Claim Sheet

SyncSharp Version 2.0

4/15/2010
National University Of Singapore (NUS)
Team Excalibur

1. Features List

- Create, edit and delete synchronization profiles
- Drag and Drop folders to create task
- Import/export synchronization profiles
- Ability to use environment variables in folder paths
- Configure settings for file conflicts
    - **Users can configure various file conflicts settings through task setup form**
- Preview Synchronization tasks
    - Display list of operations to be performed before synchronization
    - Enable users to exclude multiple items from synchronization
    - **Filters the preview display to show only files or folders**
    - **Sort the view in ascending/descending order according to the column clicked**
- Perform 2-way synchronization between source & target folders
    - Able to detect files creation/deletion/modification
    - **Able to detect files and folders renaming**
    - **Able to resolve conflicts automatically through user's settings**
    - **Able to synchronize all folder pairs from the main window**
- Configure inclusion/exclusion filters
    - Support simple files filters (filename/extension)
    - Support exclusion of sub-directories
    - Support files attributes filters (hidden, read only, temp, system)
- Backup files in source folder to target folder
- Restore a previous backup task
- Generate log file after each synchronization operation
- **Auto-sync upon USB device plug in**
    - **Able to remove/re-order tasks which are in queue in PlugSync form during sync. In addition, user can cancel the auto-sync process by clicking the return button before synchronization starts**
    - **Able to configure countdown timer (between 0 – 100 seconds) for auto-sync in global settings form**
    - User is able to set each task to enable/disable PlugSync (enabled by default) in General tab page of task setup form
- Clean:
    - All metadata/log files/profile information are stored in one centralized location. This is set to be in the same folder as SyncSharp.exe in a folder called Profiles.

The files are automatically managed. i.e. When tasks are deleted/renamed, the corresponding data files are also deleted/renamed.

- Updating of root drive letter:
    - SyncSharp is designed to run on removable media. In the event that the removable media's root drive letter is different from previous runs, SyncSharp will automatically update all affected folder paths.
- Checking of free drive space before sync:
    - SyncSharp will estimate the amount of free space required for each synchronization and prevent the user from syncing if the source/target folder does not have enough free space.
- Scalability
    - Our team has run the 1 million files test on V2.0, and it completed successfully. (We can show you the 1 million test case log file around 100 MB)
- Performance
    - We have tested our program against SyncToy and our program performs faster than SyncToy when comes to renaming. The differences are much more obvious when we run the test on PC with more RAM like 4 GB. (Please refer to http://code.google.com/p/syncsharp/wiki/SyncSharpVSSyncToy for the actual results)

Synchronization algorithms (i.e. Detector and Reconciler) were written by our own team members from scratch, we did not make use of any external SDKs like Microsoft Sync Framework to synchronize the files

2. Software Engineering Principles

- **Code Refactoring**
    - We have refactored our code to follow C# naming conventions for functions and data members in classes
    - Unnecessary functions and classes were removed from the solution
    - Classes were grouped into namespaces like *DataModel*, *Storage*, *GUI* and *Business* to follow our architecture design
    - Functions like ***AnalyzeFolderPair()*** and ***SyncFolderPair()*** have been broken down into sub-functions to promote **modularity** and **abstraction**
- **Regression Testing**
    - We did regression testing after we refactored our code to make sure that we did not change the intended behavior of our program. Testing were done by running the ATD to make sure our program passes all test cases
- **System Testing**
    - Our team has did a system testing on SyncSharpV2.0 before the project submission, and Hong Lei has discovered 2 bugs in the system.
        1. SyncSharp crashes when writing metadata files to system protected directories like C:\Windows\System32\.
        2. Applying file filter for first time sync and removing the filter for second time sync will delete the file
- **Defensive programming**
    - Our team has practiced defensive programming in our code. This is prevent error like "*object not set to an instance of object*" or "*null reference exception*"
- **Polymorphism**
    - Our team has applied polymorphism in our project through the use of **Delegates**. Basically delegates are functions pointers that are used to **encapsulate** functions with same signatures and return types. Delegates can be used to achieve **dynamic binding** by executing the respective functions being assigned to it. (Please refer to *MainForm, AutoRunForm* for examples)

    - We have also used Delegates to implement multi-threading for *MainForm* and *AutoRunForm* UIs
- **Inheritance**
    - We have applied inheritance for our project also. Default listView control does not support double buffering, so the control will sometime flickers when displaying items to user. So eliminate this, we have created a custom listView class by inheriting the default listView to remove the flickering.

3. Patterns applied in our project

We have applied the following patterns in our projects

- **The Observer pattern**

Observer pattern was applied in MainForm and AutoRunForm UIs, where SyncSharpLogic class need to update the UIs with analysis and synchronization status in the status bar of the UIs.

In V0.9 and V1.0, what we did is to pass in the *ToolStripLabel* to SyncSharp logic from the UI, so that logic update the *ToolStripLabel* directly. However, this increases **coupling** of our system design and Dr. Damith suggested our team to use Observer pattern during code review.

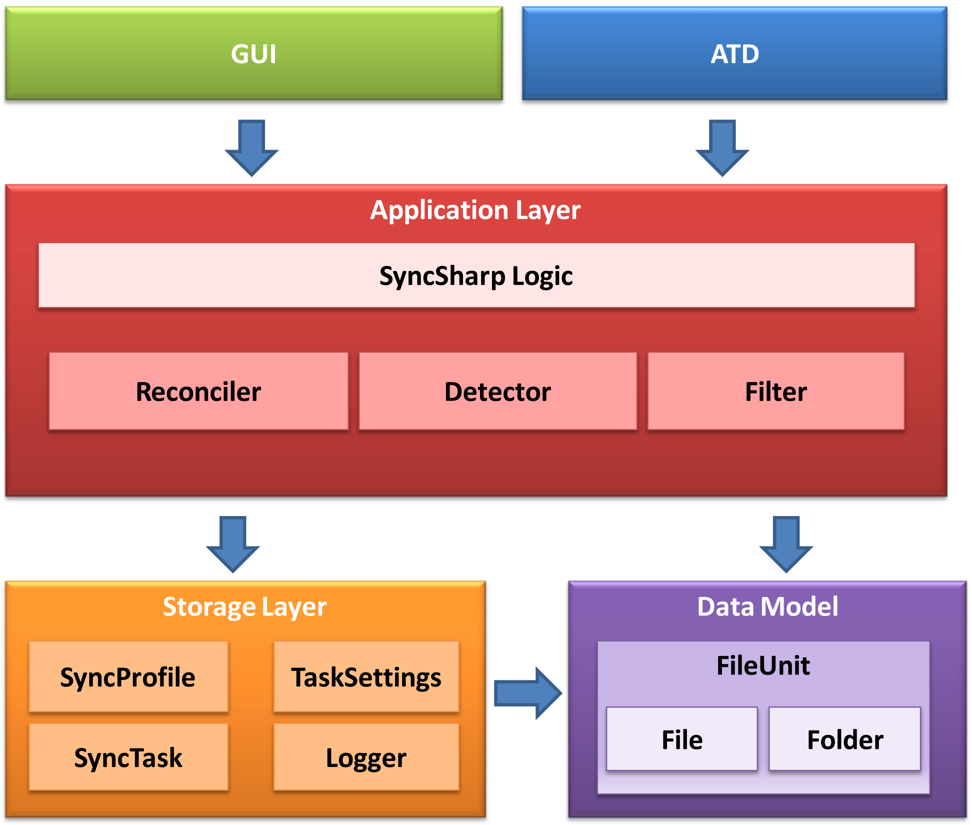
In V2.0, we have remove the *ToolStripLabel* parameters from the functions and made use the Observer pattern in SyncSharpLogic to notify the UI when there is a update to synchronization/analysis status. (See Observer.cs, MainForm, AutoRunFrom for the actual implementation)

- **The Facade pattern**

Our team has also applied the facade pattern in our project. Basically we have a intermediary class (SyncSharpLogic) to handle all functions call request from the MainForm UI. SyncSharpLogic will in turn call the internal functions of the classes to perform the required task. (Please refer to SyncSharpLogic.cs for more details)

- **The Model View Controller pattern**

Moreover, our overall system architecture design follows the Model View Controller pattern. Basically **MainForm** (Main UI) will detect UIs events (controller) and display information (view) to the user. It also updates the DataModel classes like *SyncTask* and *SyncProfile* through our **SyncSharpLogic** class when operations like *AddTask()*, *DeleteTask()*, *SyncFolderPair()* etc are called. This is to separate UI from our Logic class, so that only MainForm (our UI) is aware of the Logic class, and Logic class does not need to know about MainForm. (Please to our system architecture diagram below)


GUI
ATD
Application Layer
SyncSharp Logic
Reconciler
Detector
Filter
Storage Layer
SyncProfile
TaskSettings
SyncTask
Logger
Data Model
FileUnit
File
Folder

4. Automated Test Driver (ATD)

- ATD provides automatic testing of a number of critical features:
- Detector Component
  - Compare folders function: checks if FileList are created properly for the reconciler: 32 scenarios tested varying in complexity

- Reconciler Component
  - Sync function: performs sync based on detector's FileList. Returns successful if both source and target folders are in sync: 32 scenarios tested

- Conflict Settings
  - Target deleted conflict (delete source)
  - Source deleted conflict (delete target)
  - Double conflict (keep latest | keep source | keep target)
  - Rename conflict (rename to target)

  32 test scenarios each

- SyncProfile component
  - Task exist
  - Add task (including all checks on task name and folder pairs)
  - Get task
  - Remove task

  36 Test cases

- SyncSharpLogic component
  - Check profile exists
  - Load profile (together with save profile)
  - Check auto run

  4 test cases

Total number of tests run by ATD: **296**

5. Reconciler’s Algorithm Design

[Accomplishment]

- Our heuristic algorithm is designed to detect file/folder renames and moves. If a file or folder was only renamed or moved, just the rename or move operation on other replica. This can avoid creation and deletion file operations which is extremely on USB drive.
- Our synchronization algorithm is designed to handle over 1 millions files.

[Challenges]

- Existing data structures like <List> aren’t able to achieve desirable performance on large number of files and <Dictionary> aren’t able to support multi keys. A new data structure is needed.
- During the implementation of the new algorithm, a number of problems encountered:
    - Unable to add or remove entry in the data structure, this affected how we can detect renaming. This requires making some adjustment to the algorithm.
    - It is possible for user to have multi files with the content and this means multi entries with the same hash code. The new data structure unable to cater multi entries with the same value. This requires making some adjustment to the data structure and the algorithm.

[End Product]

(Backup and Restore)

- Able to backup data from one replica to another.
- Able to restore data back from the target replica.

(Preview)

- List the required changes on the replicas.
- Allow user to enable or disable the changes to be propagated.

(Sync)

- Propagate changes directly from one replica to another if only one replica is “dirty”.
    - File creation on source. (Expected action: Copy new file from source to target)
    - File creation on target. (Expected action: Copy new file from target to source)
    - File deletion on source. (Expected action: Delete existing file from target)
    - File deletion on target. (Expected action: Delete existing file from source)

  - File renaming on source. (Expected action: Rename existing file from target)
  - File renaming on target. (Expected action: Rename existing file from source)
  - Folder creation on source. (Expected action: Create new folder on target)
  - Folder creation on target. (Expected action: Create new folder on source)
  - Folder deletion on source. (Expected action: Delete existing folder on target)
  - Folder deletion on target. (Expected action: Delete existing folder on source)
  - Folder rename on source. (Expected action: Rename existing folder on target)
  - Folder rename on target. (Expected action: Rename existing folder on source)

- Resolve conflict and propagate changes between replicas.
  - Concurrent modified conflict: (Expected action: Based on user setting; Default: Keep both copies)
  - Concurrent modified-deleted conflict: (Expected action: Based on user setting; Default: Copy file over)
  - Name collision – concurrent create conflict: (Expected action: Based on user setting: Default: Keep both copies)
  - Name collision - concurrent create-rename conflict: (Expected action: Based on user setting: Default: Keep both copies)
  - Name collision - concurrent rename-rename conflict: (Expected action: Based on user setting: Default: Keep both copies)
  - Folders rename/move to different name: (Expected action: Based on user setting; Default: Rename both folder to the same as the source replica)

6. Detector's Algorithm Design

A number of improvements have been made to the detector since the V0.0 release of SyncSharp.

- Changes needed to be made to accommodate the new meta data structure we adopted. We moved from a List of FileUnits to CustomDictionary objects.
- The detector no longer needed to compare files against the source/target folders (except for removing excluded files, and estimating required disc space for sync), as there was no longer a need to determine file/folder actions. This task was transferred to the reconciler. Instead the detector only needed to determine which files were created/deleted/modified on source and target folder separately.
- The V0.0 detector did not have a file/folder exclusion functions. This was one of the major additions to the new detector.

Problems and Solutions

Most of the problems involved folder/file filtering.
Initially the idea was to completely exclude (skip) files/folders that are filtered, as the detector was gathering the current folder information. This resulted in some problems:

- Previous state of excluded files was not captured. This means that when the filters were removed, these files would have been seen as conflicting creations on both source and target (when either one has been modified). This resulted in a lot of unnecessary duplicate files.
- Folders are treated differently from files by the reconciler. Using this method to exclude folders, resulted in their deletion whenever they were excluded.
- Filtering each side separately caused problems when filtering attributes, as users may change the attributes of one file and not the other, resulting in inconsistent behaviour.

These problems were solved by:

- Checking if the same file has been excluded on either source/target folders before excluding them. E.g. when filtering files with read-only attributes. Pairs of files with at least one of them tagged read-only will be filtered.

  - Adding excluded files/folders to the clean list. No actions will be performed on these files, and they will be added into the metadata, saving their state for the next synchronization.

Estimation of disc space required for synchronization:

Problems occurred during first time sync, and if both folders were already synchronized. Our algorithm initially added the file sizes of all dirty files as estimated disc space required. This resulted in double counting. Preventing the users from synching their files, if source/target folder was already greater than the remaining disc size.

This was resolved, by taking the difference in file sizes on their corresponding folders, when a file is detected to have been created/modified. This solves the problems and results in more accurate estimation of disc space required.

7. Logger

Log system

Researched, reviewed and filtered existing log systems already available for the C# .NET platform, narrowing down to one main candidate, log4Net; a quick, lightweight, scalable and flexible logging system. Born from the well-established log4J system used for the Java language; similarly log4Net provides for multi-level logging, and customizable xml configuration file options that would facilitate for future changes.

We had several deliberations and careful considerations, on the basis of pursuing a small footprint and minimal complications. As a consequence, significant had to be incurred before the starting to implement our logging system. We finally opted for a small custom designed logger just sufficient for our purposes.

API docs

In looking to produce an industry standard documentation for our libraries, we set out to review tools that would help achieve just that. Particularly, we were interested in generating a searchable application programming interface that is comprehensive enough to provide for the other major languages in .NET such as VB.Net, etc.

Then we stumbled upon Sandcastle, a tool much awaited, with extensive functionalities to facilitate visual studio project documentation. Having a mostly command-line interface, we did more searching to find a UI interface that would allow us to efficiently tap on the power of the Sandcastle engine.

The DocProject system was one among other tools that did just that. Thereafter came the learning process to setup and reference our existing C# project for documentation. That turned out to be a painful process due to the poor documentation available for both the above tools.

After all, the effort paid off, enabling us to come out with a nifty API doc. At the same time, we begin to realize to a greater extent, the value of good and systematic documentation.