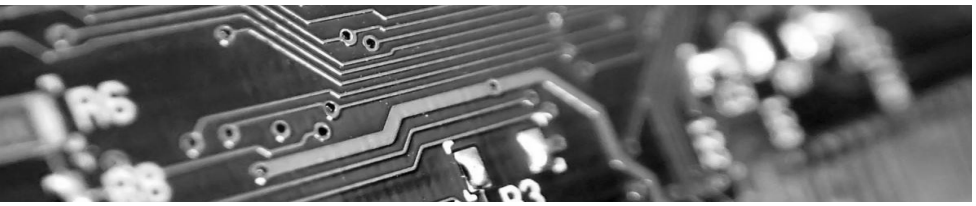
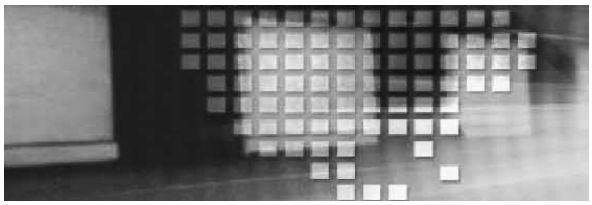
CS3215 Software Engineering Project

NETS

Developer Guide





Version 2.0

By:
Dang Thu Giang
Hoang Nguyen Nhat Tao
Nguyen Hoang Hai
Nguyen Thi Yen Duong
Tran Binh Nguyen
Vu An Hoa

Table of Content

1. Introduction

What are covered in this section:

- ✓ What is **NETS**?
- ✓ Information
- ✓ What's New in Version 2.0

WHAT IS NETS?

NETS is an **open-source File Synchronization** software, written in **C# language** on **.NET framework**.

This document covers the **design and implementation details** of **NETS**. It also includes **suggestion on improvement** of the program structure and features.

INFORMATION

- ✓ Current Version: 2.0
- ✓ Operation System: Windows XP/Windows Vista/Windows 7
- ✓ System Requirement: .NET framework 3.0+
- ✓ Website: http://nothing-else-to-sync.info

WHAT'S NEW IN VERSION 2.0

- ✓ **Smart Sync:** can now be done via folders drop-and-drag besides right-click
- ✓ **IVLE Sync:** able to select which modules to download
- ✓ **Faster** sync for big files
- ✓ **Polished** and **improved GUI**
- ✓ **Bugs fixed** and enhancements

2. Feature Overview

What are covered in this section:

- ✓ **Basic** Synchronization
- ✓ **Right-Click** Features
- ✓ **IVLE** Synchronization

BASIC SYNCHRONIZATION

- Supports synchronizing content of two folders **on the same computer**
- Supports synchronizing folders on different computers through **intermediate storage media** (USB, Flash drive, Portable Hard disk, etc.)
- Supports **one-way** and **two-way** synchronization
- Supports **file filter**
- Supports folders **Drag-and-Drop**
- Supports saving sync information via **profiles** to execute the next time. Profile name is **automatically suggested** by the program and can be customized
- Supports executing **multiple sync jobs**
- Supports Delete To Recycle Bin/Delete Permanently options
- Supports action **logging** actions during sync process for future review

RIGHT-CLICK FEATURES

- **Smart Sync:** an **intelligent** and **automatic** synchronization feature which can dynamically detect the corresponding partnership of a chosen folder and synchronize them. It also supports **multiple folder selection** and handles each of them sequentially.
- **Sync With…**: prompt users for destination folder to sync with the chosen folder. It does not support multiple folder selection
- **Smart Sync** and **Sync With…** entries can be added/removed from **right-click menu** via customized settings

IVLE SYNCHRONIZATION

- A unique feature targeted at **National University of Singapore (NUS) students**
- Synchronizes **Integrated Virtual Learning Environment (IVLE) Workbins** with a **local folder** on student's PC
- **Module Filter**: allows students to choose **which module workbins** to sync

3. Design and Implementation

What are covered in this section?

- ✓ GUI Screenshot
- ✓ Use Cases
- ✓ Architecture
- ✓ Component Classes Overview
- ✓ Sequence Diagrams
- ✓ Important APIs

GUI SCREEN SHOT

Figure 1: Main Window

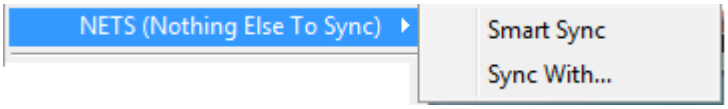


Figure 2: Context Menu Entries

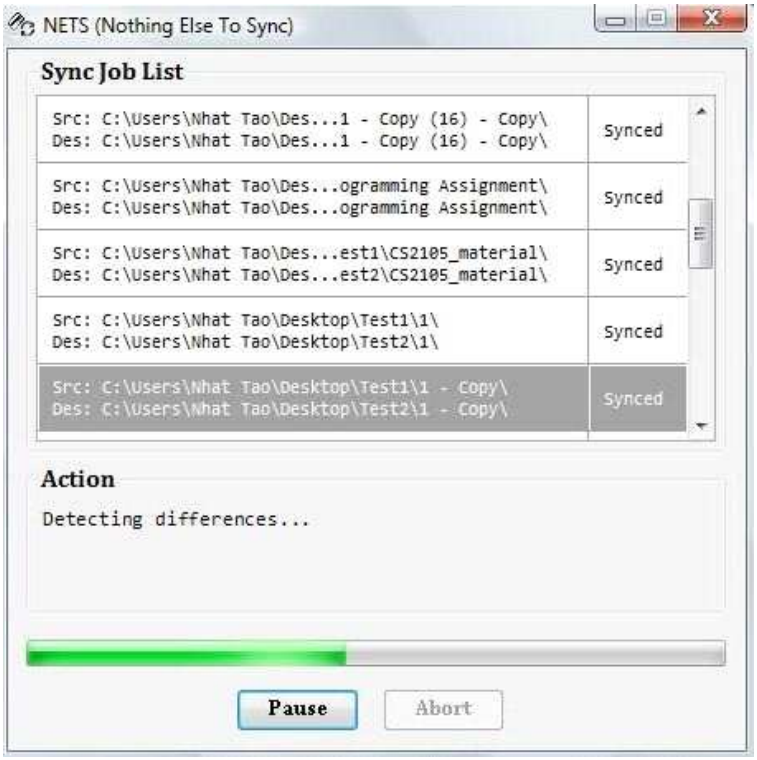


Figure 3: Context Menu Entries

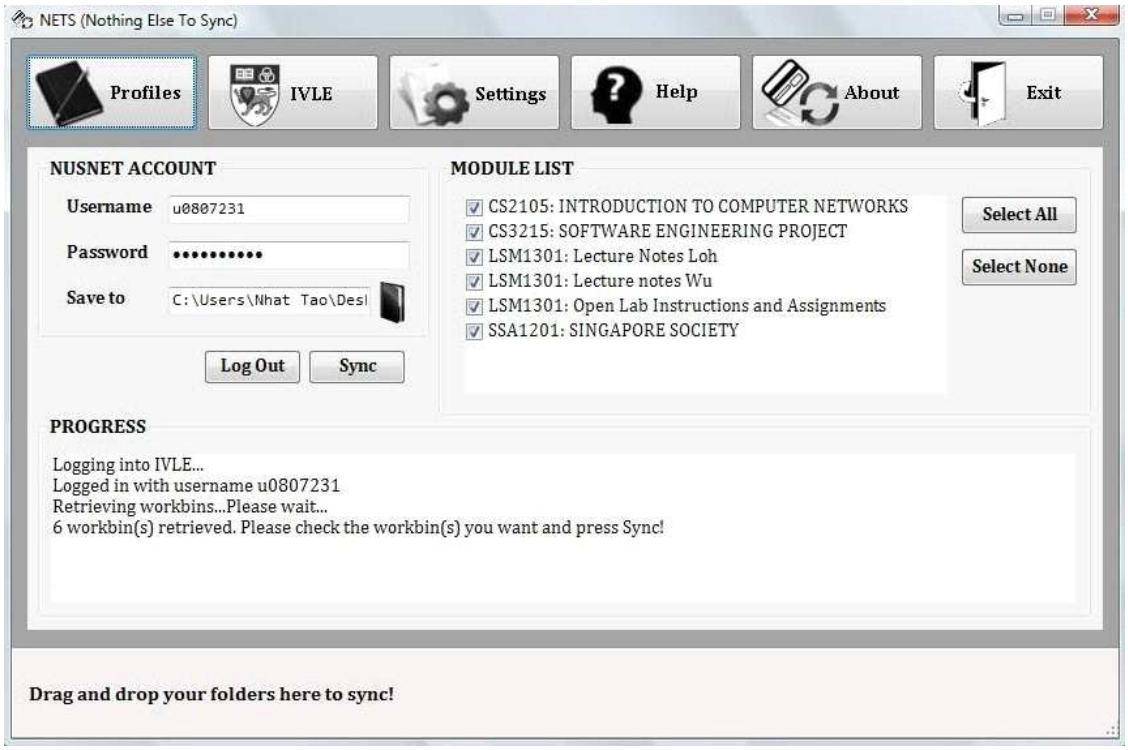


Figure 4: IVLE Page

USE CASES

Use case: 01 – SYNC TWO FOLDERS
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running.
Guarantees: NA
MSS:

1. User provides two folders to be synced and other information.
2. User chooses to sync two folders.
3. NETS informs user upon sync completed.

Extensions:

2a. User does not provide enough/valid sync information.
2a1. NETS asks user for more/valid sync information.
2a2. User provides more/valid information.
Step 1a1-1a2 repeats until user provides enough and valid sync information.

*a. User chooses to abort the sync job.
*a1. NETS requests user for confirmation.
*a2. User confirms the abortion.
Use case ends.

Use case: 02 – SMART SYNC MULTIPLE FOLDERS
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NA
Guarantees: NA
MSS:

1. User selects multiple folders.
2. User chooses to smart sync these folders.
3. NETS informs user upon completed sync.

Extensions:

2a. One of the selected folders does not have partnership folder
2a1. NETS asks user to provide partnership folder for that folder to sync.
2a2. User provides partnership folder for that folder.
Steps 2a1-2a2 repeats until all folders have been processed or no more folders does not have partnership folder.

2b. One of the selected folders has multiple partnership folders
2b1. NETS asks user to select one partnership folder for that folder to sync.
2b2. User selects one partnership folder for that folder.
Steps 2b1-2b2 repeats until all folders have been processed or no more folders has multiple partnership folders.
*a. User chooses to abort the sync job.
*a1. NETS requests user for confirmation.
*a2. User confirms the abortion.
Use case ends.

Use case: 03 – SYNC IVLE WITH LOCAL FOLDER
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running internet connection available
Guarantees: NA
MSS:
1. User provides username, password and local folder.
2. User chooses to login.
3. NETS informs users of successful login.
4. NETS displays module list.
5. User chooses modules to sync.
6. NETS informs user upon sync completed.

Extensions:
2a. User does not provide enough/valid information to login.
2a1. NETS asks user to provide more/valid information.
2a2. User provides more/valid information.
Steps 2a1-2a2 repeats until user provides enough and valid information to login.
Use case resumes from step 3.
5a. User does not provide valid local folder to sync with IVLE.
5a1. NETS asks user to provide local folder.
5a2. User provides local folder.
Steps 2a1-2a2 repeats until user provide valid local folder to sync.
Use case resumes from step 6.
*a. User chooses to cancel sync.
Use case ends.
*b. There is no more internet connection.
*b1. NETS informs user of internet disconnection.
Use case ends.

Use case: 04 – CREATE NEW PROFILE
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running
Guarantees: New profile will be created and saved.

MSS:

1. User fills in profile information.
2. User chooses to save profile.
3. NETS displays the updated profile list.

Extensions:

2a. User does not provide enough/valid information.
2a1. NETS asks user to provide more/valid information.
2a2. User provides more/valid information.
Steps 2a1-2a2 repeats until user provides enough and valid information.
Use case resumes from step 3.

Use case: 05 – EDIT A PROFILE
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running
Guarantees: Profile is edited and saved.
MSS:

1. User selects a profile to edit.
2. User edits information of that profile.
3. User chooses to save profile.
4. NETS displays the updated profile list.

Extensions:

3a. User does not provide enough/valid information.
3a1. NETS asks user to provide more/valid information.
3a2. User provides more/valid information.
Steps 3a1-3a2 repeats until user provides enough and valid information.
Use case resumes from step 4.

Use case: 06 – DELETE A PROFILE
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running
Guarantees: Profile is deleted.
MSS:

1. User selects a profile to delete.
2. User chooses to delete that profile.
3. NETS requests user for deleting confirmation.
4. User confirms the profile deletion.
5. NETS displays the updated profile list.

Use case: 07 – RUN A PROFILE
System: NETS – Nothing Else To Sync
Actor: User
Preconditions: NETS is running
Guarantees: NA
MSS:

1. User selects a profile to run.
2. User chooses to run that profile.
3. NETS informs user upon sync completed.

Extensions:

2a. One of two folders in the profile is not found.
2a1. NETS informs user of unfound folders.
Use case ends.
*a. User chooses to abort the sync job.
*a1. NETS requests user for confirmation.
*a2. User confirms the abortion.
Use case ends.

ARCHITECTURE

The application consists of three main components: GUI, Logic and Storage and other helper subcomponents.

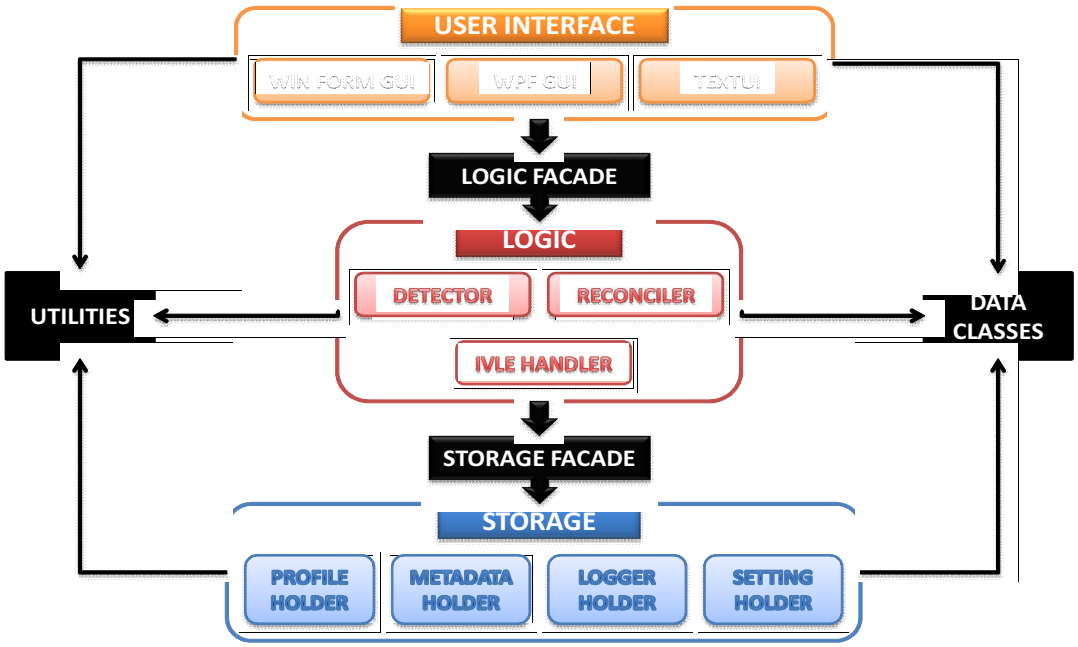


Figure 5: Architecture

GUI

NETS have three UI versions: **Text UI**, **WinForm GUI** and **WPF GUI**. **Text UI** version is basically for debugging purpose and unrevealed to user. **WPF GUI** will be available in next releases. In this release, our main UI is **WinForm GUI**, which is implemented using Window Form Library.

We use **multithreading** to make our **GUI** more responsive. Also, **background worker** allows **NETS** to do IVLE sync in parallel with other sync jobs.

Moreover**, Observer pattern** is used to make achieve loose coupling between **GUI** and **Logic.**

Our **GUI** is designed to be **fast**, **responsive** and **resource-effective**. We have **one Window Form** and **multiple Panels**. Each time we want to display different pages, we can easily switch between Panels in only one Window Form. This is achieved by using **Tab Control**, which is fast and responsive. Creating many **Window Forms** or changing Panels without using **Tab Control** could be slow and cost more resources.

Lastly, we have one central class to handle events occurring in **GUI** called **GUIEventHandler.** Its presence allows us to maximize the change of **reusing codes**, especially when many classes listen to the same events.

Logic

Logic is the core components of the program. It can detect file manipulation inside a particular folder based on information from the last synchronized job. After user sync two folders, file information inside each folder will be saved by **Storage**. When users sync these two folders again, **Logic** can detect changes inside each folder and reconcile two folders based on last synchronizing repository.

The file information is stored as **metadata** in a centralized database. We use both **checksum** and **last date modified** for metadata.

That is how our **Logic** works. Besides, we also strive to improve the cohesion and loose coupling inside **Logic**. The task of detecting the changes in each folder is separated from reconciling them. For each sync job, a list of differences of two folders will be produced from **detecting task** and handled by **reconciling task** after that.

To prevent **GUI** from accessing into different classes in **Logic**, we apply **Façade pattern** and provide **LogicFacade** as an interface between **GUI** and **Logic.** Each time **LogicFacade** receives a sync job passed from **GUI**, it will handle by calling other classes in **Logic** accordingly.

IVLE Sync use one-way sync to update the local folder according to **IVLE** module workbins. It uses the same synchronization logic as one-way sync but different IO methods. Logic provides **IVLEHandler** to handle **IVLE** logic.

Storage

Storage component includes service classes to manage underlying data of the running program, including metadata, profile, setting and logger information. To ensure read/write speed to these data, most frequently used information is stored in memory (RAM) and written back to file at certain occasions.

All Storage components implements **Singleton pattern** with the **StorageFacade** as the interface with Logic components.

Currently no outside database library is used. However, the way information is stored and retrieved is highly optimized.

- *Note: Data stored in storage is separated by '|' notation. So take note of special input*

COMPONENT CLASSES

GUI

*Class **GUIEventHandler**:*

- Handles events occurred in other **GUI** classes
- Uses multithreading to multitask in **GUI**: one Thread to call **Logic** to perform the sync job and another Thread to display the Window Form.
- Receives sync events passed from other **GUI** classes, passes sync information to Logic to handle in one thread then call another thread to display the progress accordingly.
- It listens to **Logic** as an observer to update the Window Form accordingly each time **Logic** needs to ask user for information or finishes processing one file pair.

*Class **ApplicationWindow**:*

- The Main Window in the program.
- Receives Panels as inputs and adds them to Tab Control for future display.

*Class **PageStart**:*

- It is the default page in Main Window and a Panel in one Tab of the Main Window.
- It contains two subpanels: **PageProfileList** to display list of profiles and **PageProfileDetails** to key in/display information of a specific profile.
- **PageProfileList** is reused to ask user to choose one profile in a profile list when user **smart sync** a folder having multiple partnership folders.
- Page **ProfileDetails** is reused to ask user for partnership of a folder in **sync with…** or when user **smart sync** a folder having no partnership folder.

*Class **PageIVLE**:*

- The page (one Tab) in Main Window for IVLE user.

*Class **PageSettings**:*

- The page (one Tab) in Main Window for user to customize program settings.

*Class **PageAboutUs**:*

- The page (one Tab) in Main Window to display information about the software and developer team.

Logic

*Class **LogicFacade**:*

- Acts as an interface between UI and Logic
- Contains methods called by GUI to synchronize two folders given sync information, manipulate profiles (load, save, edit, delete), load/save program settings.
- It is not only the interface between **GUI** and **Logic** but also between **GUI** and **Storage**

*Class **Detector:***

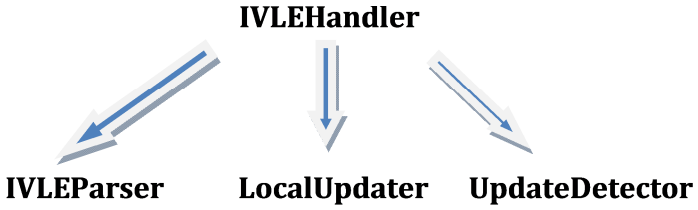
- Detects differences between two given folders
- Receives two input folders and return their list of differences or list of their file pairs which need to be reconciled

*Class: **Reconciler***

- Reconciles two folders by handling their list of differences.
- Receives a list of file pairs and reconcile them accordingly following **one-way/two-way** synchronization rule specified before.

*Class: **IVLEHandler***

- The heart of the IVLE Synchronizer.
- Depends on **IvleParser**, **LocalUpdater** and **UpdateDetector** to handle specific tasks.



*Class: **IVLEHandler***

- Parses (or get information from) the workspace and workbin HTML pages. It extracts information including the list of workbins, list of files, folder structure, etc.

Class: ***UpdateDetector***

- Detects changes or updates in the folder structures.

Class: ***LocalUpdator***

- Downloads and manage new files.

Storage

Manages necessary information and transfers them between memory and local disk.
Main classes:

Class: ***StorageFacade***

- Acts as an interface between storage and higher components
- Provides high level operations to retrieve/save metadata, profiles' information, write/save log

Class: ***MetaDataHolder***

- Stores and provides method to manage metadata
- Implements Singleton pattern

Class: ***ProfileHolder***

- Contains information of one profile, with support of profileHolder containing methods to retrieve information of one profile
- Implements Singleton pattern

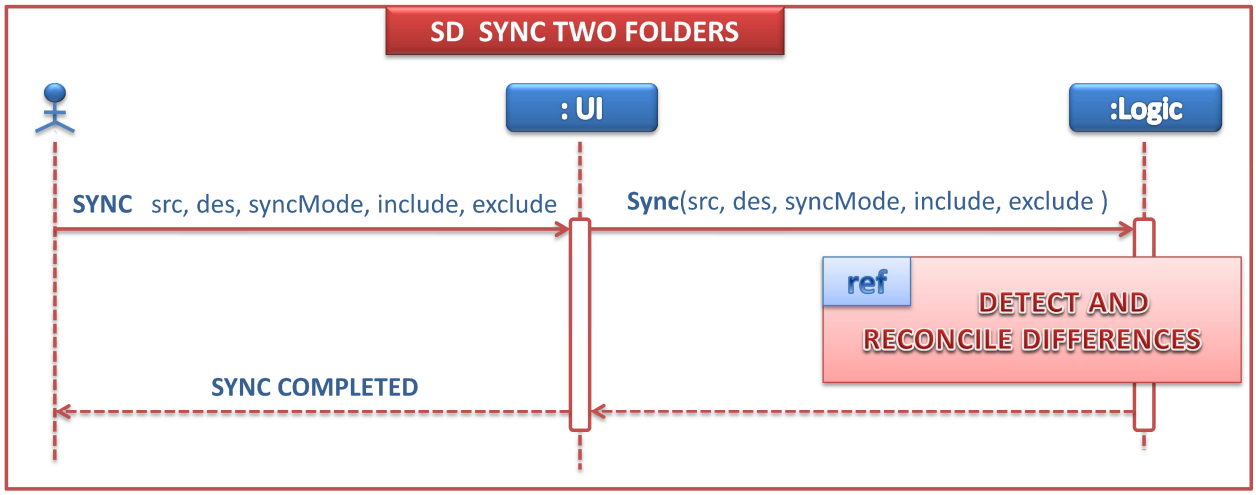
Class: ***LoggerHolder***

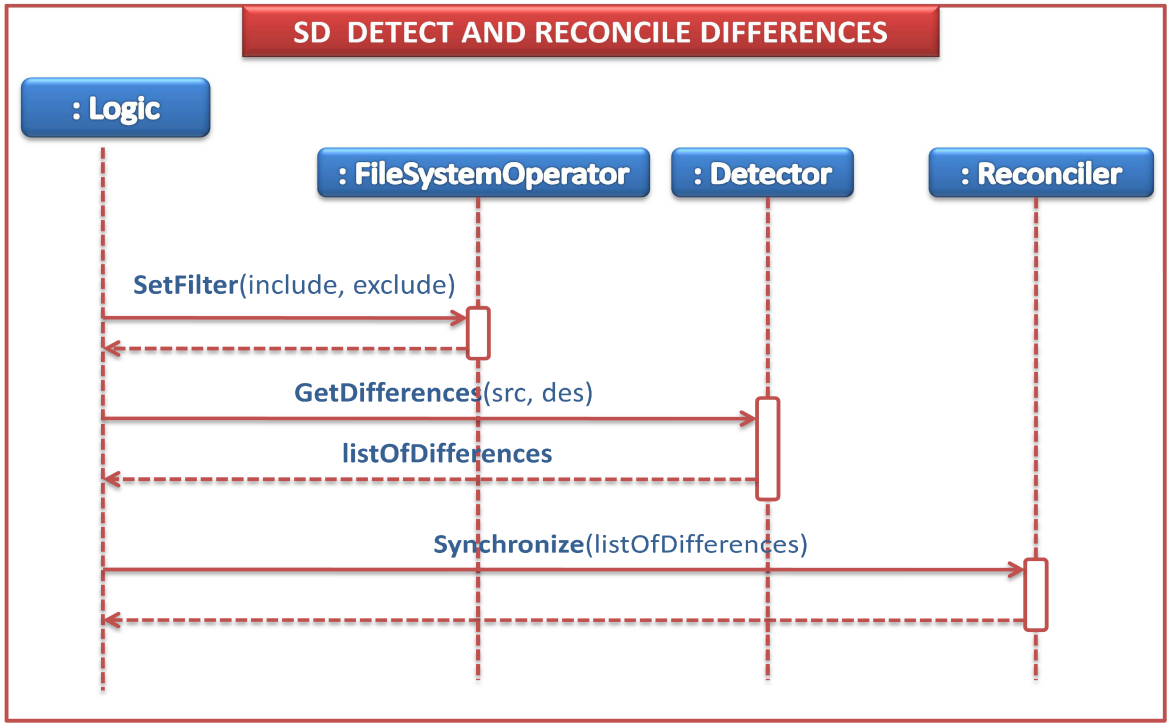
- Logs and stores necessary information
- Implements Singleton pattern

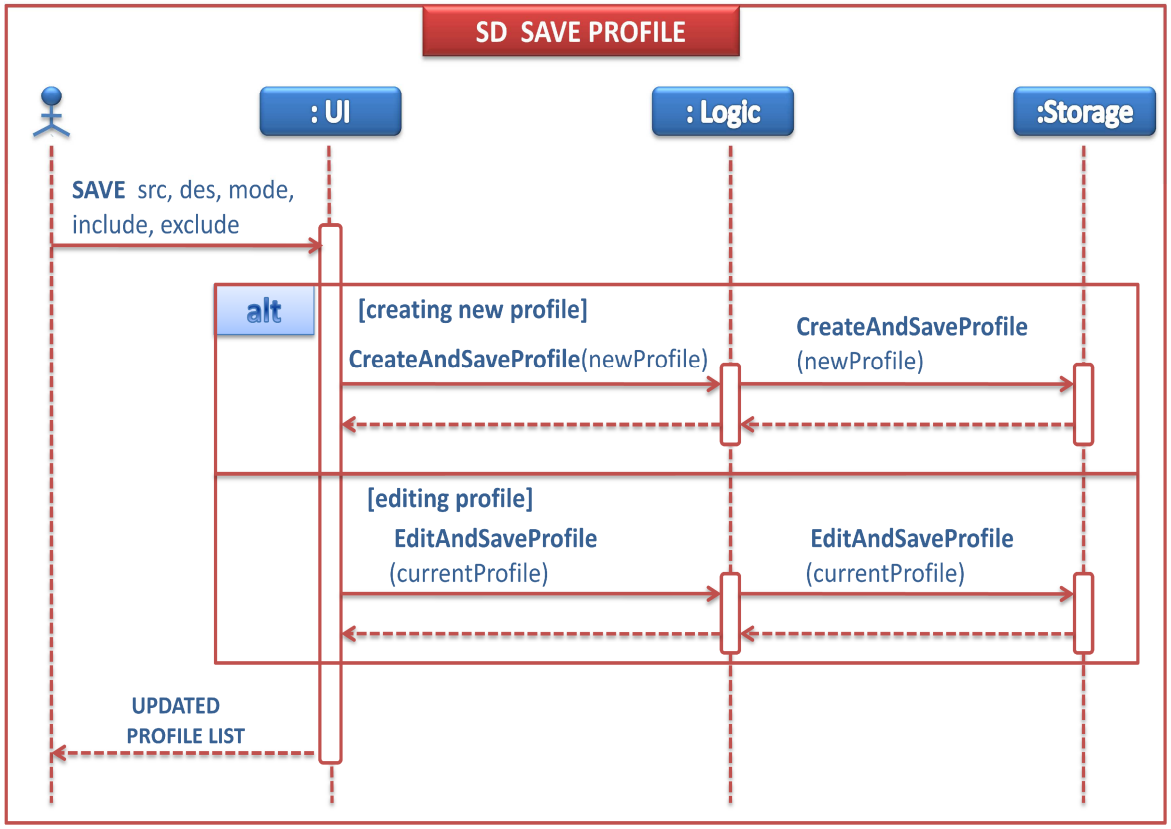
Class: ***SettingHolder***

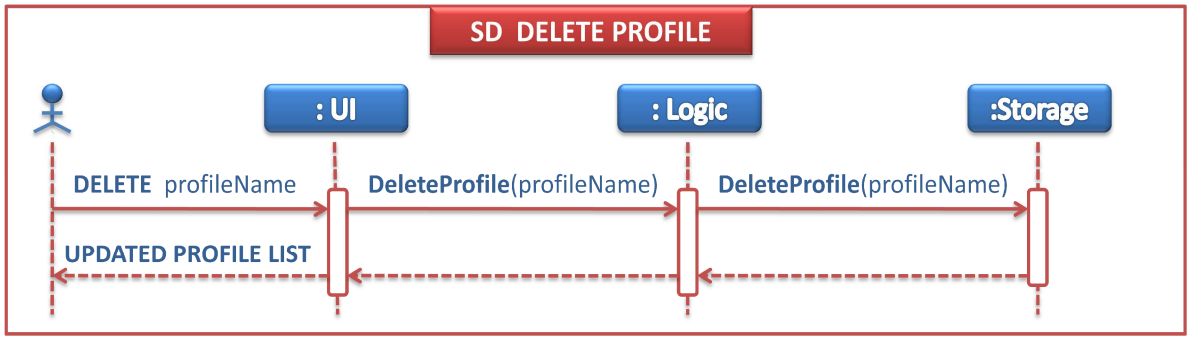
- Stores and provide methods to manage program settings
- Implements Singleton pattern

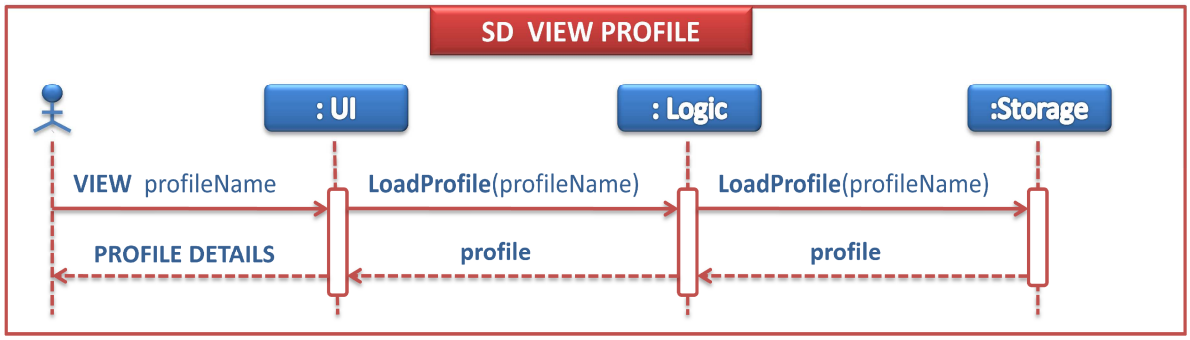
SEQUENCE DIAGRAMS






SD SAVE PROFILE
: UI
: Logic
:Storage
SAVE src, des, mode, include, exclude
alt
[creating new profile]
CreateAndSaveProfile(newProfile)
CreateAndSaveProfile (newProfile)
[editing profile]
EditAndSaveProfile (currentProfile)
EditAndSaveProfile (currentProfile)
UPDATED PROFILE LIST


SD DELETE PROFILE
: UI
: Logic
:Storage
DELETE profileName
DeleteProfile(profileName)
DeleteProfile(profileName)
UPDATED PROFILE LIST


SD VIEW PROFILE
: UI
: Logic
:Storage
VIEW profileName
LoadProfile(profileName)
LoadProfile(profileName)
PROFILE DETAILS
profile
profile

IMPORTANT APIS

CLASS: LOGICFACADE

Operation: SaveCurrentProfile(): bool

Description: Set the information for the current profile
Parameters: none
Preconditions: none
Postcondition: current profile is saved

Operation: SaveProfile(Profile profile): void

Description: Save a profile to profile storage file
Parameters: profile object
Preconditions: none
Postcondition: the profile is saved

Operation: SetCurrentProfile(Profile profile) void

Description: Set the information for the current profile
Parameters: profile object
Preconditions: profile is not null
Postcondition: current profile is updated with information from the new profile

Operation: GetCurrentProfile(): Profile

Description: Get the information for the current profile
Parameters: none
Preconditions: none
Postcondition: current profile is returned

Operation: ResetCurrentProfile(): void

Description: Set the default information for the current profile
Parameters: none
Preconditions: none
Postcondition: current profile is reset to default values

Operation: GetProfileNameList(): List<string>

Description: Retrieve the profile name list
Parameters: none
Preconditions: none
Postcondition: profile name list is returned

Operation: LoadProfile(string profilename): Profile

Description: Load a profile from profile list

Parameters: name of the profile
Preconditions: none
Postcondition: the requested is returned

Operation: DeleteProfile(string profilename): bool

Description: Delete a profile from profile storage file
Parameters: name of the profile
Preconditions: none
Postcondition: profile is removed

Operation: ExecuteProfile(string profilename): void

Description: Execute the profile specified by its name
Parameters: name of the profile
Preconditions: profile exists
Postcondition: profile is executed

Operation: ExecuteCurrentProfile(): void

Description: Execute the current profile
Parameters: none
Preconditions: none
Postcondition: current profile is executed

Operation: Sync(Profile profile): void

Description: Sync a profile
Parameters: profile object
Preconditions: profile exists
Postcondition: profile is executed

Operation: RetrieveListOfDifferences(): List<FilePair>

Description: Retrieve the list of differences between two synchronized folders.
Parameters: none
Preconditions: none
Postcondition: list of file pairs is returned

Operation: PrintDifferences(): void

Description: Print out the differences between two synchronized folders
Parameters: none
Preconditions: none
Postcondition: list of file pairs is printed

Operation: InitializeStorage(string src_des): void

Description: Inform Storage of the source and destination roots

Parameters: the source_destination string
Preconditions: none
Postcondition: All metadata information are set up

Operation: FinalizeStorage(): void

Description: Inform Storage of completed synchronization
Parameters: none
Preconditions: none
Postcondition: All metadata information and logs are writen back to files

Operation: InitializeLogic(): void

Description: Initialize Detector and Reconciler
Parameters: none
Preconditions: none
Postcondition: detector and reconciler are set up

Operation: RetrieveContainingProfiles(string folderpath): List<string>

Description: Load the list of all profiles containing specific folder path
Parameters: absolute path of the folder
Preconditions: none
Postcondition: the name list of profiles containing the folder path is returned

Operation: InitializeStorage(string src_des): void

Description: Perform initialization of metadata storage
Parameters: the source_destination string
Preconditions: none
Postcondition: metadata are prepared and loaded into RAM

Operation: FinalizeStorage(): void

Description: Perform finalization of metadata storage
Parameters: none
Preconditions: none
Postcondition: metadata and logs are writen back to files

Operation: LoadProfile(string profileName): Profile

Description: Load a profile from profile storage file
Parameters: name of the profile
Preconditions: none
Postcondition: requested profile is returned if exists

Operation: LoadDefaultProfile(): Profile

Description: Load the default profile
Parameters: none
Preconditions: none
Postcondition: default profile is returned

Operation: LoadProfileNameList(): List<string>

Description: Load the list of all profiles from the profile storage file
Parameters: none
Preconditions: none
Postcondition: profile name list is returned

Operation: LoadContainingProfiles(string folderPath): List<string>

Description: Load the list of all profiles containing specific folder path
Parameters: absolute path of the folder
Preconditions: none
Postcondition: the name list of profiles containing the folder path is returned

Operation: SaveProfile(Profile profile): Profile

Description: Add a new profile entry to profile storage file
Parameters: profile object
Preconditions: none
Postcondition: profile is saved

Operation: RemoveProfile(string profileName): void

Description: Remove a profile from the profile storage file
Parameters: name of the profile
Preconditions: none
Postcondition: profile is removed

Operation: GetLogger(string name): Logger

Description: Return the logger instance with specified name
Parameters: name of the logger instance
Preconditions: none
Postcondition: requested logger object is returned

Operation: SaveLogger(): void

Description: Save all logger instances' information to storage
Parameters: none
Preconditions: none
Postcondition: all logger intances are writen back to files

Operation: SaveMetaData(string filesystemPath, string metadata): void

Description: Store the metadata of a filesystem path to the metadata hashtable
Parameters: absolute path of the file and its metadata to save
Preconditions: none
Postcondition: metadata of the file/folder is saved

Operation: LoadMetaData(string filesystemPath): string

Description: Load the metadata of a filesystem path from the metadata hashtable
Parameters: absolute path of the file to load
Preconditions: none
Postcondition: metadata of the file/folder is returned if exists

Operation: RemoveMetaData(string filesystemPath): void

Description: Remove the metadata of a filesystem path from the metadata hashtable
Parameters: absolute path of the file to load
Preconditions: none
Postcondition: metadata of the file/folder is removed

CLASS: STORAGEFACADE

Operation: InitializeStorage(string src_des): void

Description: Perform initialization of metadata storage
Parameters: the source_destination string
Preconditions: none
Postcondition: metadata are prepared and loaded into RAM

Operation: FinalizeStorage(): void

Description: Perform finalization of metadata storage
Parameters: none
Preconditions: none
Postcondition: metadata and logs are writen back to files

Operation: LoadProfile(string profileName): Profile

Description: Load a profile from profile storage file
Parameters: name of the profile
Preconditions: none
Postcondition: requested profile is returned if exists

Operation: LoadDefaultProfile(): Profile

Description: Load the default profile
Parameters: none

Preconditions: none
Postcondition: default profile is returned

Operation: LoadProfileNameList(): List<string>

Description: Load the list of all profiles from the profile storage file
Parameters: none
Preconditions: none
Postcondition: profile name list is returned

Operation: LoadContainingProfiles(string folderPath): List<string>

Description: Load the list of all profiles containing specific folder path
Parameters: absolute path of the folder
Preconditions: none
Postcondition: the name list of profiles containing the folder path is returned

Operation: SaveProfile(Profile profile): Profile

Description: Add a new profile entry to profile storage file
Parameters: profile object
Preconditions: none
Postcondition: profile is saved

Operation: RemoveProfile(string profileName): void

Description: Remove a profile from the profile storage file
Parameters: name of the profile
Preconditions: none
Postcondition: profile is removed

Operation: LoadLogger(string name): Logger

Description: Return the logger instance with specified name
Parameters: name of the logger instance
Preconditions: none
Postcondition: requested logger object is returned

Operation: SaveLogger(): void

Description: Save all logger instances' information to storage
Parameters: none
Preconditions: none
Postcondition: all logger intances are writen back to files

Operation: SaveMetaData(string filesystemPath, string metadata): void

Description: Store the metadata of a filesystem path to the metadata hashtable

Parameters: absolute path of the file and its metadata to save
Preconditions: none
Postcondition: metadata of the file/folder is saved

Operation: SaveSetting(string property, string value): void

Description: Store a setting to setting hashtable then write back to setting file
Parameters: a setting property and its value
Preconditions: none
Postcondition: setting is updated

Operation: LoadSetting(string property): string

Description: Load the value of a setting property from the metadata hashtable
Parameters: a setting property
Preconditions: none
Postcondition: value of the setting property is returned

Operation: InitializeMetaData(string srcFolder, string desFolder): void

Description: Initialize metadata at the beginning of a sync job
Parameters: absolute source path and destination path of a sync job
Preconditions: none
Postcondition: metadata for specified sync job is loaded to metadata hashtable

Operation: FinalizeMetaData(string srcFolder, string desFolder): void

Description: Finalize metadata at the end of a sync job
Parameters: absolute path of the file to load
Preconditions: none
Postcondition: metadata for specified sync job is written back to metadata file

Operation: LoadMetaData(string filesystemPath): string

Description: Load the metadata of a filesystem path from the metadata hashtable
Parameters: absolute path of the file to load
Preconditions: none
Postcondition: metadata of the file/folder is returned if exists

Operation: RemoveMetaData(string filesystemPath): void

Description: Remove the metadata of a filesystem path from the metadata hashtable
Parameters: absolute path of the file to load
Preconditions: none
Postcondition: metadata of the file/folder is removed

4. Supplementary

What are covered in this section:

- ✓ More details about right-click feature implementation
- ✓ More details about IVLE feature
- ✓ More details about Synchronization Logic
- ✓ More details about Filter implementation

MORE DETAILS ABOUT RIGHT-CLICK FEATURE IMPLEMENTATION

We use **Context Menu Handler technology** to handle user right-click event occurring on folders. Using this way, we easily retrieve the list of multiple folders which have been selected without having to maintain a single instance of the program using Mutex, which is not flexible and convenient in this case. After retrieving the list of selected folders, all folder paths are combined into one string and passed as an argument for our main program to handle. Since we have **Smart Sync** and **Sync With…**, we also passed **--SyncWith** and **--SmartSync** as another argument for the main program to realize each case. These argument also help the program to tell apart when user right-click to sync from when user run the main application (no parameter is passed).

The main purpose of **Context Menu Handler** is to dynamically retrieve information of folders selected and pass them as arguments for other programs to handle. Therefore, we created a new project **nets-contextmenuhandler** to handle specifically this task and built a nets-**contextmenuhandler.dll** to embed with our main application. This helps us to keep our main program unaware of **Context Menu Handler**. Also, with the help of **RegAsm of Win32**, we can allow user to register/unregister NETS entries. All we need to do is calling **RegAsm** on nets-**contextmenuhandler.dll** with parameter **/register** or **/unregister**.

nets-contextmenuhandler project provides class **ContextMenuManager** to handler the events mentioned above with the help of classes from **Win32**.

MORE DETAILS ABOUT IVLE FEATURE

Currently, IVLE is handled by four classes: **IVLEHandler**, **IVLEParser**, **LocalUpdater** and **UpdateDetector**. In the future, there will be one component that the above four classes that the above four can use: **InfoManager**. The InfoManager objects manage offline data including user information and the parsed data so that we can speed up the process.

Beneath the main functional classes, we have several helper and data structures objects that all of the above depends on.

- **ExtendedWebClient**: this is a 'customized version' of WebClient class to handle the network I/O. (It is not actually derived from WebClient.)
  This client supports POST Http request and Cookies.
- **StringFormatPattern:** to parse the formatted strings.
- **XmlTool:** manage Xml element creation and query.
- **IvleWorkbin, IvleWorkbinFolder, IvleWorkbinFile:** data structure classes for Ivle workbin, folder and file respectively.

MORE DETAILS ABOUT SYNCHRONIZATION LOGIC

Our synchronization logic (**Detector** and **Reconciler**) is based on **status** of each file from the last synchronization. That status is detected by **Detector** and **Reconciler** performs corresponding actions for each status. Our implementation has 5 **statuses** for a file: **No-change** (the file is not changed since the last sync), **New** (the file did not exist in the last sync), **Updated** (the content of the file is modified since the last sync), **Deleted** (the file existed in the last sync but now has been deleted), **Not-exist** (the file did not exist in the last sync and does not exist now).

We consider the scenario when two folders on two computers are synced via the USB medium. If we use one-way synchronization, the destination folder will be the same as source folder after they are synced, which is simple. For two-way synchronization, if there are **New** files in one of the two folders, we simply copy them to the other folder. If a file's status in one folder is **Updated/Deleted** and **No-change** in other folder, we updated/delete in other folder accordingly.

What if a file's status is both **Updated** in two folders? Or **Updated** in one folder and **Deleted** in other folder? In the case when the file is both **Updated,** we choose the one **modified later** to overwrite the other one. In the case when the file is **Updated** in one folder and **Deleted** in other folder, we restore the **Deleted** one for safety.

MORE DETAILS ABOUT FILTER IMPLEMENTATION

NETS allows user to include and exclude certain files in the synchronization by specifying patterns of their names. Exclude always has higher priority than include. If a file is both included and excluded, it will be excluded.

Only wildcard * is supported in file name patterns to specify no character, one or more character. For example, if we user excludes *cs3215*.pdf, the following files will be excluded: cs3215Lecture01.pdf, cs3215.pdf, cs3215.pdf.doc, etc.

File Filter is handled by class **FilterPattern**. It provides one public static method **IsExcluded(filename)**. A file is said to be excluded if its name matches **Exclude Pattern** or its name does not match **Include Pattern**. **Exclude Pattern** is empty means no file is excluded. **Include Pattern** is empty means all files are included.