Name:

Abstract Factory Pattern

CSC-490, Winter 2020

1. (10pts) Consider the following story:

   - There is a class called `Container` that has a data member of type `Lid`. The class `Container` could have methods like `fill()`, `empty()`, `getVolume()`,.... The class `Lid` might have methods like `open()`, `close()`, `getArea()`,...
   - There are subclasses of those described above called `CylinderContainer`, `BowlContainer`, `CubeContainer`, `RoundLid`, and `SquareLid`. These classes have data members describing their sizes.
   - There is a class `Factory` that has factory methods `createContainer()` and `createLid()`. When a client calls these factory methods, they should return a container and a lid *that are compatible*. For example, a client should never get a `CylinderContainer` with a `SquareLid`.

   Draw a class diagram that could be implemented for this story. Include all the factory methods in your diagram, but you don't need to include any of the methods for the lids or containers.

2. (a) (2pts) Copy down the intent of the Abstract Factory pattern from the GoF:

(b) (8pts) Draw the class diagram for the Abstract Factory pattern.

(c) (5pts) Explain how the Abstract Factory pattern uses the Factory Method pattern.

(d) (8pts) Remember that a *factory method* is a method that is responsible for creating an object, and a *factory class* is a class whose single responsibility is creating objects. Determine whether each of the following statements is true or false:

i. The Abstract Factory pattern always involves a factory method?
ii. The Factory Method pattern always involves a factory method?
iii. The Abstract Factory pattern always involves a factory class?
iv. The Factory Method pattern always involves a factory class?
v. Every factory method is part of an Abstract Factory pattern?
vi. Every factory method is part of a Factory Method pattern?
vii. Every factory class is part of an Abstract Factory pattern?
viii. Every factory class is part of a Factory Method pattern?

3. Recall the class `BadGuy` from our discussion on the Strategy pattern. Suppose we want to create a bad guy at a given level in the game. There should be `BadGuyFactory` with a method `createBadGuy()` that does the job. Now, remember that a bad guy is determined by its chasing and fighting behaviors. For this exercise, let's consider the following behaviors:

fights with:	gun	knife	
chases by:	boat	swimming	running

That means that to create a bad guy you need to create its two behaviors. Here's the rub: only certain behaviors are appropriate for any particular level. For example, suppose Level 1 is a land level where bad guys all chase by running, and fight with either a gun or a knife. On the other hand, Level 2 is a water level where some bad guys chase in a boat with a gun and other bad guys swim with a knife.

(a) (10pts) Draw a class diagram that uses the Abstract Factory pattern implement the `BadGuyFactory`.

(b) (2pts) Include pseudocode for the implementation of the operation `createBadGuy()`.

4. This exercise is concerned with the program in the CoCalc folder `AbstractFactory/RandomFactories`. In that file you'll see a bare bones implementation of the Abstract Factory pattern.

   (a) (10pts) Create a new subclass of `Factory` called `RandomFactory`. The factory methods in this class should return random products of the appropriate type. In other words, the first time the factory methods are called on an object they might return products of type A1 and B2, and the next time those methods are called on the *same* object they might return products of type A2 and B2.

   (b) (10pts) Create another concrete factory called `RandomCompatibleFactory`. For this factory, calling `createProductA()` should randomly return a product of type A1 or A2. Whenever `createProductA()` returns A1, successive calls to `createProductB()` should return B1. Similarly, if `createProductA()` returns A2, then `createProductB()` should return B2.[1]

[1] This type of scenario might be appropriate for something like the container-lids story. In that case the factory would return a random container (product A) and then return a lid (product B) that fits the container.