Name:

Adapter Pattern

CSC-490, Winter 2020

1. (10pts) Suppose you have an abstract class `BoxOffice` that has methods `sellFilmTicket(film)` and `getTotalSales(film)`. This class is currently being used by a client for some important reason. In particular, modifying its interface would be costly. You also have a `BoxOfficeFactory` with a static factory method `createBoxOffice(type)` which creates various `ConcreteBoxOffice`'s.

   Now, another client comes along and requests an abstract class called `TicketOffice` with methods `sellMovieTicket(movie)` and `getTotalRevenue(movie)`. This second client is not willing to rename the class or the methods (maybe they already started working on other classes that rely on `TicketOffice`, or maybe they're just stubborn). The client would also like a `TicketOfficeFactory` with a static factory method `createTicketOffice(type)`

   At this point you are stuck with two different interfaces that essentially serve the same purpose. Find a way to implement a `ConcreteTicketOffice` without duplicating the implementation of methods in the `ConcreteBoxOffice`'s. Your solution cannot involve modifying the classes described in the previous paragraphs. In particular, you are not allowed to rename anything, or add/remove any new methods or data members to the classes in the previous paragraphs. Draw a class diagram for your solution. Include pseudocode for `createTicketOffice(type)` and all the methods in `ConcreteTicketOffice`.

2. (a) (2pts) Copy down the intent of the Adapter pattern from the Gang of Four:

   (b) (5pts) Draw the class diagram for the (object) Adapter pattern from the Gang of Four:

3. An adapter is sometimes called a *wrapper*.

   (a) (2pts) What other design pattern that we've seen is also called a wrapper?

   (b) (4pts) How is the Adapter pattern similar to your answer from part (a)?

   (c) (4pts) How is the Adapter pattern different than your answer from part (a)?

4. Consider an abstract class `OldClass` with methods `method1()`, `method2()`, and `method3()`. There is also a `ConcreteOldClass` where these methods are implemented. There are three types of clients for the abstract class. The first type `ClientA` only uses methods 1 and 2; the second type `ClientB` only uses method 3; and finally `ClientC` that make use of all three methods.

   (a) (2pts) Which SOLID design principle tells you that you should split `OldClass` into two classes?

   (b) (5pts) Your team decides to put `method1()` and `method2()` into a new abstract class called `ClassA`, and `method3()` into a new abstract class called `ClassB`. This decision makes `ClientA` and `ClientB` happy, but `ClientC` is not too happy. To appease `ClientC` you decide to keep the `OldClass` as a *facade* so that `ClientC` does not have to modify any of its code. Draw the class diagram for the desired new setup with classes `OldClass`, `ClassA`, `ClassB`, `ConcreteClassA`, and `ConcreteClassB` and the three clients. Include pseudocode for all the methods in the class acting as the facade.

   (c) (5pts) You realize that separating the implementation of the three methods into the separate classes, while doable, will take some time. However, the clients are ready to make their changes and move on. As a first step towards the goal (i.e. the diagram above) use the adapter method so that `ClassA` and `ClassB` make use of the current implementation of the methods in `OldClass`. Draw the corresponding class diagram involving the three abstract classes from the previous part and their corresponding concrete subclasses. Include pseudocode for all the methods in the two adapter classes.

(d) (5pts) You are able to move the implementation of `method2()` and `method3()` into their respective new concrete classes with relative ease. However, `method1()` is giving you some trouble because its current implementation is unnecessarily entangled with data and methods that really belong in `ClassB`. You have a plan for solving this issue, but it will take some time. For now, `method1()` will have to rely on the implementation in `ConcreteOldClass`. Draw the updated class diagram with only `method1()` relying on an adapter.

(e) (2pts) When will you be able to remove the `ConcreteOldClass`?

(f) (4pts) When are the clients allowed to change their dependency from `OldClass` to one of the new abstract classes? What happens if a client forgets to change a dependency from `OldClass` to `ClassA`?

5. In the CoCalc folder `Adapter/String` you'll find a class called `String` which is a wrapper for the standard string class. Such wrappers are not uncommon since programmers (or large teams of programmers) often prefer different interfaces for string methods. For instance, I don't really like writing `cout << str << "\n"` every time I want to print a string. I'd rather write `str.printLine()`. Also, the standard string class may not support methods that you commonly want to use such as splitting a string into a list of substrings.

   (a) (3pts) Is `String` an adapter or a decorator of the standard string class? Why?

   (b) (15pts) Implement each of the methods declared in `String.h` that are not already properly implemented in `String.cpp`. Read the comments in `String.h` to see what each method should do. All of your implementations should be in `String.cpp`. You can modify the `Main.cpp` file to check if your implementations are working.

   If you're not sure how to get going, to get started write a stub for each method that is missing in `String.cpp` so that the program compiles. I already put a stub in for `getLength()`. After everything compiles you'll know that your stub-implementations are *well-formed*. Then start replacing all your stubs (including the one I wrote) with proper implementations.