Leong Shengmin - Project Portfolio

PROJECT: WishBook

Overview

WishBook (WB) is a piggy bank in a digital app form, made for people who need a user-friendly solution for keeping track of their disposable income. Not only does WB keep records of your savings, it also allows users to set items as goals for users to work towards, serving as a way to cultivate the habit of saving. WB is made for people who have material wants in life, but are uncertain of purchasing said wants, by helping them to reserve disposable income so that they know they can spend on their wants.

Below is a mockup of `WishBook`

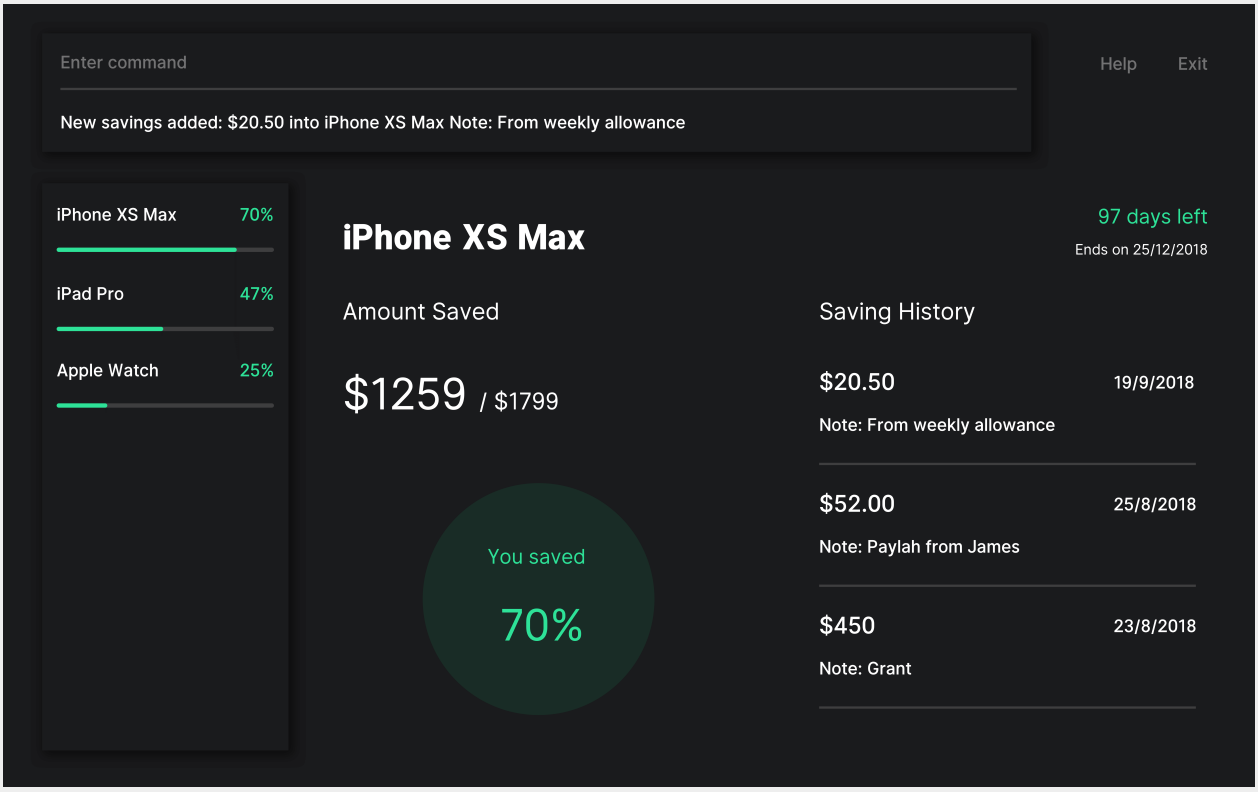

Fig 1: Mockup of WishBook

Summary of contributions

Major enhancement: added the ability to view savings history

Below is a screenshot of the savings history for a `Wish`

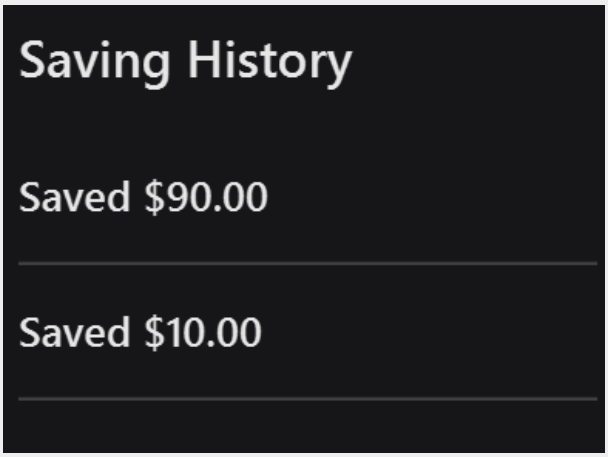

Fig 2: Savings History of a Wish

- What it does: Shows a history of savings you have allocated for any particular wish in the wishbook, from newest to oldest.
- Justification: This feature improves the product significantly because the automatic book-keeping of savings for each Wish allows the user to easily and clearly keep track of his or her savings over a length of time.

  This feature also allows for extensions since the data collected in terms of amount saved as well as frequency of saving allows for precious insights to be drawn about the saving habits as well as saving priorities of the user.

- Highlights: This enhancement affects existing commands and commands to be added in future.

  The implementation of this feature involves many different components and is intrinsically tied to the state of the `WishBook`, which changes when the user executes an action-based command such as the `Add`, `Save`, `Edit` command. <<<

Areas involved:

- Persistent Storage
  - Facilitate reading and writing of savings histories of all wishes to hard disk. This data will also be saved to the hard disk when triggered by a change in the state of the wishbook.
- Dynamic updating of savings history
  - In order for savings histories of each `Wish` to be updated dynamically, the implementation follows an event-driven model. When an action-based command is executed, subscribed classes will be notified of the change in state of the `WishBook`.
  - The implementation of tracking the savings history of each `Wish` also needs to follow a `VersionedModel` that responds when notified by the change in `WishBook` state.
- Testing
  - Unit and Integrated testing for the code contributed.

- Credits:
  - Persistent storage
    - Savings history saved as `xml` format using the `JAXB` framework.
  - Dynamic updating of savings history
    - Follows event-driven model of `AddressBook-L4`
    - `VersionedModel` implementation inspired by `undo`, `redo` functionality in `AddressBook-L4`.
  - Testing
    - `JUnit` testing framework adopted.

Minor enhancement: updated the `Help Window` display.

The `Help Window` is a pop-up window that appears when the `Help` command is executed. It is a guide to aid the user navigate the `WishBook`.

The image below shows a screenshot of the `Help Window`.

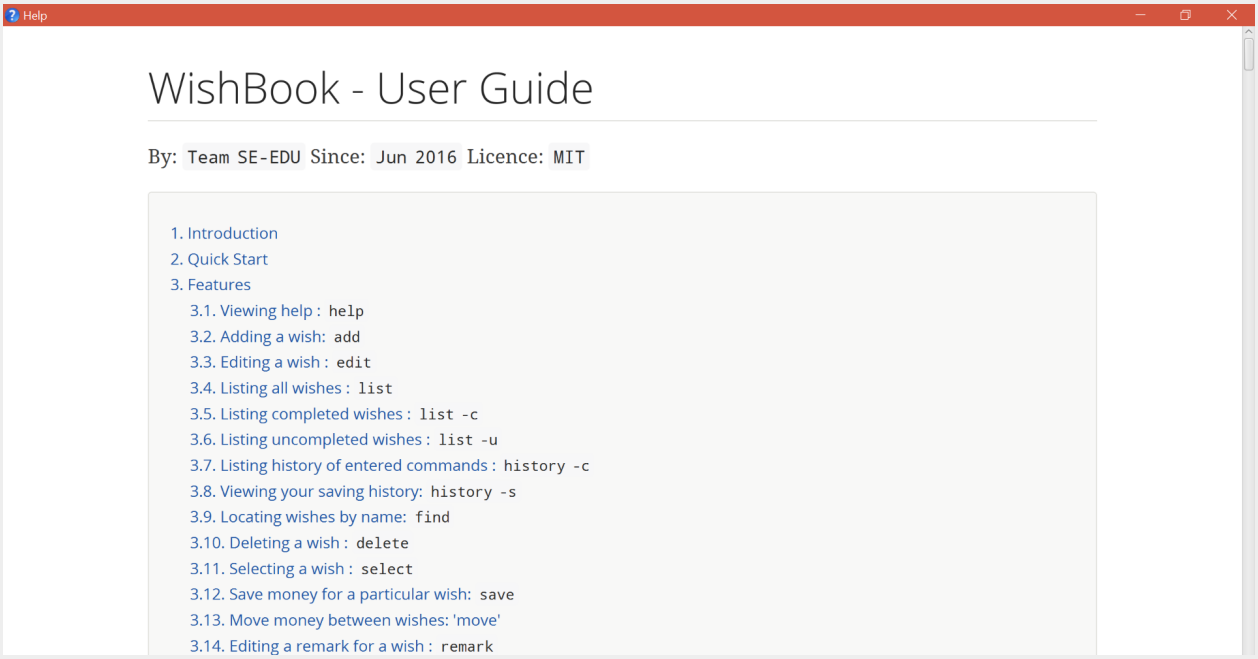

Fig 3: Help Window

Code contributed:

The following links contain code snippets of code contributed

Other contributions:

- Documentation:
- User Guide:
  - Wrote the first version of the user guide containing all features in the first version of `WishBook`. (#14
  - Added content for `History -s` command. #15
- Developer Guide:
  - Added content for `History -s` command. #16

- Diagrams
  - Refactored undo, redo UML diagrams to reflect current version of WishBook.

The following images show the undo, redo UML diagrams:

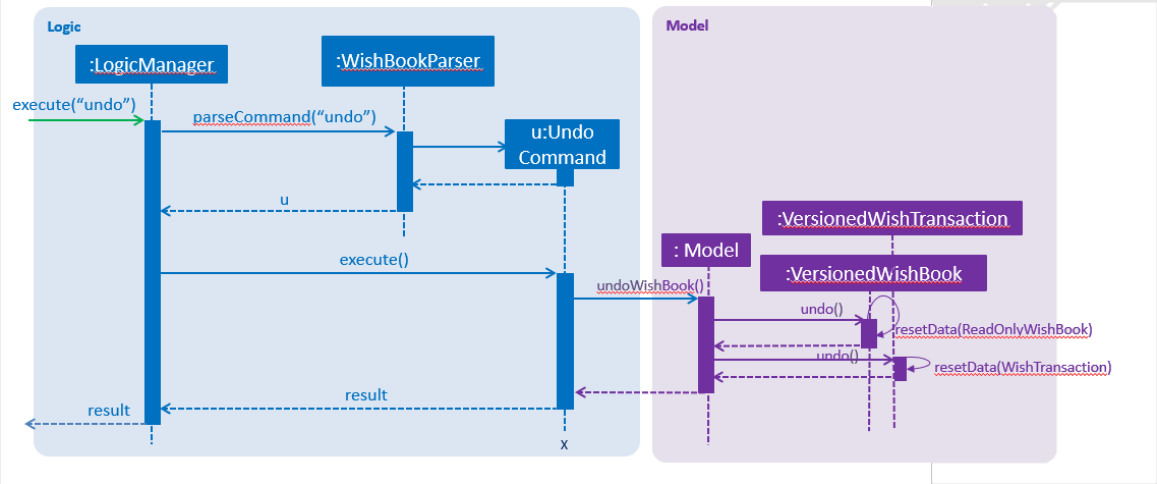

Fig 5.1: Undo Redo Sequence Diagram

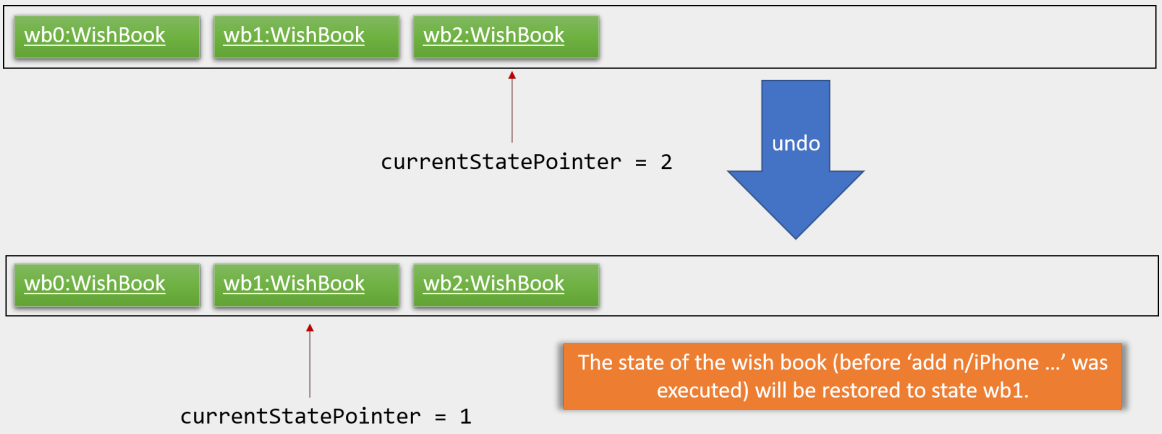

Fig 5.2.2 Undo Redo State Diagram after `Undo` *command executed*

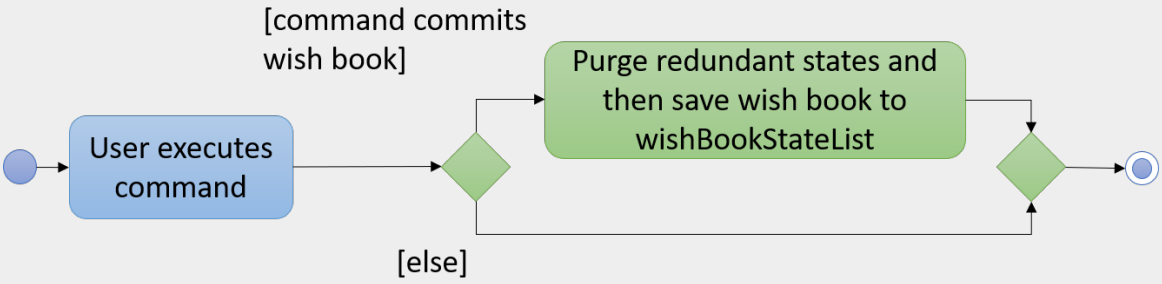

Fig 5.3 Undo Redo Activity Diagram

- Updated the Model class diagram.

  The following image shows the updated Model Class Diagram.

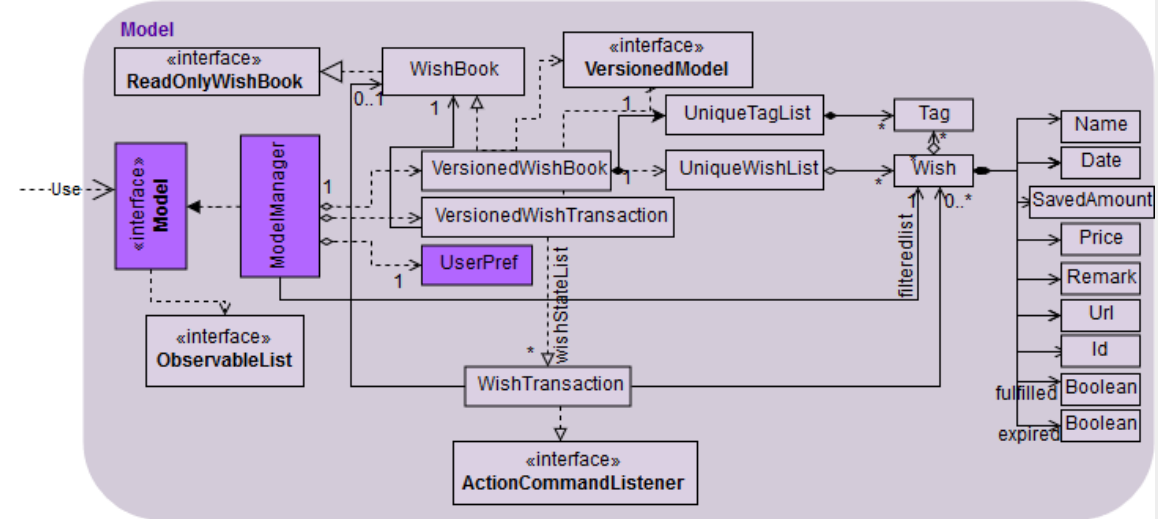

Fig 6: Model Class Diagram

- Updated the Storage class diagram.

  The following image shows the updated Storage Class Diagram.

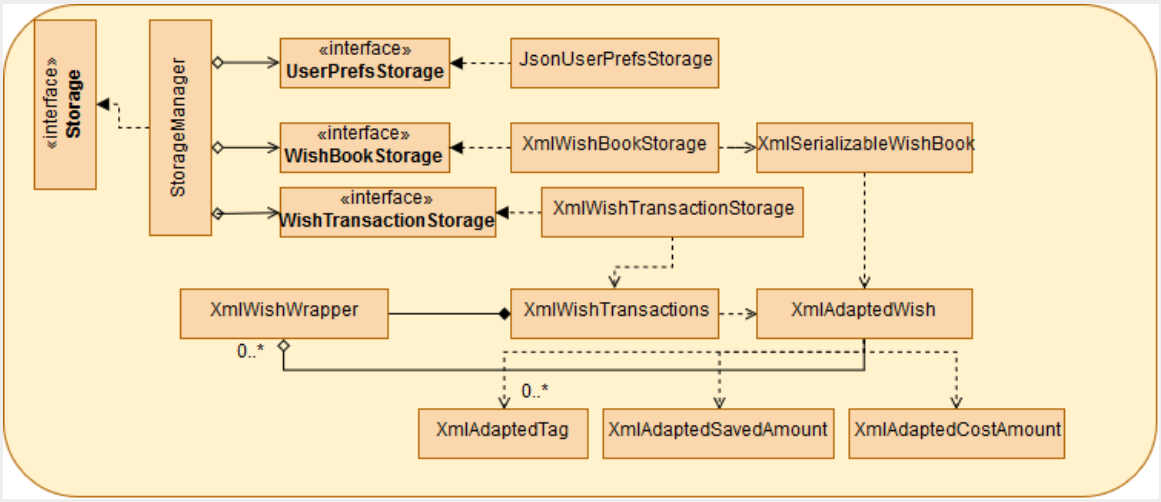

Fig 7: Storage Class Diagram

- Community:
  - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42
  - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
  - Helped to integrate savings history feature with other components worked on by team members (1, 2)
  - Resolved issues raised by team members (1, 2) <<<

- Tools:
    - Configured `RepoSense`, a Contribution Analysis Tool for the team repo (#42)

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Viewing your saving history: `history -s`

Shows a history of savings you have allocated, from newest to oldest.
Format: `history -s`

NOTE	Only history of wishes which currently exist in the `WishBook` will be stored. (i.e. wishes that have been deleted will no longer be tracked.)

Proposed enhancement for V2.0

Encrypting data files `[coming in v2.0]`

{explain how the user can enable/disable data encryption}

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Savings History feature

Capturing the state of `WishTransaction`

The current state of the savings history of the `WishBook` is captured by `VersionedWishTransaction`. `VersionedWishTransaction` extends `WishTransaction` and has an undo/redo history, similar to the implementation of the Undo/Redo feature, and is stored internally as a `wishStateList` and `currentStatePointer`. Additionally, it implements `VersionedModel` and so contains the implementation of the following operations:

- `VersionedWishTransaction#commit()` — Saves the current wish transaction state in its history.
- `VersionedWishTransaction#undo()` — Restores the previous wish transaction state from its history.
- `VersionedWishTransaction#redo()` — Restores a previously undone wish transaction state from its history.

These operations are exposed in the `Model` interface as `Model#commitWishBook()`, `Model#undoWishBook()` and `Model#redoWishBook()` respectively.

Capturing the state of each `Wish`

`WishTransaction` keeps track of the state of all wishes in `WishBook` via a `wishMap` which maps the unique ID of a `Wish` to a list of `Wish` states. `WishTransaction` implements `ActionCommandListener` such that any state changing command performed to a `Wish` or the `WishBook` such as `AddCommand()`, `EditCommand()`, `SaveCommand()`, etc will result in the `WishMap` being updated accordingly in `WishTransaction`.

Persistent storage

`VersionedWishTransaction`, `WishTransaction` can be easily converted to and from xml using `XmlWishTransactions`. `XmlWishTransactions` is saved as an xml file when the user explicitly closes the window, thereby invoking `MainApp#stop()` which saves the current state of `VersionedWishTransaction` in the `wishStateList` to hard disk.

If the user's command triggers a change in the state of the `WishBook`, a `WishBookChangedEvent` will be raised, causing the subscribed `StorageManager` to respond by saving both the current state of the `WishBook` and `WishTransaction` to disk.

Given below is an example usage scenario and how the savings history mechanism behaves at each step.

Step 1. The user launches the application. The default file path storing the previous state of the `WishTransaction` will be retrieved, unless otherwise specified by the user, and the contents from the xml file will be parsed and converted into a `WishTransaction` object via the `XmlWishTransactions` object. If the file at the specified location is behind the current state of the `WishBook`, content of the `WishTransaction` will be overwritten by the `WishBook`.

NOTE	The `wishStateList` starts off with the initial state of the `WishTransaction` as the first item in the list.

Step 2. The user executes `add n/iPhone ⋯` to add a new wish. The `add` command calls `Model#commitWishBook()`, causing the current state of the modified wish transaction state to be saved into `wishStateList`. As this is a command that changes the state of the `WishBook`, `Model#addWish()` will call `VersionedWishTransaction#addWish()` to add a new wish to the `WishMap`.

- If a command fails its execution, it will not call `Model#commitWishBook()`, so the wish transaction state will not be saved into the `wishStateList`.
- If the `WishMap` contains an identical wish (such is identified by `Wish#isSameWish()`), then the call to add this wish will fail. As such, the wish will not be added to the `WishMap` or the `WishBook`.

Step 3. The user now decides that adding the wish was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoWishBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous wish transaction state, and restores the wish transaction to that state.

NOTE	If the `currentStatePointer` is at index 0, pointing to the initial wish transaction state, then there are no previous wish transaction states to restore. The `undo` command uses `Model#canUndoWishBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The `redo` command does the opposite — it calls `Model#redoWishBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the wish transaction to that state.

NOTE	If the `currentStatePointer` is at index `wishStateList.size() - 1`, pointing to the latest wish transaction state, then there are no undone wish transaction states to restore. The `redo` command uses `Model#canRedoWishBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 4. The user then decides to execute the command `list`. Commands that do not modify the state of the `WishBook`, such as `list`, will usually not call `Model#commitWishBook()`, `Model#undoWishBook()` or `Model#redoWishBook()`. Thus, the `wishBookStateList` remains unchanged.

Step 5. The user finally exits the app by clicking on the close button. The most recent state of the `WishTransaction` will be converted into xml format via the the `XmlWishTransactions` object and be saved into the same file path it was first retrieved from.

NOTE	If there was some error saving the current state of the `WishTransaction` to the specified file path in hard disk, an exception will be thrown and a warning will be shown to the user. The current state of the `WishTransaction` object will not be saved to hard disk.

Proposed enhancement for V2.0

Savings History feature

Capturing the state of `WishTransaction`

The current state of the savings history of the `WishBook` is captured by `VersionedWishTransaction`. `VersionedWishTransaction` extends `WishTransaction` and has an undo/redo history, similar to the implementation of the Undo/Redo feature, and is stored internally as a `wishStateList` and `currentStatePointer`. Additionally, it implements `VersionedModel` and so contains the implementation of the following operations:

- `VersionedWishTransaction#commit()` — Saves the current wish transaction state in its history.
- `VersionedWishTransaction#undo()` — Restores the previous wish transaction state from its history.
- `VersionedWishTransaction#redo()` — Restores a previously undone wish transaction state from its history.

These operations are exposed in the `Model` interface as `Model#commitWishBook()`, `Model#undoWishBook()` and `Model#redoWishBook()` respectively.

Capturing the state of each `Wish`

`WishTransaction` keeps track of the state of all wishes in `WishBook` via a `wishMap` which maps the unique ID of a `Wish` to a list of `Wish` states. `WishTransaction` implements `ActionCommandListener` such that any state changing command performed to a `Wish` or the `WishBook` such as `AddCommand()`, `EditCommand()`, `SaveCommand()`, etc will result in the `WishMap` being updated accordingly in `WishTransaction`.

Persistent storage

`VersionedWishTransaction`, `WishTransaction` can be easily converted to and from xml using `XmlWishTransactions`. `XmlWishTransactions` is saved as an xml file when the user explicitly closes the window, thereby invoking `MainApp#stop()` which saves the current state of `VersionedWishTransaction` in the `wishStateList` to hard disk.

If the user's command triggers a change in the state of the `WishBook`, a `WishBookChangedEvent` will be raised, causing the subscribed `StorageManager` to respond by saving both the current state of the `WishBook` and `WishTransaction` to disk.

Given below is an example usage scenario and how the savings history mechanism behaves at each step.

Step 1. The user launches the application. The default file path storing the previous state of the `WishTransaction` will be retrieved, unless otherwise specified by the user, and the contents from the xml file will be parsed and converted into a `WishTransaction` object via the `XmlWishTransactions` object. If the file at the specified location is behind the current state of the `WishBook`, content of the `WishTransaction` will be overwritten by the `WishBook`.

NOTE	The `wishStateList` starts off with the initial state of the `WishTransaction` as the first item in the list.

Step 2. The user executes `add n/iPhone ⋯` to add a new wish. The `add` command calls `Model#commitWishBook()`, causing the current state of the modified wish transaction state to be saved into `wishStateList`. As this is a command that changes the state of the `WishBook`, `Model#addWish()` will call `VersionedWishTransaction#addWish()` to add a new wish to the `WishMap`.

- If a command fails its execution, it will not call `Model#commitWishBook()`, so the wish transaction state will not be saved into the `wishStateList`.
- If the `WishMap` contains an identical wish (such is identified by `Wish#isSameWish()`), then the call to add this wish will fail. As such, the wish will not be added to the `WishMap` or the `WishBook`.

Step 3. The user now decides that adding the wish was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoWishBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous wish transaction state, and restores the wish transaction to that state.

NOTE	If the `currentStatePointer` is at index 0, pointing to the initial wish transaction state, then there are no previous wish transaction states to restore. The `undo` command uses `Model#canUndoWishBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The `redo` command does the opposite — it calls `Model#redoWishBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the wish transaction to that state.

NOTE	If the `currentStatePointer` is at index `wishStateList.size() - 1`, pointing to the latest wish transaction state, then there are no undone wish transaction states to restore. The `redo` command uses `Model#canRedoWishBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 4. The user then decides to execute the command `list`. Commands that do not modify the state of the `WishBook`, such as `list`, will usually not call `Model#commitWishBook()`, `Model#undoWishBook()` or `Model#redoWishBook()`. Thus, the `wishBookStateList` remains unchanged.

Step 5. The user finally exits the app by clicking on the close button. The most recent state of the `WishTransaction` will be converted into xml format via the the `XmlWishTransactions` object and be saved into the same file path it was first retrieved from.

NOTE	If there was some error saving the current state of the `WishTransaction` to the specified file path in hard disk, an exception will be thrown and a warning will be shown to the user. The current state of the `WishTransaction` object will not be saved to hard disk.