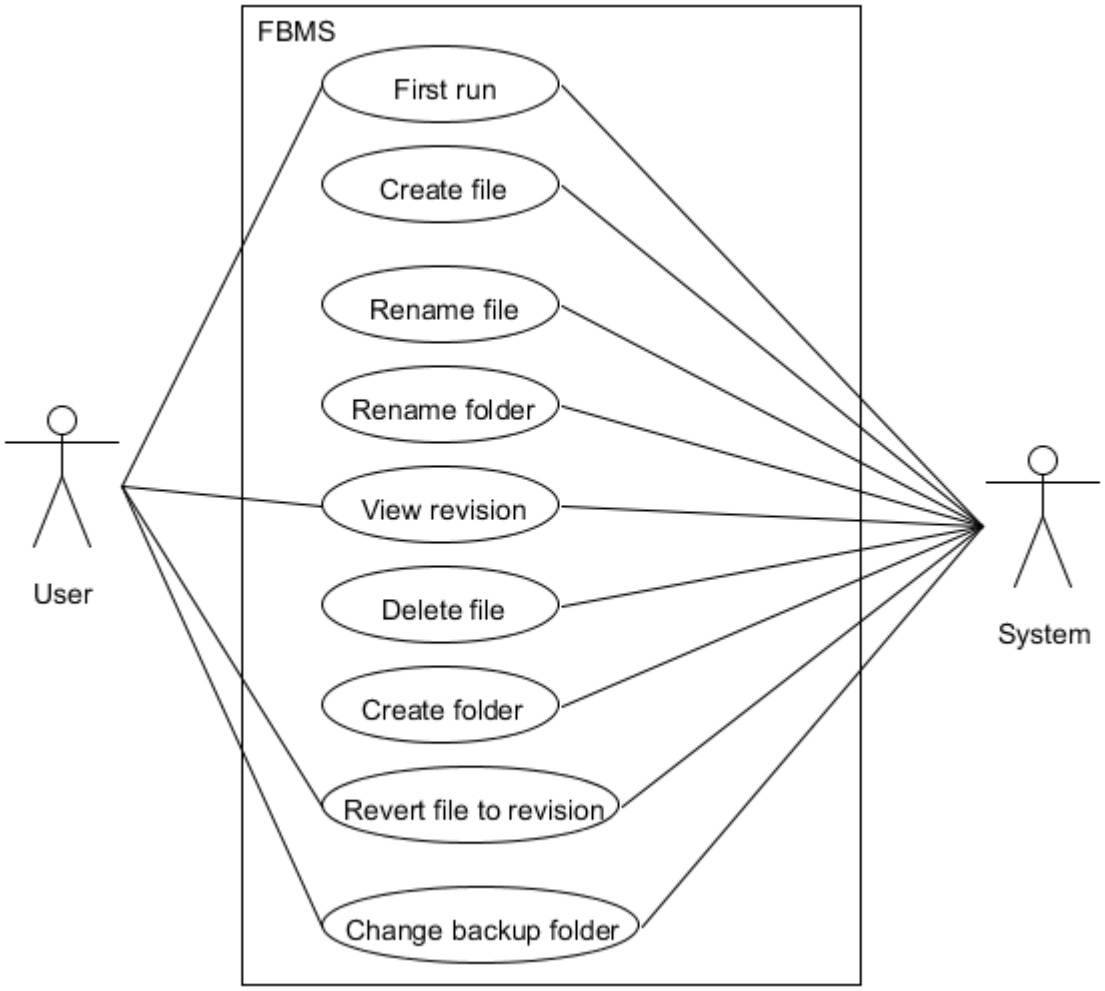
To maximize efficiency, the watcher module, which notices changes to files, adds these changes to lists. At specific intervals, the program handles the files in these lists. This reduces the program's use of system resources by allowing the program to frequently sleep instead of constantly looping while the program is running. It also drastically reduces the number of file operations. Some operating systems and programs will actually modify a file multiple times in just a few milliseconds (even if the file only appears to have been modified once from the user's perspective). The interval based system combines duplicate operations. So if the user modifies a file a dozen times within the interval, the program only stores one modification as a revision (with that revision being all the changes since the file was last modified).

Similarly, we use the JNotify library specifically because it binds with the operating systems for maximum performance. At one early point of time, we actually considered checking *all* the files in the live directory for changes in their CRC value, which would have been much slower.

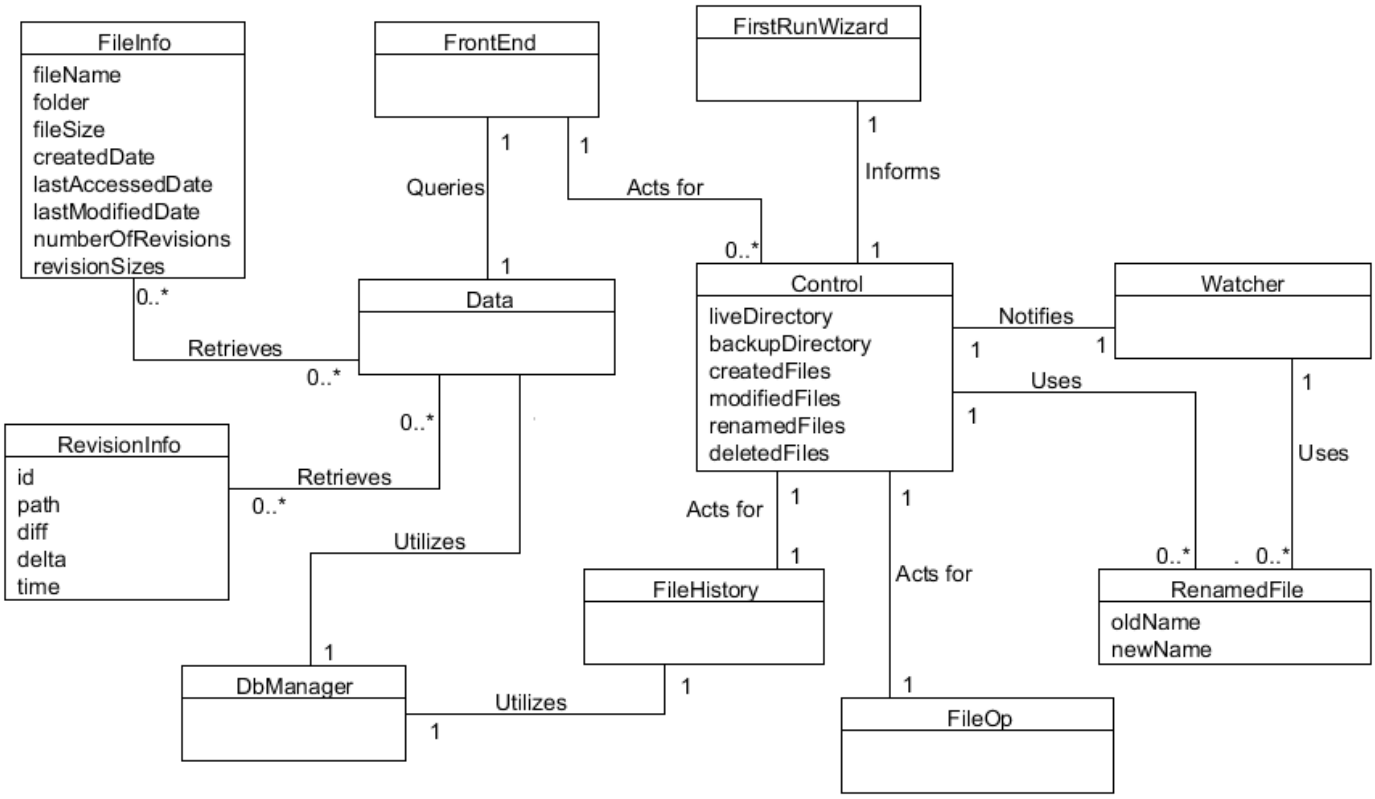
The program also handles errors in a user-friendly way. Fatal errors are displayed as dialog boxes and also copied to the log (which helps find the source of errors). Non-fatal errors use a non-obtrusive error pop-up message that appears in the bottom corner of the screen and disappears on its own after a few seconds.

Finally, when file operations cannot be performed on a file (for reasons such as the file being open in another program or invalid permissions), the operation is retried after a short delay before informing the user of the error (which is a non-fatal error, as the file can simply be skipped).

4 - Use Case Diagram



5 - Domain Model

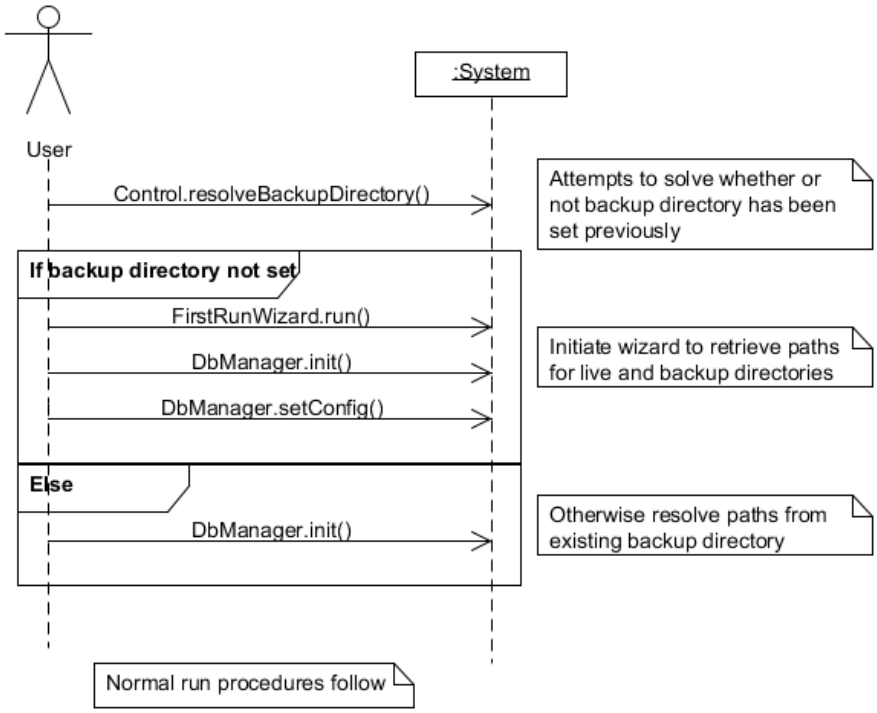


6 - Glossary

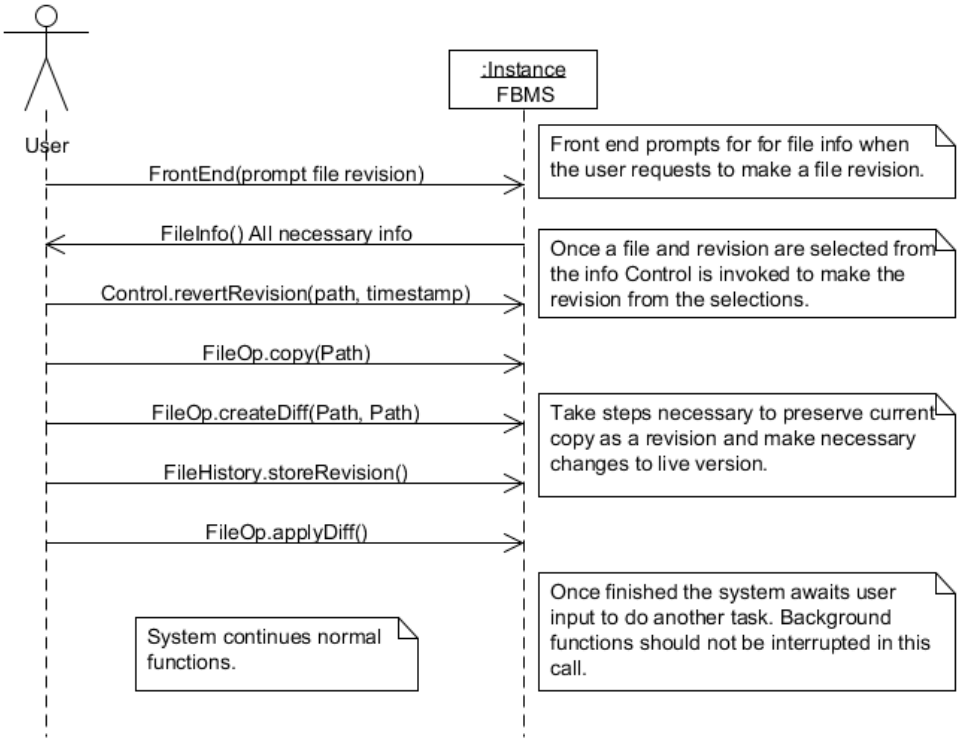
- **Live directory**: The directory that the program monitors for changes. In other words, the directory that is being backed up.
- **Backup directory**: The directory where the backed up files and revisions from the live directory are placed.
- **Revision**: A snapshot of a file at a given time.
- **Diff**: Data that details what changed from one revision of a file to another. Also called a *patch*.
- **Revision database**: The database where revisions are stored as diff files.
- **Front end**: The visible portions of the program, including the main GUI window, the toolbar icon, and dialogs.
- **Watcher**: The program component that watches the live directory for changes. It operates concurrently from the rest of the program.
- **Control**: The heart of the program that performs setup and acts on the files that the watcher flags as "changed".
- **File operations**: Operations that change a file's content, name, or location. This includes operations such as file creation, renaming, moving, or deletion.
- **Error alerts**: GUI dialogs that alert the user to an error. They are split into two branches:
    - **Fatal alerts**: Obtrusive dialogs that indicate that the program cannot proceed.
    - **Non-fatal alerts**: Non-obtrusive dialogs that appear in the bottom right corner of the screen and automatically disappear after some time. These indicate something has gone wrong and cannot be recovered, but do not halt the progress of the program. For example, a file that cannot be backed up (perhaps due to permission issues).

7 - System Sequence Diagram

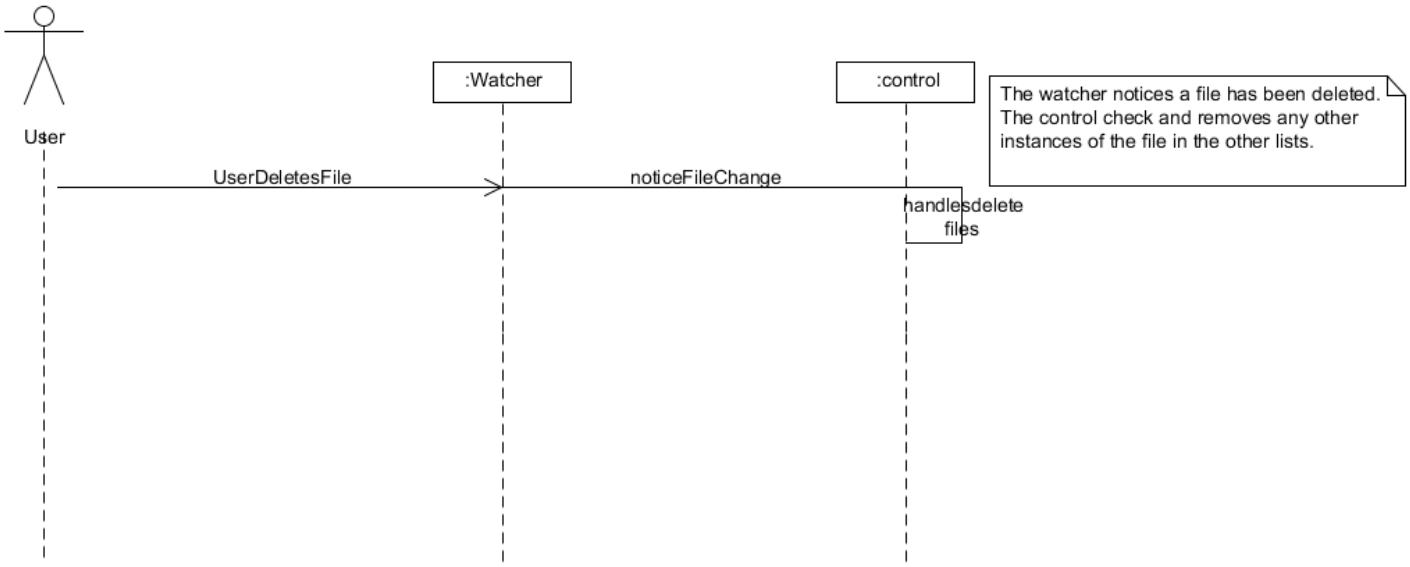
First run (Hoffert)



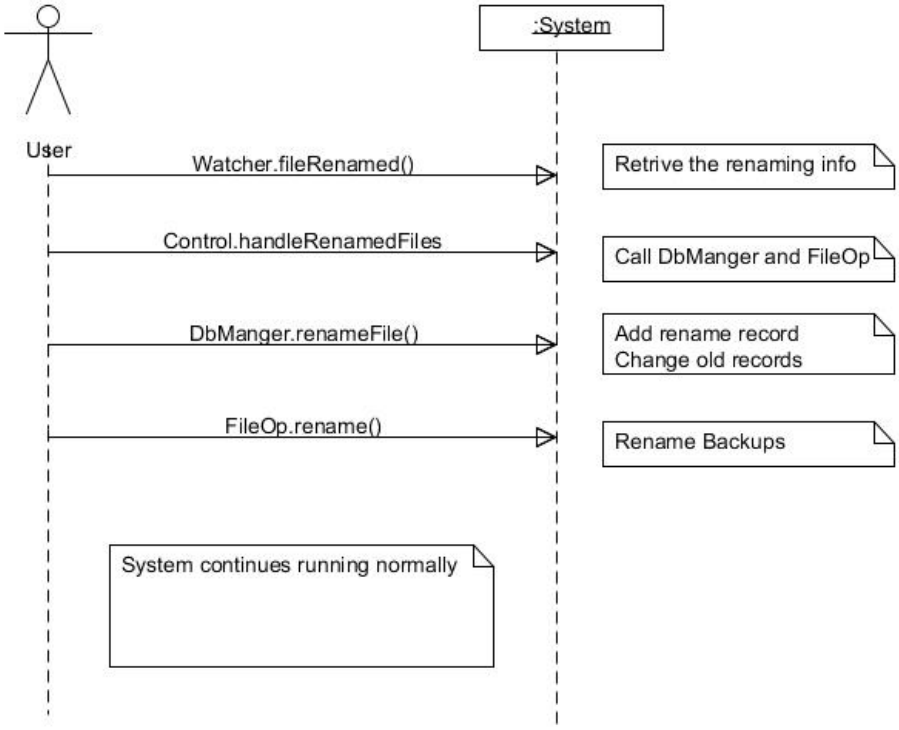
Revert to file revision (Butler)



Delete file (Rizvi)



Rename file (Tao)



8 - Operation Contracts

Change backup folder (Hoffert)

Cross references

- Copy file operation.
- Delete file operation.
- Set configuration database operation.

Pre-conditions

- None

Post-conditions

- The backup directory is changed, both in the program and in the file that stores the backup directory's location.
- If the user agreed to it, the contents of the previous backup directory were copied to the new directory.
- If the user agreed to it, the old backup directory is deleted.

First run (Hoffert)

Cross references

- Initialize database operation.
- Set configuration database operation.

Pre-conditions

- None

Post-conditions

- The live and backup directory locations are set.
- The database file is created if this is a new backup.

Create File (Tao)

Cross references

- Watcher
- Insert new record into database.
- Rename file if necessary.

Pre-conditions

- Backup folder exists and has enough space.

Post-conditions

- The newly created file is backed.
    - Previous backup will be renamed if the backup has the same name and locations as the newly created one.

Rename File

Cross references (Tao)

- Watcher

- Insert a rename record into database.
- Change file name in previous records.
- Change previous backups' name.

Pre-conditions

- A file is renamed.
- Backup folder exists.
- Backups of the file renamed exist.
- Records of the file exists in database.

Post-conditions

- A file rename record is inserted into database.
- All file names of the records of the file renamed are changed.
- All backups' name of the records of the file renamed are changed.

Rename folder (Alsharif)

Cross references

- Copy files operation.
- Delete files operation.
- Change backup folder.

Pre-conditions

- None

Post-conditions

- The folder name is changed.
- The files in the folder remain the same.

View revision (Alsharif)

Cross references

- GUI creation.
- Display the file history.
- Revert the revision.

Pre-conditions

- File should be selected.

Post-conditions

- The revision of a selected file is shown using the GUI.

Revert to file revision (Butler)

Cross references

- FileHistory (all but the renaming functions).
- Copy and Diff file operations in FileOp.

Pre-conditions

- A revision for the file must exist.

- Read/Write access currently is available.

Post-conditions

- File is reverted to previous version.
- Version that was replaced is still maintained as a revision.

Change live directory (Butler)

Cross references

- DbManager for config changes.

Pre-conditions

- New location must exist.
- Must have read/write access to that location.

Post-conditions

- The live directory is set to the new location.
- System is running in the same condition it was before the change.

Delete file (Rizvi)

Cross references

- Watcher
- Delete Folder.

Pre-conditions

- User deletes a file.

Post-conditions

- Entries of this file name in the other file lists are removed (to prevent invalid operations).

Create folder (Rizvi)

Cross references

- Watcher

Pre-conditions

- User creates a folder.

Post-conditions

- None

9 - Implementation

Our prototype includes the first run use case as well as unused implementations of several other functions, which don't yet hook into the program. The program will startup, run the first run wizard, and then sit in the background with the file listener monitoring files. It won't yet do anything with the changed files, however.

So far:

- **Mike** has setup the skeleton implementation, the `Watcher` class, the `init()` function of the `DbManager` class, the first run wizard, the`getFolderContents()` function of the `Data` class, the `Error` class, and the startup methods of the `Control` class.
- **Michael** has done the `getConfig()` and `setConfig()` classes of the `DbManager` class.
- **Da** has done the `delete()` and `filesize()` methods of the `FileOp` class.

To build the project, extract the contents of the project's zip file. In Eclipse, choose File > Import and under General, choose to import an existing project into the workspace, choosing the directory you extracted the project's files into (should contain a `.project` file). From here, Eclipse can run the program. The project currently requires Eclipse Juno and JUnit4. Final versions of the project will be setup for distribution so that Eclipse is not needed.

Note that the program is meant to run in the background after the first run wizard is completed. Therefore, you must manually terminate the program.

10 - Project Plan

Implementation has found the need for new methods and approaches, such as how we now store the revisions database in the backup folder instead of in the program folder. This example created a bit more work, but keeps the backup all in one place. Similarly, the first run wizard now allows the user to import an existing backup, the program uses 64 bit integers when working with the database to prevent possible size mismatches, and the GUI is summoned and whisked away with a system tray icon.

The assigned tasks and time estimates are otherwise unchanged from milestone 3 (with the done functionality removed from these lists):

```
                                  Estimated time req
 MIKE:                                 (in hours)
     Control.displayRevision()             3
     Control.displayRevisionChanges()      3
     Control.revertRevision()              4
     DbManager.getRevisionData()           3
                                          ---
                               Remaining:  13

MICHAEL:
     Control.handleCreatedFiles()          4
     Control.handleModifiedFiles()         4
     Control.handleRenamedFiles()          4
     Control.HandleDeletedFiles()          4
     DbManager.renameFile()                3
                                          ---
                               Remaining:  18

DA:
     FileHistory.getRevision()             3
     FileHistory.obtainRevision()          6
     FileHistory.storeRevision()           3
     FileOp.rename()                       2
     FileOp.copy() (both)                  4
     FileOp.fileValid()                    3
                                          ---
                               Remaining:  21

HATTAN:
     Control.restoreBackup()               4
     FileOp.createDiff()                   5
     FileOp.applyDiff()                    5
     FileOp.fileToList()                   3
     FileOp.isFolder()                     3
     Data.getRevisionInfo()                3
     DbManager.insertRevision()            3
                                          ---
                               Remaining:  26

SYED:
     FrontEnd.*                           20
     FileHistory.renameRevision()          3
                                          ---
                               Remaining:  23
```

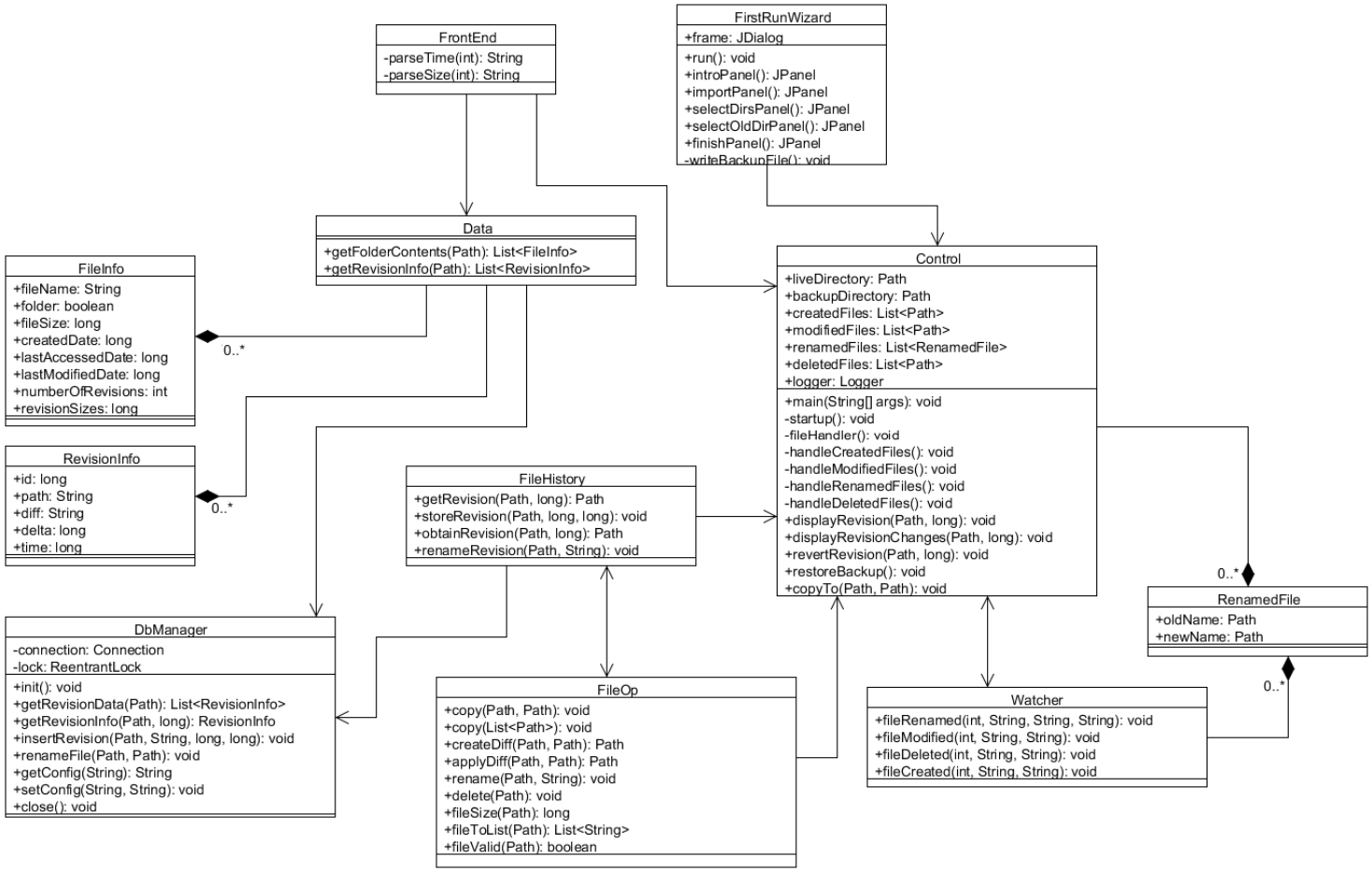
The project is being created with a mixture of agile and concurrent engineering principles. Each group member works on individual components, but code reviews are committed on each person by two others, weekly "SCRUM-like" meetings allow us to discuss changes, difficulties, and overall progress. In general, however, we follow technical document <https://code.google.com/p/fbms/wiki/TechnicalDetails> that changes with our iterations and several other planning documents that are communicated via email. This allows us to keep our planning streamlined and up-to-date, while facilitating communication and ever-changing requirements.

The hours that have been spent so far on the project are:

```
        Mike:
                - Planning and documentation:   5 hours
                - Milestones:                  10 hours
                - Utility demo:                 8 hours
                - Skeleton implemention and
                  PoD classes:                  2 hours
                - Watcher class:                1 hour
                - DbManager.init():             3 hours
                - FirstRunWizard class:         6 hours
                - Data.getFolderContents():     2 hours
                - Various Control code related
                  to startup:                   5 hours
                - JUnit testing code:           2 hours
                - Errors class:                 3 hours
                                        Total: 47 hours
        Da:
                - Rough design of modules:      1 hour
                - Rough design of functions:    1 hour
                - Work on previous milestones:  4 hours
                - Work on FileOp:               3 hours
                - Milestone 4(text):            5 hours
                - Milestone 4(graph)(WIP):      1 hours
                - Test case:                    4 hours
                                      Total:  18 hours
        Ahsen:
                -Milestone 4 (documentation):   5 hours
                -Frontend implementation:       4 hours
                -System Tray implementation:
                -GUI design:

                                     Total:    9 hours

        Hattan:
                -Milestones                     9 hours
                -Work on Presentation slides    3 hours
                                      Total:   11 hours

        Michael:
                - Milestones:                   4 hours
                - getConfig():                  3 hours
                - setConfig():                  3 hours
                                      Total:   10 hours
```

* Note: Times are estimates. Not included is research time, time used by meetings, class time, etc.

11 – Class Diagram



12 - Meetings

Date	Attendance	Discussion
10 September	All	Brainstorming ideas; program choice; identifying strengths
17 September	All	Program layout; functionality; some use cases; Milestone 2
24 September	All	Program components; GUI design; Assigned components; Requirement changes; Milestone 3
1 October	All	Current progress; presentation; code reviews; JUnit
8 October	All	Implementation review
15 October	All	Milestone 4; fleshed out use cases; UML stuff

* Note: In-class collaboration not included