SyncSharp: Plug & Sync

File Synchronization Software

Developer Guide V2.0

4/15/2010

Azhar Bin Mohamed Yasin
Loh Jianxiong Christopher
Hong Lei
Guo Jiayuan
Tian Shuang
Tan Yewkang

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 What is SyncSharp

File synchronization tools are used to synchronize files and folders across multiple computers. Users are able to modify and update files in two or more locations through certain rules. Most synchronization tools provide users with one-way sync, where files and folders are copied in one direction only, while some provide two-way sync, where files and folders are replicated in both locations.

However, most of the sync tools that are available in the market required installation which may considered as a hassle to some users. Not all computers are pre-installed with file synchronization software, and users may not be granted with administrative rights to install software, and this poses problems for users who need to perform file synchronization.

In order to solve the abovementioned problems, our team has developed a file synchronization tool called SyncSharp which provides users with a streamline file synchronization operation and installation free application.

1.2 SyncSharp Features

A summary of SyncSharp features is as follows:

- Create, edit and delete synchronization profiles
- Import/export synchronization profiles
- Ability to use environment variables in folder paths
- Configure settings for file conflicts
- Preview Synchronization tasks
- Perform 2-way synchronization between source & target folders
- Set inclusion/exclusion filters
- Backup files in source folder to target folder
- Restore a previous backup task
- Generate log file after each synchronization operation

1.3 System Requirements

- **Operating System:**

Windows 2000, Windows 2003, Windows XP[1], Windows Vista, Windows 7[2]

- **PC Configuration Requirement:**

128 MB RAM or more, 283KB of hard disk space

1.4 Support and Feedback

- **Technical support**

For technical support, please contact us by email at syncsharp-feedback@googlegroups.com

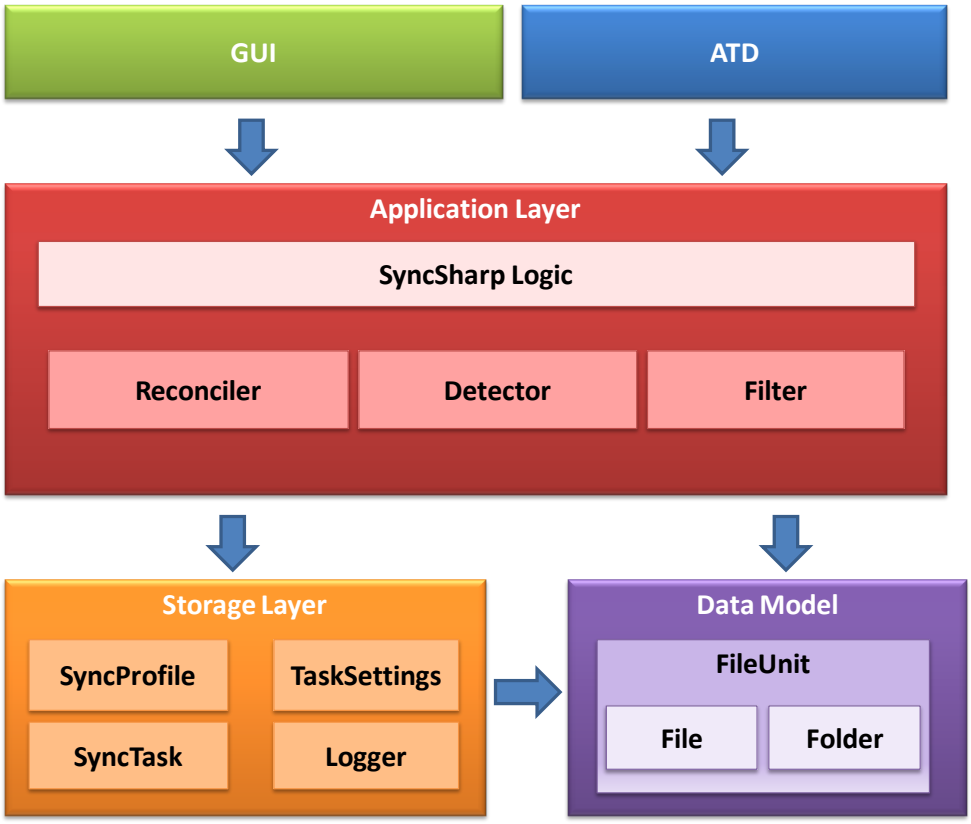
- **Feedback**

If you have any comments or suggestions for the next release, please direct them to http://groups.google.com/group/syncsharp-feedback. Your feedback is highly important for us. In order to get idea of how to make SyncSharp a better product for you, the current release is highly influenced by comments from users.

[1] For current release, "Windows Autoplay" must be enabled for the PlugSync feature (disabled by system default).
[2] See 1.

Chapter 2 Developer Guide

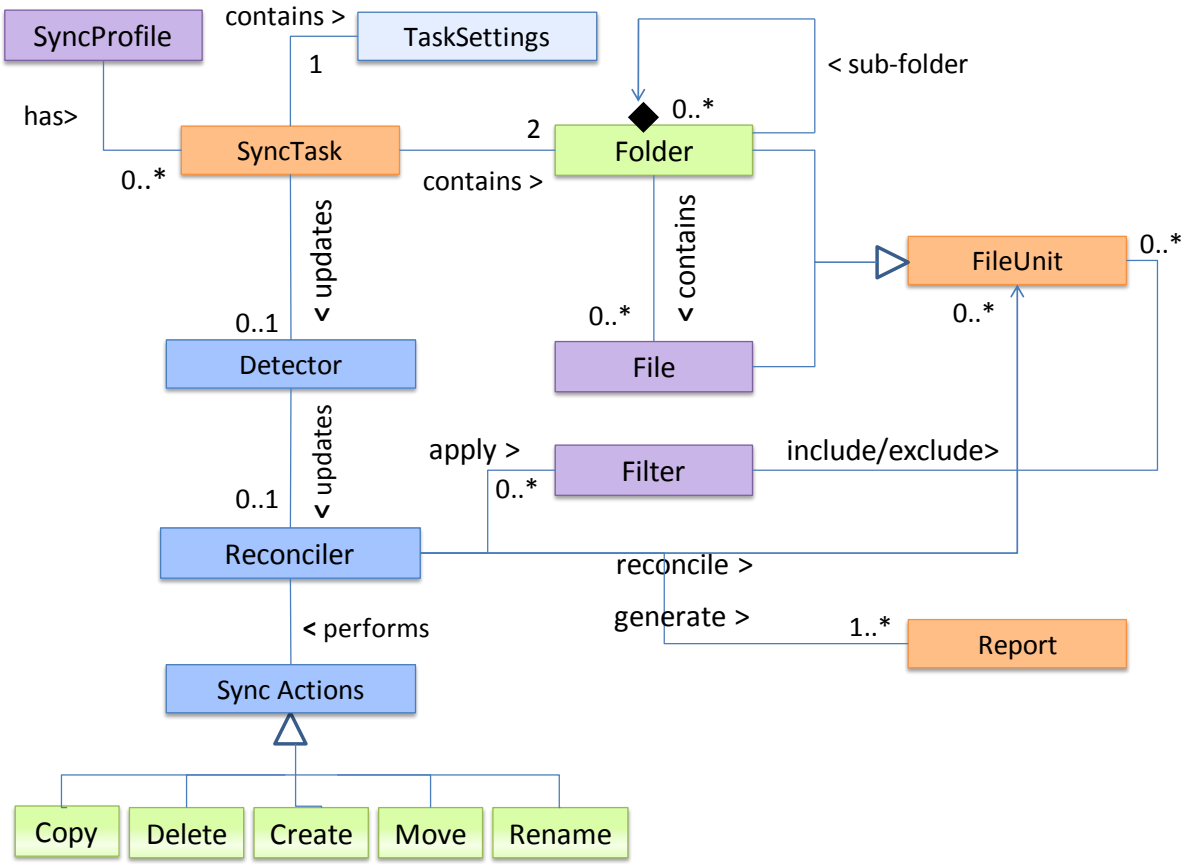
2.1 System Architecture



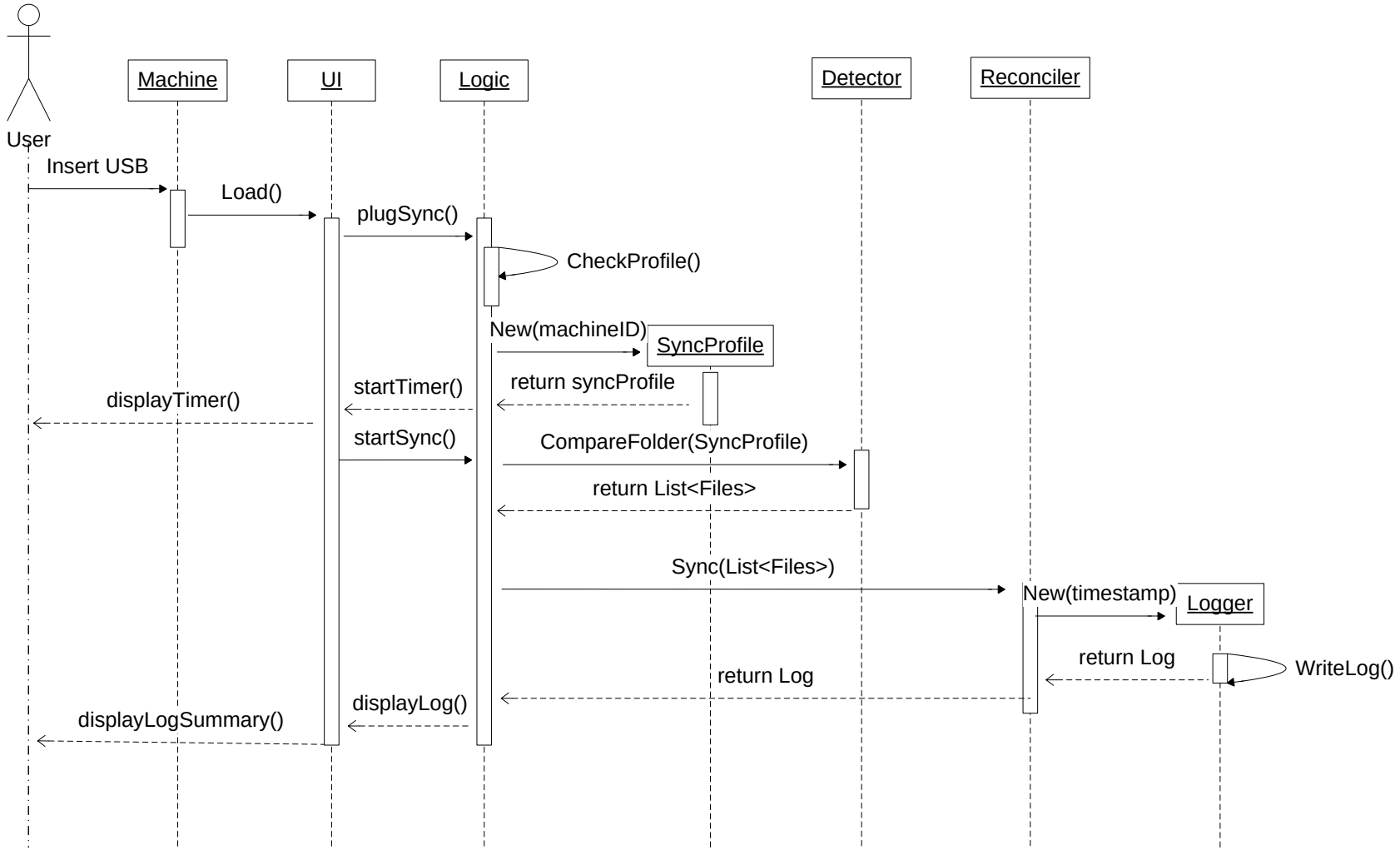
2.2 Overview of System Components

Component	Description
GUI	Provides the interface between users and application.
ATD	Provides automated testing of the application functionalities during development.
SyncSharp Logic	Receives input from GUI component and initiates a response by making function calls to various sub-components.
Detector	Evaluates the changes on the designated folders / files based on the last synchronization operation which stores a small amount of information (called metadata). Metadata captures a snapshot of every file and folders' state. Detector then passes a list of files to the reconciler to perform synchronization.
Reconciler	Performs file synchronization on the list of files obtained from the Detector. The file synchronization operation is based on the analyzed results. In the rise of conflicting updates, pre-determined users' preferences will be used to resolve the updating conflicts. The summary of the updates will be passed to the Logger. Reconciler then updates the metadata of the replica.
Filter	Provides a list of filter rules that will be used by Detector for files retrieval.
Sync Profile	Stores the machine identity and contains a list of SyncTask associated with the profile.
SyncTask	Defines the pair of folders to be used for synchronization
TaskSettings	Stores all the configuration settings for each SyncTask.
Logger	Generates the summary of the file synchronization tasks.
FileUnit	Abstract representation of a file or folder.

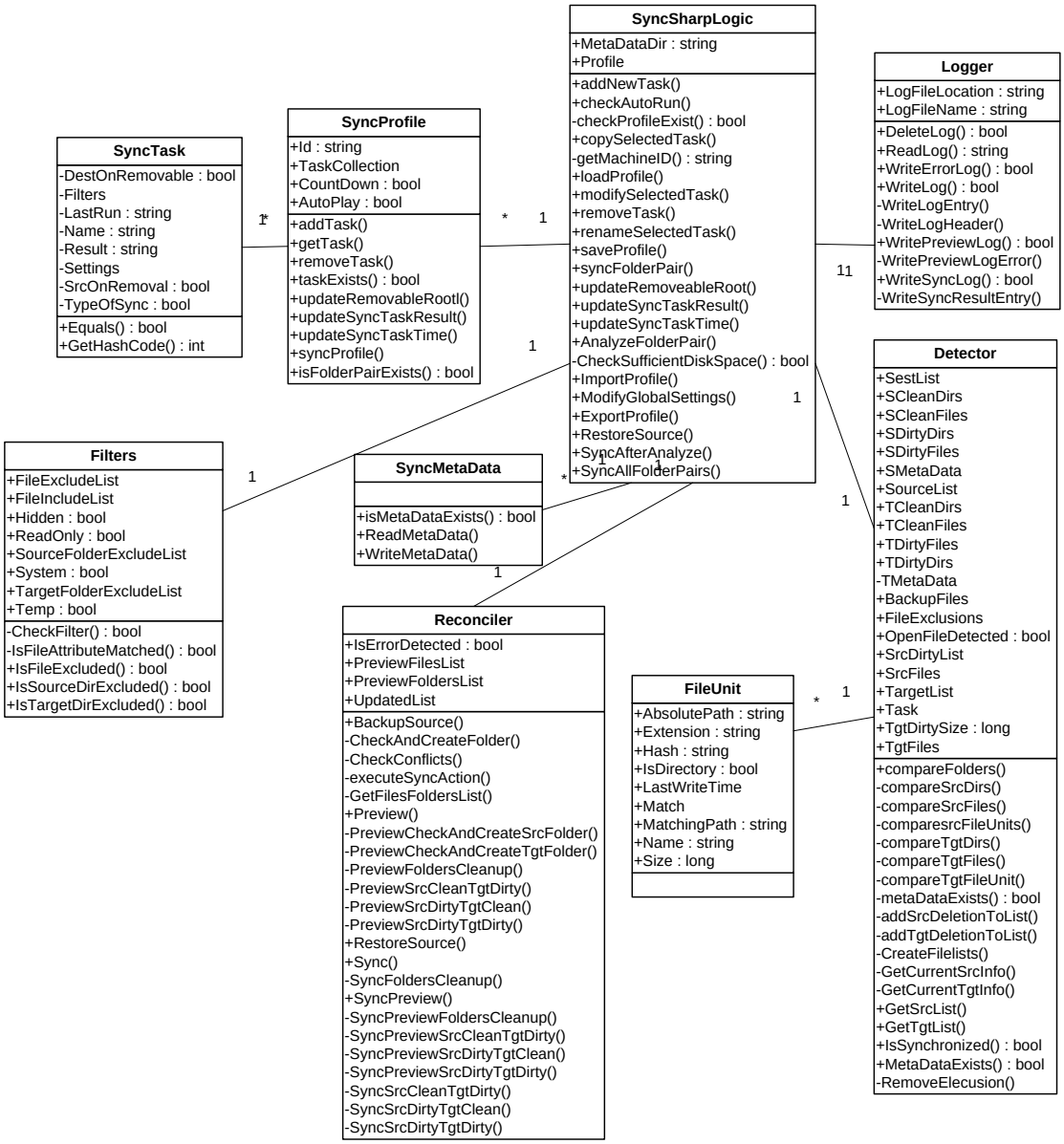
2.3 Domain Model Analysis



2.4 Sequence Diagram



2.5 Class Diagram



2.6 Use Cases Description

The following table is a summary of all use cases

No	User Case
1	Create Synchronization Tasks
2	Edit Synchronization Tasks
3	Delete Synchronization Tasks
4	Run PlugSync
5	Compare Source and Target Directories
6	Perform 2-way Synchronization Between Source and Target Directories
7	Backup Files
8	Restore Files
9	View Source/Target Folders
10	Export Synchronization Profiles
11	Import Synchronization Profiles
12	View Log Files
13	View Help Files

Use Case Number: 1 **Use Case Name: Create Synchronization Tasks**
Pre-Conditions: SyncSharp is running and at the main window
Post-Conditions: System creates new Synchronization task and is displayed on the main window
Actors: User, System
Main Success Scenario: 1. User clicks on "New" 2. System displays new SyncTask setup wizard 3. User enters name for SyncTask 4. System requests for SyncTask type: 'Synchronize' or 'Backup' 5. User selects SyncTask type 6. System requests path for Source and Target folder 7. User enters path for Source and Target folder 8. System creates new SyncTask and updates the main window Extensions(s): 3a. User enters a name that already exists for a SyncTask 3a1. System displays name already exist error

Use case resumes from step 2.
5a. User did not select a SyncTask type before attempting to proceed
5a1. System prompts user to select a SyncTask type
Use case resumes from step 4.
7a. User enters non-existing/empty path for Source/Target folders
7a1. System prompts user to enter a valid path name
Use case resumes from step 6.
7b. User selects same path for Source/Target folders
7b1. System displays error that Source/Target folders cannot be the same
Use case resumes from step 6.

User Case Number: 2 **User Case Name: Edit Synchronization Tasks**
Pre-Conditions: At least 1 SyncTask has already been created
Post-Conditions: SyncTask settings are updated and main window is updated to reflect any changes
Actors: User, System
Main Success Scenario: 1. User selects a SyncTask from the main window and clicks on "Modify" 2. System displays the task setup window 3. User modifies the SyncTask settings as desired 4. System updates the SyncTask settings and updates the main window to reflect any changes Extensions(s): 3a. User provides some invalid settings 3a1. System prompts user to correct any errors Use case resumes from step 2.

User Case Number: 3 **User Case Name: Delete Synchronization Tasks**
Pre-Conditions: At least 1 SyncTask has already been created
Post-Conditions: Select SyncTask is deleted and removed from the main window
Actors: User, System
Main Success Scenario: 1. User selects a SyncTask from the main window and clicks on "Delete" 2. System confirms with user to delete selected SyncTask 3. User selects 'OK' 4. System deletes selected SyncTask and removes it from the main window Extension(s): 3a. User selects 'Cancel'

Use case ends.

User Case Number: 4 **User Case Name: Run PlugSync**
Pre-Conditions: At least 1 SyncTask has been created and PlugSync is enabled for this SyncTask, SyncSharp is run from removable USB device, Computer's AutoPlay is enabled
Post-Conditions: Source/Target folder contents are synchronized
Actors: User, System
Main Success Scenario: 1. User inserts removable USB device 2. System automatically initiates 3. System retrieves a list of SyncTasks from current profile that has PlugSync enabled 4. System displays countdown that PlugSync is about to start 5. User waits for countdown period to end 6. System performs synchronization 7. System returns back to main window. Normal usage continues Extension(s) 5a. User cancels PlugSync by clicking on "Back to Main" Use case resumes from step 7

Use Case Number: 5 **Use Case Name: Compare Source and Target Directories**
Pre-Conditions: At least 1 SyncTask has been created
Post-Conditions: A window is displayed to the user that shows all the differences and SyncActions that would be performed by synchronization
Actors: User, System
Main Success Scenario: 1. User selects SyncTask and clicks on 'Analyze' 2. System compares Source/Target folders and displays results to user Extension(s): 2a. System determines that Source/Target folders are already synchronized, and displays message to user User case ends

Use Case Number: 6 **Use Case Name: Perform 2-way Synchronization Between Source and Target Directories**
Pre-Conditions: At least 1 SyncTask has been created, with 'Synchronization' type
Post-Conditions: Source/Target folder contents are synchronized
Actors: User, System
Main Success Scenario: 1. User selects SyncTask and clicks on 'Synchronize' 2. System proceeds to synchronize the Source/Target folders 3. System updates "Successful", and last run time in the main window for the selected SyncTask Extension(s): 2a. System encounters error during synchronization 2a1. System updates "Unsuccessful", and last run time in the main window for the selected SyncTask Use case ends

Use Case Number: 7 **Use Case Name: Backup Files**
Pre-Conditions: At least 1 SyncTask has been created with 'Backup' type
Post-Conditions: Any changes made to files/folders on Source directory will be updated on Target directory
Actors: User, System
Main Success Scenario: 1. User selects SyncTask and clicks on 'Backup' 2. System proceeds to backup the Source folder to the Target folder 3. System updates "Successful", and last run time in the main window for the selected SyncTask Extension(s): 2a. System encounters error during backup 2a1. System updates "Unsuccessful", and last run time in the main window for the selected SyncTask Use case ends

Use Case Number: 8 **Use Case Name: Restore Files**
Pre-Conditions: At least 1 SyncTask has been created with 'Backup' type
Post-Conditions: Files/folders on the Target directory will be copied to the Source Directory
Actors: User, System
Main Success Scenario:

1. User selects SyncTask and clicks on 'Restore'
2. System proceeds to restore the Target folder to the Source folder
3. System updates "Successful", and last run time in the main window for the selected SyncTask

Extension(s):
2a. System encounters error during restore
2a1. System updates "Unsuccessful", and last run time in the main window for the selected SyncTask
Use case ends

Use Case Number: 9 **Use Case Name: View Source/Target Folders**
Pre-Conditions: At least 1 SyncTask has been created
Post-Conditions: Source/Target folders are opened and displayed to the user using windows explorer
Actors: User, System
Main Success Scenario: 1. User selects SyncTask and clicks on 'Action-> Open Source/Target Folder' 2. System opens and displays Source/Target folders in windows explorer Extension(s): 2a. Source/Target folder does not exist. 2a1. System displays error that Source/Target path cannot be found Use case ends

Use Case Number: 10 **Use Case Name: Export Synchronization Profiles**
Pre-Conditions: At least 1 SyncTask has been created
Post-Conditions: All SyncTasks for profile are exported to a *.profile file
Actors: User, System
Main Success Scenario: 1. User selects 'Export Task' from main menu 2. System requests from user location and filename for exported file 3. User selects location and enters filename for exported file 4. System exports all SyncTasks for current profile into location with filename selected by user

Use Case Number: 11 **Use Case Name: Import Synchronization Profiles**
Pre-Conditions: A SyncProfile has been previously exported
Post-Conditions: SyncTasks from exported profile will be imported and added into current profile
Actors: User, System
Main Success Scenario: 1. User selects 'Import Task' from main menu 2. System request from user location of file to import 3. User selects file to import 4. System imports all SyncTasks from the export file into the current profile Extension(s): 4a. System determines that user selected file to import is not valid 4a1. System displays error message to user Use case resumes from step 2

Use Case Number:12 **Use Case Name: View Log File**
Pre-Conditions: At least 1 SyncTask has been created
Post-Conditions: System displays log file to the user
Actors: User, System
Main Success Scenario: 1. User selects SyncTask and clicks on 'Action -> View Log' 2. System displays log file to user Extension(s): 2a. System cannot find log file associated with selected SyncTask 2a1. System informs user that no log file exists for select SyncTask Use case ends

Use Case Number: 13 **Use Case Name: View Help File**
Pre-Conditions: -
Post-Conditions: Help file is displayed to user
Actors: User, System
Main Success Scenario: 1. User clicks on 'Help' on the main menu 2. System displays help file to user End of User case

2.7 Algorithm Description

2.7.1 Metadata

Custom Dictionary

SyncSharp uses a CustomDictionary class for storing metadata. The concept is similar to a Dictionary object where a key is used to reference a value. The purpose of coding a CustomDictionary is to allow two different keys (called the primary and secondary key) to be associated with the same value, which allows us to quickly and easily detect file and folder renames.

This concept is based heavily on the "C# multi-key generic dictionary" described by Aron Weiler3 with some modifications to be compatible with SyncSharp needs.

- Secondary key can be the same type as Primary key (original implementation not allowed)
- Secondary key needs not be unique (original implementation must be unique)

The CustomDictionary is as follows:

```
public class CustomDictionary<K1, K2, V>
```

It contains three dictionaries within the class as private data members:

```
private Dictionary<K1, V> primary = new Dictionary<K1, V>();
private Dictionary<K1, K2> priSub = new Dictionary<K1, K2>();
private Dictionary<K2, List<K1>> subPri = new Dictionary<K2, List<K1>>();
```

For SyncSharp's usage, K1 and K2 are both Strings, where K1 is the file/folder relative path, K2 is the corresponding file/folders tag/hashcode and V is our FileUnit object.

The primary dictionary stores the relative path as the key and FileUnit as the value. The priSub dictionary stores relative path as the key and tag/hashcode as value. The subPri dictionary stores tag/hashcode as the key and relative path as value. Since the secondary key K2 in this case may not always be unique, all relative paths are instead stored as a list.

[3] http://www.codeproject.com/KB/recipes/multikey-dictionary.aspx

Usage of metadata by the Detector/Reconciler to detect file/folder renames

The lists populated by the detector, are lists of CustomDictionary objects. The "C-", "M-" and "D-" tags are concatenated with the files hash code to form the secondary key. For example, some entries with primary, secondary, value:

("fileA.txt", "D-XYZ123", <FileUnit>) //\fileA.txt with hash XYZ123 is deleted

("fileB.txt", "C-XYZ123", <FileUnit>) //\fileB.txt with hash XYZ123 is created

The reconciler can search for potential renames as it iterates through this list. Taking \fileA.txt, we know from the secondary key that is has been deleted and contains hash code XYZ123. If a search for a corresponding secondary key; create with same hash code i.e. "C-XYZ123" returns true, we know that \fileA.txt has been renamed to \fileB.txt.

Since we cannot hash folders, the detector will append the tag with the primary key to form the secondary key for folders.

The advantage of using such a rename detection technique is that it automatically handles folder renames, as well as file/folder moves, as detection is done through each individual file's hash code and its corresponding "C-"/"D-"tags.

The main flow for the CustomizedDictionary is as follows:

public void Add(K1 primaryKey, V value)

Description:
In the case that a secondary key is not needed. This method will add a primary key, value pair to the CustomDictionary object. This information will be entered into the primary dictionary.

public void Add(K1 primaryKey, K2 secondaryKey, V value)

Description:
Adds a primary key, value pair with corresponding secondary key into the CustomDictionary object. This information will be automatically entered into the primary, priSub and subPri dictionaries.

public void RemoveByPrimary(K1 primaryKey)

Description:
Automatically removes all entries with corresponding primary key in primary, priSub and subPri dictionaries.

public void GetByPrimary(K1 primaryKey)

Description:
Returns the value object with corresponding primary key.

public List<K1> GetBySecondary(K2 secondaryKey)

Description:
Returns the list of all primary keys, with the corresponding secondary key.

public bool ContainsPriKey(K1 primaryKey)

Description:
Retur Returns true, if the CustomDictionary object contains an entry with primary key.

public bool ContainsSecKey(K2 secondaryKey)

Description:
Returns true, if the CustomDictionary object contains an entry with secondary key.

Writing and Reading the MetaData to/from disc

The SyncMetaData class is used to write/read the metadata to/from disc. At the end of the syncronization process, the reconciler will contain the updated metadata for both the source and target folders.

The SyncMetaData class contains two static methods:

public static void WriteMetaData(string path, CustomDictionary<string, string, FileUnit> metadata)

Description:
Parameter path is the location (including full file name) on disc where the meta object is to be serialized to. By default SyncSharp stores all its meta data into one centralized location: "".\Profiles\ + ID + "\"", where ID is the unique ID of the computer that SyncSharp is currently operating on. SyncSharp's convention for meta data naming is 'TaskName'.meta where TaskName is the name of the task without quotes.

public static CustomDictionary<string, string, FileUnit> ReadMetaData(string path)

Description:
This method returns the metadata as a CustomDictionary object. The parameter path is the location (including full file name) on disc where the meta object is to be retrieved from. If the meta data currently does not exist, a null value is returned instead.

2.7.2 Detector

The purpose of the detector component is to go through the source and target folders specified by a sync task, and compare their current state with the metadata state. The information collected by the Detector will be passed to the Reconciler for synchronization. This consists of eight different lists, and they are:

Information regarding the source folder:

```
CustomDictionary<string, string, FileUnit> _sCleanFiles
CustomDictionary<string, string, FileUnit> _sDirtyFiles
CustomDictionary<string, string, FileUnit> _tCleanFiles
CustomDictionary<string, string, FileUnit> _tDirtyFiles
```

Information regarding the target folder:

```
CustomDictionary<string, string, FileUnit> _sCleanDirs
CustomDictionary<string, string, FileUnit> _sDirtyDirs
CustomDictionary<string, string, FileUnit> _tCleanDirs
CustomDictionary<string, string, FileUnit> _tDirtyDirs
```

The constructor for the Detector component is as follows:

```
public Detector(String metaDataDir, SyncTask syncTask)
```

The string parameter metaDataDir states the location where the metadata for the SyncTask is stored. By default, this is set to the location '".\Profiles\" + ID', where ID is the unique ID of the computer that SyncSharp is currently operating on. The SyncTask object refers to the synchronization task that is to be performed.

Determining flags for Directories

For each directory currently on the source/target folder, the following comparison will be made to determine the appropriate flag:

- If the directory does not exist in the metadata, then it is a newly created directory since the latest synchronization. The FileUnit will be added to the DirtyDirs list, with a created flag, "C-", and the metadata is removed from the list.
- Once all directories on the current source/target folder has been looped through. Any metadata remaining will be added to the DirtyDirs list and tagged with a deleted flag, "D-".

If there is no metadata object on disc, then all directories will be tagged with a created flag, "C-".

Determine flags for Files

For each file currently on the source/target folder, the following comparisons will be made to determine the appropriate flag:

- If the file does not exist in the metadata, then it is a newly created file since the latest synchronization. The FileUnit will be added to the DirtyFiles list, with a created flag, "C-", and the metadata is removed from the list.
- Else the following check is done to determine if a file is modified or clean:
  - ✓ If the difference in last modified time is <= x seconds (as determined by the user defined settings) or if the hash code of the current file and the metadata is the same, then we treat the file as clean. It will be added to the CleanFiles list with no flag, and the metadata is removed from the list.
  - ✓ Else the file has been modified in some way since the last synchronization. It will be added to the DirtyFiles list with a modified flag, "M-", and the metadata is removed from the list.
  - ✓ Once all files on the current source/target folder has been looped through. Any metadata remaining will be added to the DirtyFiles list and tagged with a deleted flag, "D-"

Detector's Activity Diagram

Detector's Activity Description

Activity 1: Compare folders

public void CompareFolders()

Description:
Detect file/folder changes on both source and target folders based on folders' current states and its metadata.

Activity 2: Get current source information

private void GetCurrentSrcInfo(List<FileUnit> srcFiles, Stack<string> stack)

Description:
Gets information (name, size, hash code, last modified date, etc. when applicable) for all files/folders and stores them into a list.

Activity 3: Get current target information

private void GetCurrentTgtInfo(List<FileUnit> destFiles, Stack<string> stack)

Description:
Gets information (name, size, hash code, last modified date, etc. when applicable) for all files/folders and stores them into a list.

Activity 4: Remove excluded files/folders (based on filter settings)

private void RemoveExclusions(int sRevPathLen, int tRevPathLen)

Description:
Iterates through the list of source/target files/folders and removes those that are determined to be excluded based on user's filter settings.

Activity 5: Compare source file unites

private void CompareSrcFileUnits(int sRevPathLen, List<FileUnit> srcFiles, List<FileUnit> destFiles)

Description:
Iterating through the source file list, this method will pass the file units to specific methods depending if the file units correspond to a folder or file

Activity 6: Compare source directories

private void CompareSrcDirs(FileUnit u, String folderRelativePath)

Description:
Performs comparison of folders between current sate and meta data state. Iterating through the current state, if a folder exists in the meta data, it is clean, and is added

to the source clean directories list (_sCleanDirs), and its corresponding meta data removed, else it is a new creation and added to the source dirty directories list (_sDirtyDirs), with a created tag "C-".

Activity 7: Compare source files

private void CompareSrcFiles(FileUnit u, String relativePath)

Description:
Performs comparison of files between current state and meta data state. Iterating through the current state, if a file exists in the meta data, checks will be done to determine if it has been modified. A clean file will be added to the source clean files list (_sCleanFiles), a dirty file will be added to the source dirty files list (_sDirtyFiles) with a modified tag "M-", and its corresponding meta data removed. For files that do not contain meta data, it is a new creation and added to the source dirty files list with a created tag "C-".

Activity 8: Compare target file units

private void CompareTgtFileUnits(int tRevPathLen, List<FileUnit> destFiles, List<FileUnit> srcFiles)

Description:
Similar to compare source file units, except repeated for the target file list.

Activity 9: Compare target directories

private void CompareTgtDirs(FileUnit u, String folderRelativePath)

Description:
Similar to compare source directories, except repeated for the target directories.

Activity 10: Compare target files

private void CompareTgtFiles(FileUnit u, String relativePath)

Description:
Similar to compare source files, except repeated for the target files.

Activity 11: Add source deletions to list

private void AddSrcDeletionToList()

Description:
For any source meta data remaining at this point, they will be added to the source dirty files/folders list as deleted files/folders, with the deleted tag "D-".

Activity 12: Add target deletions to list

private void AddTgtDeletionToList()

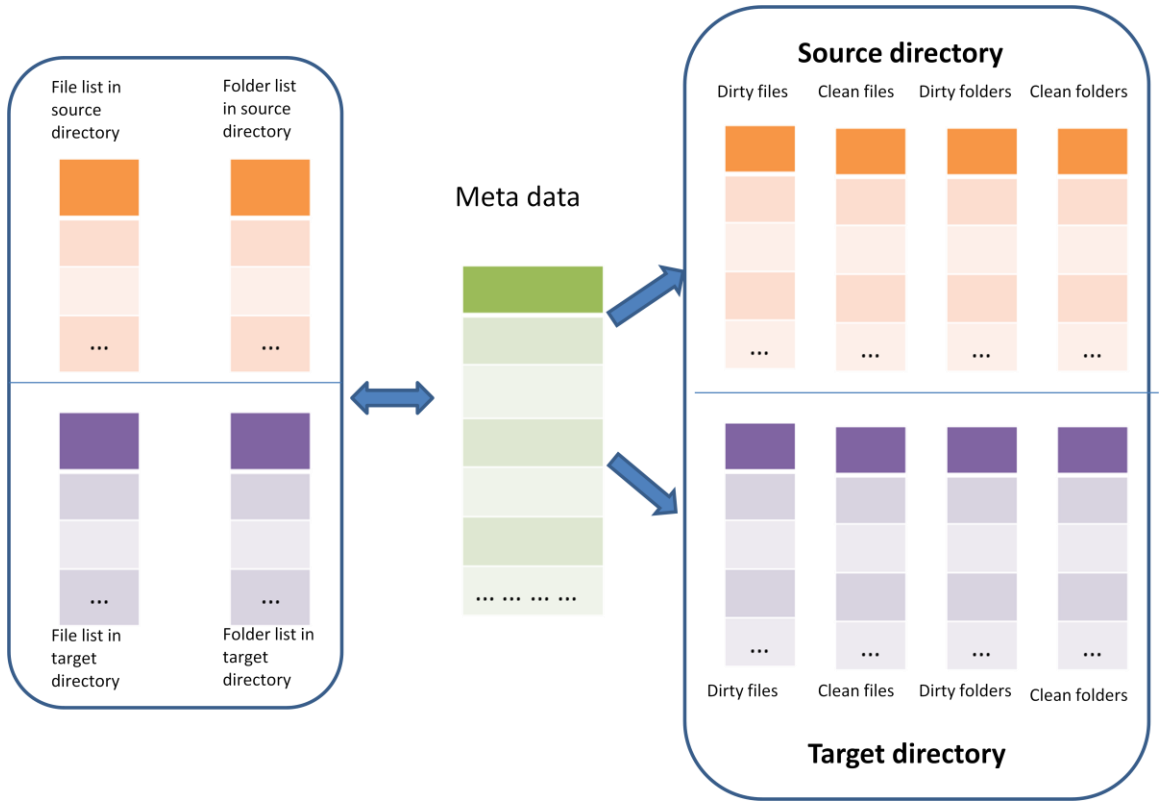
Description:
Similar to add source deletions to list, except repeated for target files.

Activity 13: Create file lists

private void CreateFileLists()

Description:
All the corresponding source/target dirty/clean files/folders lists are wrapped into a source and target FileList object, for the reconciler to process for synchronization.

Overview of Metadata format



Compare the current file/folder status with its corresponding metadata. If the current file/folder has been modified, deleted or created, mark it as "M-", "D-", and "C-" respectively and then put it into the dirty source file/folder list. Otherwise, we put it into clean source file/folder list. We do the same comparison on destination directory.

2.7.3 Reconciler

The reconciler is designed to perform files and folders synchronization between the two replicas. Based on the previous and current states of the replicas, the reconciler attempts to reconcile the conflicts between the replicas in the best possible way. In general, the reconciler conflict resolution follows the two key principles:

- If changes occurred only on one replica, propagate the changes directly to another replica.
- If changes occurred on both replicas, handle the conflicts using the pre-defined user settings and conflict resolution policy.

For each replica, detector generates four different lists:

1. List of files that have changes need to propagate over to another replica.
2. List of files that don't have changes.
3. List of folders that have changes need to propagate over to another replica.
4. List of folders that don't have changes.

Types of changes on each replica:

1. Creation of files and folders
2. Deletion of files and folders.
3. Modification of files.
4. Renaming of files.

For each synchronization task, the reconciler receives eight lists (four lists for each replica) from the detector and will determine the changes to be propagated between the replicas. The following describes how the algorithm performs the conflict resolution.

Note: For ease of explanation, we name the first replica as *source* and second as *target* and the four lists as following:

- ***DirtyFilesList*** - List of files that have changes.
- ***CleanFilesList*** – List of files don't have changes.
- ***DirtyFoldersList*** – List of folders that have changes.
- ***CleanFoldersList*** – List of folders don't have changes.

Algorithm for files

1. Check ***DirtyFilesList*** on each replica for renamed files.
    a. For each entry in the ***DirtyFilesList*.**
        i. Check the entry relative path, flag and hash code.
        ii. If the entry with a "CREATE" flag has another corresponding entry (same hash code but different relative path) with a "DELETE" flag, we consider this is a rename file.
2. Traverse through the ***DirtyFilesList*** on *source* replica to perform reconciling.
    a. For each entry in the ***DirtyFilesList***.
        i. Check any similar changes on the ***target*** replica.
        ii. If no changes made on the *target* replica (check the ***target CleanFilesList)***, propagate changes from *source* to the *target* replica.
        iii. If changes made on the *target* replica (check the ***target DirtyFilesList***), this is consider a conflict and the action taken is based on by the user pre-defined setting and the conflict resolution policy.
3. Traverse through the ***DirtyFilesList*** on *target* replica.
    a. For each entry in the ***DirtyFilesList***.
        i. Propagate the changes from the ***target*** to the ***source*** replica. Note: Only need to propagate changes from the ***target*** to ***source***, since the above function has already handled all conflicting changes between ***source*** and ***target***.

Once the files-level synchronization is done, the reconciler will perform folder cleanup process. The reconciler detects and performs folder creation, renaming and moving through the file level. Each entry in the file list contains the relative path which has the folder information. The file operation performs will create all required folders during

the synchronization. The following describes how the algorithm performs the folder cleanup.

Algorithm for folder:

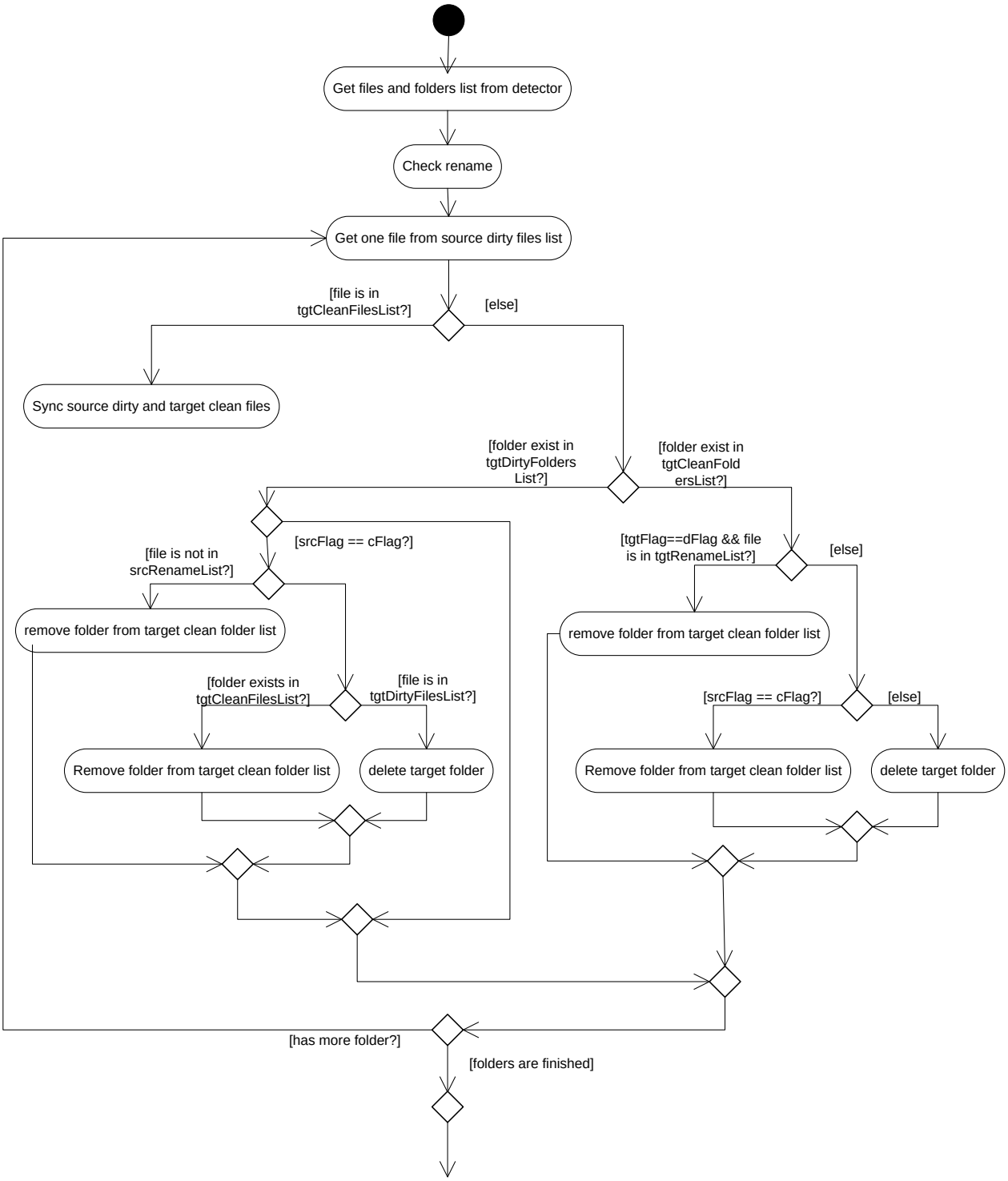
1. Traverse through the ***DirtyFoldersList*** on *source* replica.
    a. For each entry in the ***DirtyFoldersList***.
        i. If the entry with a "CREATE" flag, propagate changes to the ***target***.
        ii. If the entry with a "DELETE" flag and corresponding entry in the ***target DirtyFoldersList***, propagate changes to the target.
        iii. If the entry with a "DELETE" flag and corresponding entry in the ***target CleanFoldersList***, propagate changes to the ***target*** if the ***source*** doesn't have the folder.
2. Traverse through the ***DirtyFoldersList*** on *target* replica.
    a. For each entry in the ***DirtyFoldersList***.
        i. If the entry with a "CREATE" flag, propagate changes to the ***source***.
        ii. If the entry with a "DELETE" flag and corresponding entry in the ***target CleanFoldersList***, propagate changes to the ***source*** if the ***target*** doesn't have the folder.

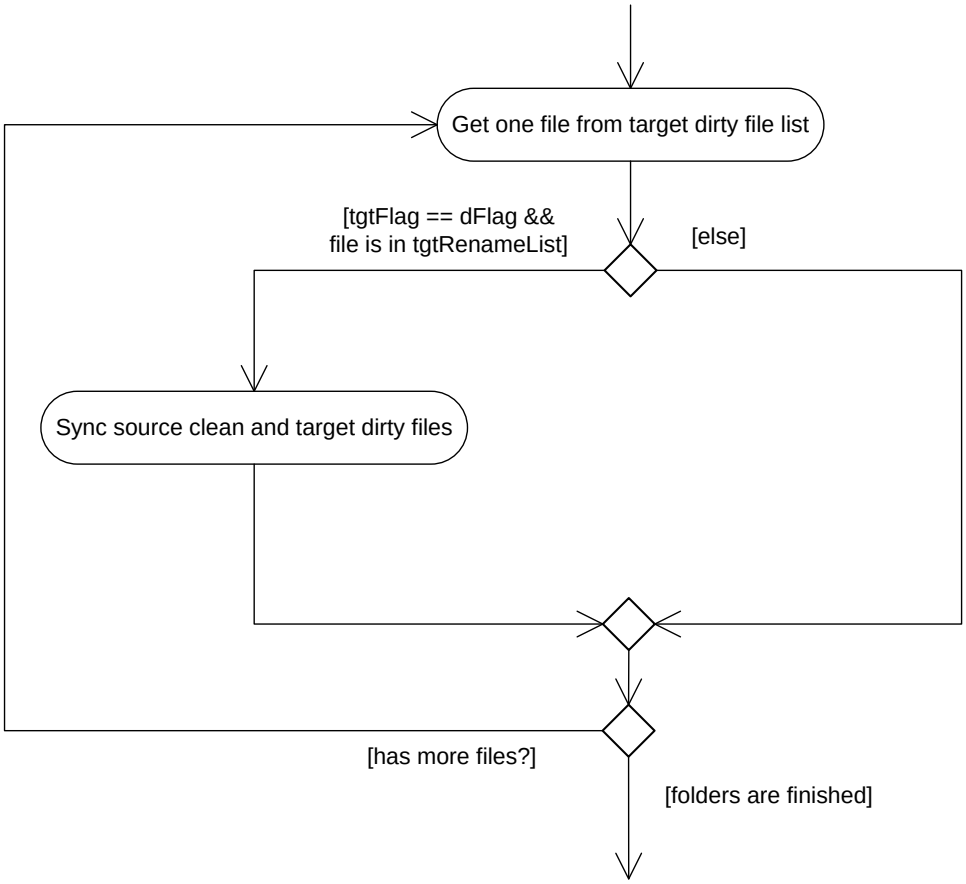
Conflict Resolution Policies

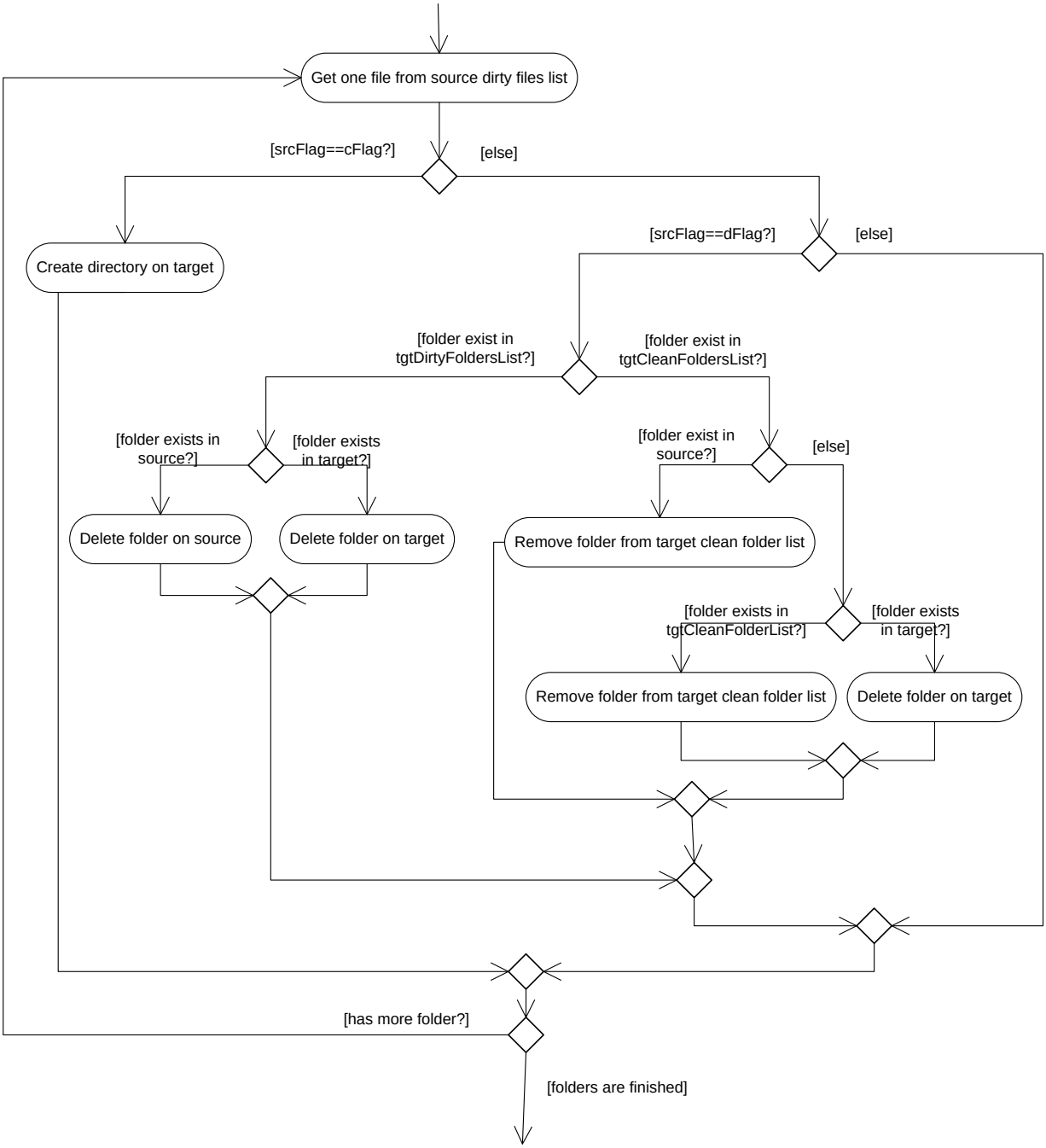
The following policy is used by the reconciler to automatically handle reconciling conflicting changes:

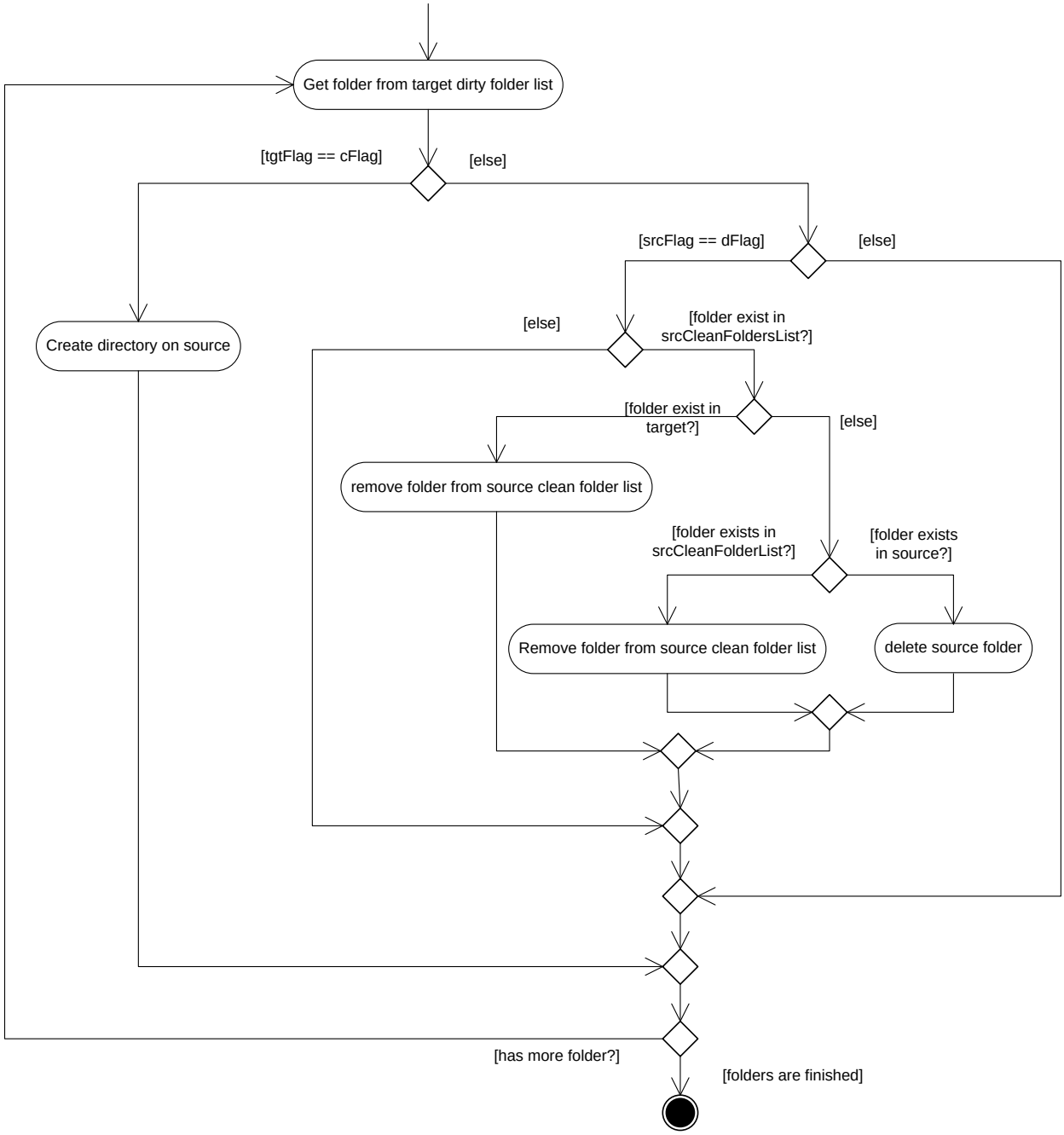
- *Concurrent modified conflict:* If existing file is modified independently on each replica, the reconciler prefers to keep both copies and rename both files to prevent name collision, but user can change this default action through the user setting.
- *Concurrent modified-delete conflict:* If existing file is modified on one replica, and the same file is deleted on another replica. The reconciler prefers to keep the modified file to prevent loss of date. User can change this default action through the user setting.
- *Name collision - concurrent create conflict:* If files with same name are created on replicas, the files' last write time and hash code is used to determine whether the content is similar. If the content is similar, no propagation is needed.
- *Name collision - concurrent create-rename conflict:* If the rename of an existing file collides with the creation of another file on another replica. The reconciler prefers to keep both copies and rename both files to prevent name collision but user can change this default action through the user setting.
- *Name collision – concurrent rename-rename conflict:* If different files on the replicas are renamed to the same name. The reconciler prefers to keep both copies and rename both files to prevent name collision but user can change this default action through the user setting.
- *Folders rename/move:* The existing folders on both replicas are renamed to different name. The reconciler is able to detect the folder renaming through the file level and change both folders to the same name.

Synchronization Action Activity Diagram




Get one file from target dirty file list
[tgtFlag == dFlag &&
file is in tgtRenameList]
[else]
Sync source clean and target dirty files
[has more files?]
[folders are finished]


Get one file from source dirty files list
[srcFlag==cFlag?]
[else]
Create directory on target
[srcFlag==dFlag?]
[else]
[folder exist in tgtDirtyFoldersList?]
[folder exist in tgtCleanFoldersList?]
[folder exists in source?]
[folder exists in target?]
[folder exist in source?]
[else]
Delete folder on source
Delete folder on target
Remove folder from target clean folder list
[folder exists in tgtCleanFolderList?]
[folder exists in target?]
Remove folder from target clean folder list
Delete folder on target
[has more folder?]
[folders are finished]


Get folder from target dirty folder list
[tgtFlag == cFlag]
[else]
Create directory on source
[srcFlag == dFlag]
[else]
[else]
[folder exist in srcCleanFoldersList?]
[folder exist in target?]
[else]
remove folder from source clean folder list
[folder exists in srcCleanFolderList?]
[folder exists in source?]
Remove folder from source clean folder list
delete source folder
[has more folder?]
[folders are finished]

Reconciler's Activity Description

Activity 1: Backup data from source

public void BackupSource(CustomDictionary<string, string, FileUnit> srcList)
Description:
Perform data backup from source replica to target replica.

Activity 2: Restore data from target

public void RestoreSource(CustomDictionary<string, string, FileUnit> srcList)
Description:
Perform data restore from target replica to source replica.

Activity 3: Preview the operations before synchronization

public void Preview()
Description:
Perform synchronization preview between source replica and target replica.

Activity 4: Synchronization the files based on the preview.

public void SyncPreview()
Description:
Perform synchronization between source and target replicas based on the preview results.

Activity 5: Perform synchronization between source and target replicas.

public void Sync()
Description:
Perform synchronization between source and target replicas.

Activity 6: Retrieves the lists of files and folders status from Detector.

private void GetFilesFoldersLists()
Description:
Get lists of dirty and clean files and folders.

Activity 7: Generate lists of renamed or moved files.

private void CheckRename()
Description:
Check the renamed or moved files on the replicas.

Activity 8: Perform synchronization preview between source dirty file and target clean file.

private void PreviewSrcDirtyTgtClean(string relativePath, string srcFlag)
Description:
Perform synchronization preview between source dirty file and target clean file. (Dirty = creation, modification, deletion, and rename)

Activity 9: Perform synchronization preview between source clean file and target dirty file.

private void PreviewSrcCleanTgtDirty(string relativePath, string tgtFlag)
Description:
Perform synchronization preview between source clean file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 10: Perform synchronization preview between source dirty file and target dirty file.

private void PreviewSrcDirtyTgtDirty(string srcRelativePath, string srcFlag, FileUnit srcFile, string tgtRelativePath, string tgtFlag, FileUnit tgtFile)
Description:
Perform synchronization preview between source dirty file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 11: Perform synchronization action between source dirty file and target clean file based on the preview result.

private void SyncPreviewSrcDirtyTgtClean(string relativePath, string srcFlag)
Description:
Perform synchronization action based on preview between source dirty file and target clean file. (Dirty = creation, modification, deletion, and rename)

Activity 12: Perform synchronization action between source clean file and target dirty file based on the preview result.

private void SyncPreviewSrcCleanTgtDirty(string relativePath, string tgtFlag)

Description:
Perform synchronization action based on preview between source clean file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 13: Perform synchronization action between source dirty file and target dirty file based on the preview result.

private void SyncPreviewSrcDirtyTgtDirty(string srcRelativePath, string srcFlag, FileUnit srcFile, string tgtRelativePath, string tgtFlag, FileUnit tgtFile)

Description:
Perform synchronization action based on preview between source dirty file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 14: Perform synchronization preview for folders.

private void PreviewFoldersCleanup()

Description:
Perform the folders operation for previewing.

Activity 15: Perform synchronization action for folders based on the preview results.

private void SyncPreviewFoldersCleanup()

Description:
Perform folder synchronization action based preview.

Activity 16: Perform synchronization action between source dirty file and target clean file.

private void SyncSrcDirtyTgtClean(string relativePath, string srcFlag)

Description:
Perform synchronization between source dirty file and target clean file. (Dirty = creation, modification, deletion, and rename)

Activity 17: Perform synchronization action between source clean file and target dirty file.

private void SyncSrcCleanTgtDirty(string relativePath, string tgtFlag)
Description: Perform synchronization between source clean file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 18: Perform synchronization action between source dirty file and target dirty file.

private void SyncSrcDirtyTgtDirty(string srcRelativePath, string srcFlag, FileUnit srcFile, string tgtRelativePath, string tgtFlag, FileUnit tgtFile)
Description: Perform synchronization between source dirty file and target dirty file. (Dirty = creation, modification, deletion, and rename)

Activity 19: Perform synchronization for folders.

private void SyncFoldersCleanup()
Description: Perform the folders cleanup after file synchronization.

Activity 20: Check files conflicts

private SyncAction checkConflicts(FileUnit sourceDirtyFile, FileUnit destDirtyFile, string s, string t)
Description: Select the desired action for conflicting files.

Activity 21: Execute synchronization action

private void executeSyncAction(FileUnit srcFile, FileUnit tgtFile, string srcFlag, string tgtFlag, string srcPath, string tgtPath)
Description: Perform file operations for conflicting files.

Activity 22: Preview folder for source replica.

private void PreviewCheckandCreateSrcFolder(string relativePath)
Description: Perform folder operation preview for source replica.

Activity 23: Preview folder for target replica.

private void PreviewCheckandCreateTgtFolder(string relativePath)
Description: Perform folder operation preview for target replica.

Activity 24: Check and create folders.

private void CheckandCreateFolder(string fullPath)
Description: Check for directory existent. If directory not exists, perform directory creation.

Handling File Name Collision Conflicts:

The file synchronization tool always follows certain key principles when it is trying to resolve conflicts. Conflict resolution must be done in a way that makes sure the convergence of data across all replicas. The most important issue is that data loss must be avoided in all possible scenarios. The general policies used by the file synchronization tool are listed below:

Letters"M", "D", "C", "R" represents the files' flags which were received from Detector.

"M" – "Modified" (i.e. file was modified)

"D" – "Deleted" (i.e. file was deleted)

"C" – "Created" (i.e. file was newly created)

"R" – "Rename" (i.e. file was renamed from another file)

We allow users to decide how the sync action will be performed based on four user settings, ***keep both copies***, ***keep source copy***, ***keep target copy,*** *and* ***keep latest copy***. Our default action is to ***keep both copies***.

(In this algorithm, we treat ***file move*** the same as ***file rename***. So there will be no move flag or actions exclusively design for move.)

Conflict Resolution Settings

The following describes the different settings that the user may choose for automatic conflict resolution:

- Concurrent Modified Conflict:
  - ✓ Keep both copies: To prevent any unintended data loss, both files are kept on source and target folders. To handle this, files that originate on the source will be appended with a (1), and files that originate on the target will be appended with a (2) to the file name.
  - ✓ Keep latest copy: The last modification time is used to determine the most recently updated file to propogate.
  - ✓ Source overrides target: The source file will always override the target file in a concurrent modified conflict.
  - ✓ Target overrides source: The target file will always override the target file in a concurrent modified conflict.

- Concurrent Modified-Delete Conflict:
  For conflicts where the source file has been modified and the target file deleted:

  - ✓ Copy file to target: The source file will be copied to the target directory.
  - ✓ Delete file from source: The source file will be deleted to match the state of the target directory.

  For conflicts where the target file has been modified and the source file deleted:

  - ✓ Copy file to source: The sould file will be copied to the target directory.
  - ✓ Delete file from target: The target file will be deleted to match the state of the source directory.

- Concurrent Folder Rename/Move:
  - ✓ Rename to source: When both source and target folders are renamed. The reconciler will rename the target folder to match the source folder.
  - ✓ Rename to target: When both source and target folders are renamed. The reconciler will rename the source folder to match the target folder.

2.7.4 Logger

Throughout the duration period of the runtime for a particular SyncTask, Logger handles:

1. Log recording operations from
    a. Sync preview plan summary
    b. File or folder I/O operations.
    c. Sync results summary
    d. System thrown exceptions (during current sync session).
2. Deletion of log file
3. Reading of log file contents

The location of stored log file is at "***[root path of Syncsharp.exe]***/Profiles***/[Machine ID]/ [Sync task name]/***"

Logger call operation flow:

Method calls for a SyncTask should adhere to the following order

WriteSyncLog () *//start syncTask*
WritePreviewLog () *//log finalised preview plan*

WriteLog() *// Record each file/folder operation*
…

…
WriteSyncLog () *//end syncTask*

Attributes

public enum LogType { CopySRC, CopyTGT, DeleteSRC, DeleteTGT, RenameSRC, RenameTGT, CreateSRC, CreateTGT };

Enum types:
CopySRC: File copy operation that originates from source location.
CopyTGT: File copy operation that originates from target location.
DeleteSRC: File copy operation that originates from source location.
DeleteTGT: File copy operation that originates from target location.
RenameSRC: File copy operation that originates from source location.
RenameTGT: File copy operation that originates from target location.
CreateSRC: Folder creates operation that originates from source location.
CreateTGT: Folder creates operation that originates from target location.

Algorithm Description

Activity 1: Write Sync Log

Public static bool WriteSyncLog(string metaDataDir, string syncTaskName, bool start)

Description:
Set begin or end (to signal start and completion) of a particular SyncTask log recording.
Logic:

```
if (start)
    WriteLogHeader();
else
    WriteSyncResultsLogEntry();
```

Begin: ***WriteLogHeader****()*

- *Append sync begin log header to log file for current sync task session.*
- *Append record timestamp.*
- *Reset file and folder operation counters.*

End: ***WriteSyncResultsLogEntry****()*

- *Append file & folder I/O operation results count summary*
- *Record timestamp.*
- *Append log footer for current SyncTask.*

Activity 2: Write Preview Log

public static bool WritePreviewLog(string metaDataDir, string syncTaskName, int srcCopyTotal, long srcCopySize, int srcDeleteTotal, long srcDeleteSize, int srcRenameTotal, long srcRenameSize, int tgtCopyTotal, long tgtCopySize, int tgtDeleteTotal, long tgtDeleteSize, int tgtRenameTotal, long tgtRenameSize)

Description:
Record file I/O operation count summary to be carried out for sync preview for current sync task session into log file

Activity 3: Write Log

public static bool WriteLog(LogType logType, string srcPath, long srcSize, string tgtPath, long tgtSize)

Description:
Record file I/O operation count summary to be carried out for sync preview for current sync task session into log file.
Parameters:

logType: *name of sync task*
srcPath*: file path on source path.*
srcSize*: filesize(bytes) on source.*
tgtPath*: total number of files to copy from target path.*
tgtSize*: filesize(bytes) on target.*

Expected parameters when calling for particular ***logType****:*

CopySRC*: requires file path and size to copy from source and to destination*
WriteLog(LogType.CopySRC, ***srcPath, srcSize, tgtPath, tgtSize****);*

CopyTGT*: requires file path and size to copy from destination and to source*
WriteLog(LogType.CopyTGT, ***srcPath, srcSize, tgtPath, tgtSize****);*

RenameSRC*: requires file path and size to rename from source and to destination*
WriteLog(LogType.RenameSRC, ***srcPath, srcSize, tgtPath, tgtSize****);*

RenameTGT*: requires file path and size to rename from destination and to source*
WriteLog(LogType.RenameTGT, ***srcPath, srcSize, tgtPath, tgtSize****);*

DeleteSRC*: requires file path and size to delete from source*
WriteLog(LogType.DeleteSRC, ***srcPath, srcSize****, null, 0);*

DeleteTGT*: requires file path and size to delete from destination*
WriteLog(LogType.DeleteTGT, null, 0, ***tgtPath****,* ***tgtSize****);*

CreateSRC*: requires folder path and size to create on source*
WriteLog(LogType.CreateSRC, ***srcPath****, 0, null, 0);*

CreateTGT*: requires folder path and size to create on destination*
WriteLog(LogType.CreateTGT, null, 0, null, ***tgtSize****);*

Note: null or 0 values passed in as input parameters have no relevance for the particular selected logtype in use. For these unused parameters for specific operations, caller passes in -Path = null; -Size = 0;

Activity 4: Write Error Log

public static bool WriteErrorLog(string errorMsg)

Description:
Reports general program error exceptions thrown into log file

Activity 5: Delete Log

public static bool DeleteLog(string syncTaskName)

Description:
Delete log file and return whether successful.

Activity 6: Read Log

public static string ReadLog(string syncTaskName)

Description:
Read log file and return log contents to caller.

Chapter 3 Glossary

Term	Definition
SyncSharp	Name of our sync tool
SyncTask	Configuration file that contains source and target directory information to be synchronized
SyncProfile	Contains list of SyncTasks for a Particular PC/Laptop
FileUnit	Abstract representation of a file or folder, contains information such as name, size, hash code, last modified date, etc.
PlugSync	A feature of our program. Program will synchronize all tasks automatically when the USB device is plugged into a computer. Upon execution, the program will count down for 5 seconds waiting for user interruption. If there is no interruption, the program will proceed to synchronize all the tasks listed in the window.
1 way sync	Update destination directory to have the same content as source directory
2 ways sync	Update source and destination directories to have the same state
Report/Logger	Log file that records the operations perform in the synchronization process
Target	The destination directory to be sync or compared
Detector	The sub-system that detect changes of the source or destination directory
Reconciler	The sub-system that resolves conflicts between the source & destination directories
TaskSettings	Contains configuration settings made for each SyncTask