Worksheet 3: Error Handling and Separation of Concerns

Updated: 19th March, 2018

1. Discussion: Exception Handling

Identify *four problems* with the exception handling in the following code, and say how you would fix them.

```
public void writeToFile(String filename, List<String> lineList)
{
    try
    {
        PrintWriter writer = new PrintWriter(filename);
        for(String line : lineList)
        {
            writer.println(line);
        }
        writer.close();
    }
    catch(Exception e)
    {
    }
    System.out.println("File successfully written");
}
```

2. Discussion: Separation of Concerns

For each of the following systems, suggest what the different "concerns" might be that ought to be separated.

Keep Model-View-Controller (MVC) in mind, but consider other concerns, and more fine-grained concerns as well. It may help to first list the actors (human and non-human) with which the system interacts.

(a) An aircraft autopilot system.

(b) A multiplayer first-person shooter game.

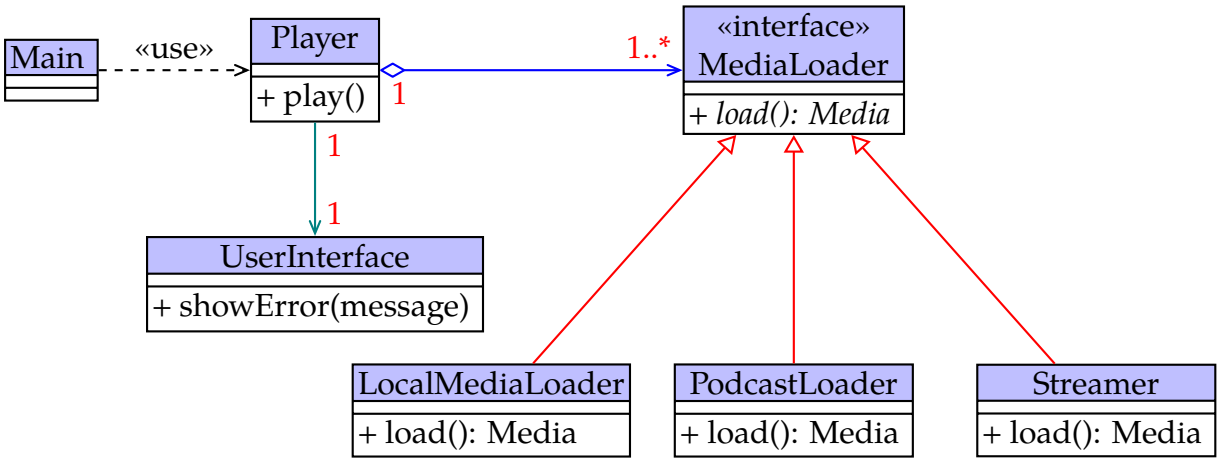
(c) A streaming/podcast application.

Note: This sort of exercise determines the large-scale structure of your software, and is basically the first step in the design process.

3. Discussion/Design: Chaining Exceptions

Imagine that the streaming/podcast application includes the following classes:

> **Note:** We're not looking at the complete system here – just enough of it for the purposes of this question.



The three subclasses of `MediaLoader` can all encounter `IOException` (standard Java) and `DecoderException`. However, `PodcastLoader` and `Streamer` rely on third-party Java libraries. These can throw `PodDownloadException` and `StreamConnectException` respectively.

Assume that the various `load()` methods are called by `Player.play()`, and that all four exception types should result in an error message being displayed.

> **Note:** Exceptions are generally designed to contain a string message, supplied when the exception is first created, and retrieved at any point within a `catch` block (e.g. `e.getMessage()` in Java).

(a) What is the role of the `MediaLoader` interface here, in terms of exception handling?

(b) If a key design goal is *separation of concerns*, then what is the ideal way to organise exception handling in this situation? Describe what each method does in terms of exception handling.

> **Warning:** Only `UserInterface` itself is allowed to use `System.out.println()` or equivalent!

4. Packages and (Sort-Of) MVC

Take the AddressBook system – ideally your previous code from worksheet 2. (The original worksheet 1 code would also work, but is slightly less interesting.) Separate the code into three packages, with the following names:

- `addressbook.model`
- `addressbook.view`
- `addressbook.controller`

> **Note:** You would not normally break up such a small system into packages. This exercise is just to give you practice.

To do this, you will need to:

- Set up the appropriate directory structure. (See the lecture notes.)
- Split up `AddressBookApp` into two or three classes, because some parts of it logically belong in different packages. The `addOption()` mutator mentioned in the previous worksheet may come in handy.
- Add `package` and `import` declarations (in that order) to each .java file as needed.
- Modify the build.xml file, replacing `AddressBookApp` with the "fully-qualified" name of the new class containing the `main()` method.

Once compiled, and if you're in the "AddressBook" directory (containing "build.xml", "resources", "src"), you should be able to run the application as follows:

```
[user@pc]$ java -jar dist/AddressBook.jar
```

OR

```
[user@pc]$ java -cp build [fully-qualified-main-class-name]
```

> **Note:** The "`-cp build`" option adds the "build" directory to the CLASSPATH temporarily. *Don't* add each individual package directory to the CLASSPATH – only the base directory that *contains* the package directories ("build" in this case).
>
> After compilation, the "build" directory contains the same subdirectories as "src", but with .class files instead of .java files. AddressBook.jar contains a zipped version of the "build" directory, plus a note saying which class contains `main()`.

5. Email (Optional Extra)

> **Note:** This is an old exercise from previous years. I've left it in, since it is still relevant as extra practice. However, the code may not work correctly on the lab machines, and unfortunately it cannot easily be translated to Java or C++ without the use of third-party libraries.
>
> However, the Python code should be reasonably easy to read and work with.

Obtain a copy of Emailer.zip from Blackboard. This is a very crude email-sending program, written in Python. You can run it as follows:

```
[user@pc]$ python2 emailer.py
```

Your task is twofold:

(a) By the ideals outlined in Lecture 3, the program is poorly structured. There are at least two major instances where separation of concerns is ignored. Where are these instances? How would you refactor the program so that it consists of well-separated "model", "view" and "controller" components?

(b) Email can be problematic. The program currently has no error handling. How would you handle the following problems?

(i) The user fails to enter a "to" address.

(ii) The user's username and password are rejected by the server.

(iii) The server connection is successful, but the server does not support TLS (an encryption standard).

In doing this:

- Consider what should actually happen.
- Consider what *part* of the program (class and method) is responsible for making it happen.
- Consider what code (including to create and/or catch exceptions) is needed to make it happen.

Note: Python's SMTP (email sending) library will produce exceptions under these conditions (along others):

- `starttls()` throws `SMTP.SMTPException` if TLS is not supported.
- `login()` throws `SMTP.SMTPAuthenticationError` if the server rejects the username and password.

Once you've determined what modifications need to be made, see if you can refactor the program.

End of Worksheet