Tee Hao Wei - Project Portfolio

PROJECT: Budget Buddy

Overview

Budget Buddy is a desktop transaction manager application that allows users to track their expenses. The main method of input by the user is using a command-line interface (CLI), but the application presents information using a graphical interface.

Budget Buddy is based on the AddressBook-Level3 application by SE-EDU, and currently consists of about 15 thousand lines of code.

Summary of contributions

- **Major enhancement**: added a scripting engine and library
    - What: This feature provides the user the ability to evaluate arbitrary scripts which have full access to the data and user interface of the application, and save these scripts in a library for convenient reuse.
    - Why: This feature allows the user to perform complex operations that may be inconvenient or impossible to do through the CLI, and extend the application with new commands and features on their own, without having to modify the application's source code directly.
    - Highlights: This feature required thought to provide a ergonomic interface to allow scripts to do common operations easily, without restricting scripts from doing more complex operations.
    - Credits: This feature is implemented using the Nashorn JavaScript engine, bundled with Java 11.
- **Major enhancement**: added the ability to import and export transactions
    - What: This feature provides the user the ability to import and export transactions.
    - Why: This feature allows the user to import transactions exported from their bank transaction history and export transactions for further manipulation in an external program, such as Google Sheets or Excel.
    - Highlights: This feature required retrieving exports from various banks and writing parsers for each of their formats.
- **Minor enhancement**: added a command alias feature to allow users to create shorthands
- **Code contributed**: [Functional code] [Test code]
- **Other contributions**:
    - Project management:
        - Managed releases `v1.1` to `v1.3` (4 releases) on GitHub

- Enhancements to existing features:
    - Updated the GUI color scheme; see PRs #33, #34
    - Wrote additional tests for existing features to increase coverage from 88% to 92%; see PRs #36, #38
- Documentation:
    - Did cosmetic tweaks to existing contents of the User Guide; see PR #14
- Community:
    - Reviewed PRs (with non-trivial comments) #12, #32, #19, #42
    - Rebased PRs #nn, #nn, #nn
- Tools:
    - Integrated Netlify, Coveralls and Codacy to the team repository

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.

Import and export: `import`, `export`

Import a file: `import`

Imports transactions from the file at the given path.

Format: `import [f/<format>] [p/<file path>]`

- The file path can be relative to where you launched Budget Buddy from, or absolute. If the path is omitted, a file browser is opened for you to select the file.
- Format is one of dbs (DBS Bank/POSB), ocbc (OCBC Bank), sc (Standard Chartered), csv (generic comma-separated values file). If the format is omitted, automatic detection is attempted.

Export transactions: `export`

Exports all transactions to the given path.

Format: `export [p/<file path>]`

- The file path can be relative to where you launched Budget Buddy from, or absolute. If the path is omitted, a file browser is opened for you to select where to save the file.
- The file is a comma-separated values file.

Scripting: `script`

Evaluate a script: `script eval`

Evaluates a script and displays the result.

Format: `script eval <script>`

- The scripting language is JavaScript (specifically, ECMAScript 5.1).

Add a stored script: `script add`

Stores a script for future invocation.

Format: `script add <script name> [p/<file path> | s/<script>]`

- The script name may contain only alphanumeric characters, underscores, and dashes.
- If neither a file path nor the script code is given, a file browser is opened for you to select the script file.

WARNING The script is not checked for correctness before it is stored. Any syntax errors will be reported only when the script is run.

Delete a stored script: `script delete`

Deletes a previously-stored script.

Format: `script delete <script name>`

Run a stored script: `script run`

Runs a previously-stored script.

Format: `script run <script name> [<argument>]`

- The argument is the rest of the command line after the script name, and is passed to the script as a single string.

Aliases: `alias`

Add an alias: `alias add`

Adds an alias.

Format: `alias add <alias name> c/<alias replacement>`

- When executed, the alias name will be replaced by the replacement, and the resulting command line executed.
  - For example, suppose you add an alias named `abcd efgh 7890`, with replacement `script run x`.
  - Executing `abcd efgh 7890 abcd` is equivalent to executing `script run x abcd`.
- The alias name must appear at the start of a command line, followed by a space, for it to be recognised.
- There is no restriction on the characters in the alias name. However, leading and trailing whitespace will be trimmed.
- Aliases can expand to other aliases.

WARNING Built-in commands take precedence. If you add an alias with the same name as a built-in command, it will have no effect.

Delete an alias: `alias delete`

Deletes an alias.

Format: `alias delete <alias name>`

FAQ

Q: Which version of Java do I require to run this application?

A: Java 11

Q: How do I switch between the different tabs without clicking on them?

A: Simply execute the `list` command for that category, e.g. `rule list`. On the other hand, executing any command from that category will switch you over as well.

Q: How do I reset the application data?

A: All data is stored within the same folder as your execution path, under the "data/" folder. Deleting that folder will reset the application data.

Command Summary

Rules

- `rule add` - Add a new rule
- `rule list` - List rules
- `rule edit` - Edit a rule
- `rule delete` - Delete a rule

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.

Import and export

Implementation

The import functionality takes in a file and parses it into `Transaction`s.

All of the supported formats are comma-separated value (CSV) files. The bulk of the work is parsing the different ways the different banks represent the same data, and mapping each column from bank transaction export into the fields contained by Budget Buddy `Transaction`s.

The following class diagram illustrates the design of the import and export function.

TODO: Class diagram

Design considerations

TODO

Scripting

Implementation

The scripting engine works independently of the rest of the application. At its core, it uses the Nashorn ECMAScript 5.1 engine bundled with Java 11 to evaluate scripts.

A set of convenience functions are provided to make basic tasks, such as manipulating transactions and accounts, easier. The full model and UI are nevertheless exposed to scripts, and scripts are able to access any classes provided in the Java 11 standard library, as well as any dependencies included in the application.

There is a simple mechanism to store scripts to be run in future. This works together with rules to give the ability to have complex predicates and actions outside of those supported inherently by the program. This also works with aliases to allow, in effect, custom commands to be created.

The following class diagram illustrates the design of the scripting engine and model.

TODO: Class diagram

Design considerations

TODO

Aliases

Implementation

The alias is a simple hook into the command parsing engine. If there is no built-in command corresponding to a command line, then the alias map is checked. If there is a matching alias, then the alias name in the command line is replaced, and the command execution is re-tried.

To prevent alias loops where the user creates an alias `x` mapping to `y`, and an alias `y` mapping to `x`, we track the aliases that have been applied, and stop evaluation if we see that the same alias has been applied more than once.

Design considerations

TODO