CET2012 - Programming Paradigms: Advanced Java - Practicum 02

Topics Covered: Files I/O, Regular Expressions, Collections, OOP

Learning Objectives:

- Familiarize working with files in Java
- Familiarize with the usage of regular expressions
- Familiarize with the usages of some Java Collection Framework classes
- Learn and implement the Command Design Pattern

Deliverables:

- Submit a single zip file called `CET2012_P02_<Your_Name>.zip` (e.g. `CET2012_P02_John_Doe.zip`) containing your Java codes, include your driver class (aka class with the `main` method).

Background

About the Task

A company wants to develop a small tool that is able to store some employee data into a file. This tool allows users to **add, list, update, delete** and **undo the previous instruction**. In addition, this tool is able to load from a saved file at startup (if any) and store its content to a file after all commands have been executed, analogous to a cashier entering items to an Point of Sales machine or calling up a previous transaction then generating a receipt.

english characters only

The payload are all made up of all ASCII based characters only (i.e. all payload are Strings), however they are entered using the format detailed below. Depending on the command, the data items in the payload may differ. However, **there is a restriction for** `<data3>`. It is only able to accept **a single email address or Latin letters (case insensitive), digits 0 to 9 and underscores**. Each data item is to have Title Case (aka first letter is always in capitals) except emails.

email remains same

```
# Payload 1
<data1> <data2> <data3>
# Payload 2
<index> <data1> <data2> <data3>
# Payload 3
<index>
```

Although the email address syntax has a very wide range of allowable characters as detail in this Wikipedia article, the company has chosen to restrict the allowable characters to the following:

- **Local part:**
  - uppercase and lowercase Latin letters A to Z and a to z
  - digits 0 to 9
  - printable characters `._-` (dot, underscore & dash). Note that for the printable characters, they must not be present as either the first or last character and also do not appear consecutively. The underscore character is the exception to this rule.
- **Domain part (2 parts):** format `yyyy.xxx`
  - uppercase and lowercase Latin letters A to Z and a to z
  - digits 0 to 9
  - printable characters `.-` (dot & dash). Note that for the printable characters, they must not be present as either the first or last character and also do not appear consecutively
  - the "`.xxx`" part of the domain is allowed to have a range of minimum 2 to a maximum of 3 characters. In addition, only lowercase Latin letters a to z are allowed.

user input can be anything,
so need error handling of inputs

As stated above, there are 5 commands: `add`, `update`, `delete`, `list` and `undo`. Each command uses either a variation of the structure of *Payload 1* or *Payload 2* or none at all. The following table details the structure for each command.

Command	Payload Structure
Add	`<data1> <data2> <data3>` Other than the rules listed for `<data3>` above, `<data1>` and `<data2>` do not require error checking. All 3 data items are required to be filled when this command is used. only data3 needs error checking
Update	`<index> <data1> <data2> <data3>` The `<index>` and at least 1 data item is required. Order of the data item matters, i.e. `<data1>` value will update `<data1>` value added from the *Add* command. must have corresponding data indexes, cannot have missing items, e.g. missing data 2
Delete	`<index>` Numerical index based off the index shown using the *List* command.
List	No inputs required. Lists out the current items stored in the data store.
Undo	No inputs required. Undo the previous command. Applicable to the `add`, `update` & `delete` commands only. can undo more than once

new document, only cannot undo

Command Design Pattern in Brief

The command design pattern is a behavioural design pattern that turns a request/command into a standalone object that contains all information about the request/command. This pattern allows commands to be delayed or queued before execution, in addition, it supports undo operations.

The base structure of the Command Design Pattern is shown in figure 1 below.

concrete commands

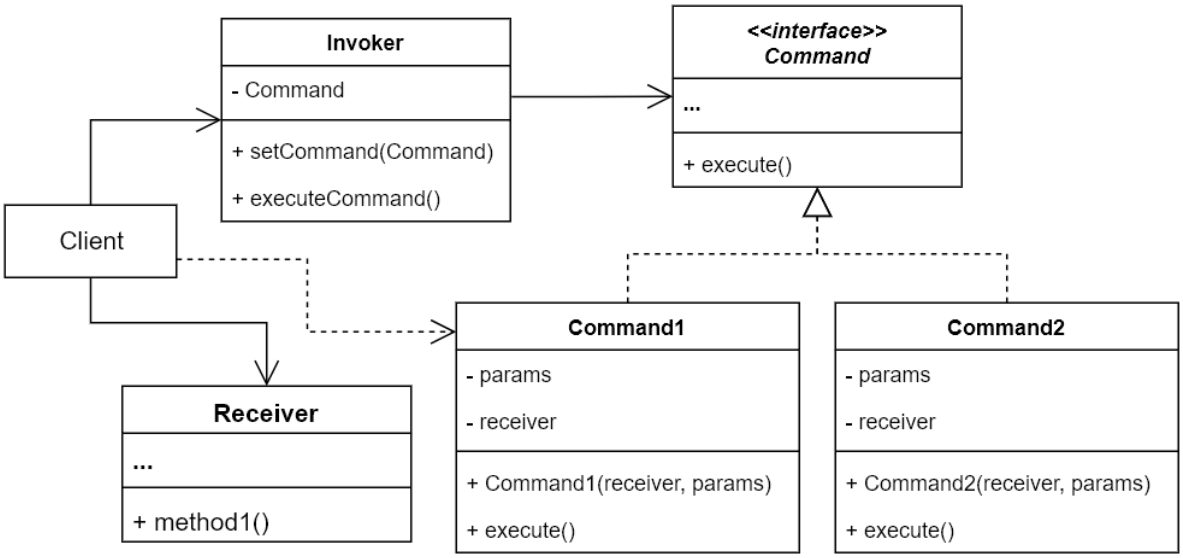

Figure 1: Command design pattern structure.

- The **Invoker** class is responsible for initiating the commands. It must have a field to store the reference to a command object. The Invoker **triggers** the commands instead of sending the request directly to the receiver. The invoker is not responsible for creating the command object, it usually gets a pre-created command from the client.
- The **Command** interface usually declares just a single method for executing the command.
- **Concrete Commands** implements the various kinds of commands. A concrete command isn't supposed to perform the work on its own, but rather to pass the call to one of the business logic objects. Parameters required to execute a method on a receiving object can be declared as fields in the concrete command.
- The **Receiver** class contains some business logic. Almost any object may act as a receiver. Most commands only handle the details of how a request is passed to the receiver, while the receiver itself does the actual work.
- The **Client** creates and configures the concrete command objects. The client must pass all of the command parameters, including a receiver instance, into the command's constructor. After that, the resulting command may be associated with one or multiple senders.

Real World Example

Word editors such as Microsoft Office or Notepad++ all have commands that result in the change in the state of the editor for example, cutting and pasting of text. Taking the Cut and Paste function from those editors, cut removes the selected text from the editor and places it in some memory (called the clipboard) whereas paste takes the text from the clipboard and places it at the location of the text cursor.

After any command is executed, it's placed into the command history (a stack of command objects) along with the backup copy of the editor's state at that point. Thus, if the user needs to revert an operation, the application can take the most recent command from the history, read the associated backup of the editor's state, and restore it.

The client code (GUI elements, command history, etc.) isn't coupled to concrete command classes because it works with commands via the command interface. This approach enables us to introduce new commands into an application without breaking any existing code.

Task

Your task is to develop this tool using the Command Design Pattern (from the previous section) and the following requirements.

- You are to use **one** `java.util.Stack` to store your command history and **one** `java.util.ArrayList` for your data store.
- For long-term storage of the entered data, you are to store the data in a text file called `dataStore.txt`. The location of the file should be created or read from the `src` folder of your project. **DO NOT submit this text file.** relative path
- You are to use regular expressions for checking the values for data item `<data3>`.
- You are to implement a single custom exception class for all non `Throwable` error messages.
- Include a `storeToFile()` method in your class that handles the data store. storeToFile() must be callable
- Concrete commands are to have the following classnames:
  - `AddCommand`
  - `UpdateCommand`
  - `DeleteCommand`
  - `ListCommand`
  - `UndoCommand`
- Ensure that your program is backwards compatible up to Java 8.
- Organize your Java files into appropriate folders.
- Include any other methods that you many require.
- Using the Javadoc tool, you are to include the documentation of your program to allow users to understand and be able to incorporate your tool to their own projects. **You are to ensure that your Javadoc of your project is generable.**
- The class diagram of the `Invoker` class is given below

cannot change invoker class

Invoker
- cmdToExecute: Command[]
+ setCommandsForExecution(Command[]) + executeCommand(Stack<Command> history)

(Stack<Command> history) to store command history

You may refer to an example of the behaviour of the program shown below

```
add
add
add
add
List
01. First_name Last_name Email
02. John Doe simple@example.com
03. Hanna Moon tetter.tots@potatoesarelife.com
04. Ah Boon green-tea@teaforlife.com
update  # 3 Adam
List
01. First_name Last_name Email
02. John Doe simple@example.com
03. Adam Moon tetter.tots@potatoesarelife.com
04. Ah Boon green-tea@teaforlife.com
update  # 1 blue bell ice-cream@alaskaFields.org
List
01. Blue Bell ice-cream@alaskaFields.org
02. John Doe simple@example.com
03. Adam Moon tetter.tots@potatoesarelife.com
04. Ah Boon green-tea@teaforlife.com
Delete  # 1
List
01. John Doe simple@example.com
02. Adam Moon tetter.tots@potatoesarelife.com
03. Ah Boon green-tea@teaforlife.com
Undo
List
01. Blue Bell ice-cream@alaskaFields.org
02. John Doe simple@example.com
03. Adam Moon tetter.tots@potatoesarelife.com
04. Ah Boon green-tea@teaforlife.com
```

no VPL, need to check with tutors for test cases
partner with Wei Long