Project 3: Bakery

1 Objective

This project is the first time you'll create your own objects, along with private data, methods, etc. This isn't a complete working system with a user interface; it's just a chance for you to create classes, then instantiate and test them. You'll create two *supplier* classes that interact with each other (through a "has a" relationship).

2 Classes You'll Create

Here are UML Class Diagrams for the classes you are to create. Pay attention to the diagram notation indicating whether methods are public (+) or private (-); ask questions if you need clarification. Remember that UML is generic; it's not Java. Understanding the model is of *critical* importance here.

BakeryOrder

(*you figure out the private data needed*)

+BakeryOrder(orderNum : Integer)
+BakeryOrder(orderNum : Integer, pieCount : Integer, cakeCount : Integer, cupcakeDozCount: Integer)

+getOrderNumber() : Integer
+getPieCount() : Integer
+getCakeCount() : Integer
+getCupcakeDozCount() : Integer
+getItemCount() : Integer
+getPrice() : Double
+toString() : String

+setPieCount(pieCount : Integer)
+setCakeCount(cakeCount : Integer)
+setCupcakeDozCount(cupcakeDozCount : Integer)

BakeShop

(*you figure out the private data needed*)

+BakeShop()
+BakeShop(bakerCount : Integer)

+getPiesOnOrderCount() : Integer
+getCakesOnOrderCount() : Integer
+getCupcakeDozOnOrderCount() : Integer
+getOrderCount() : Integer
+getTotalSales() : Double
+getAvgOrderPrice() : Double
+getBakerCount() : Integer
+toString() : String

+hireOneBaker()
+fireOneBaker()

+submitOrder(oneOrder : BakeryOrder) : Double

You *may* also create a Main class with a runnable main method that demonstrates what your project can do. This is not a required part of the project; do it if it makes you feel better about how everything works together. Note that the test code should exercise the code in similar ways (i.e., integrating the objects and creating has-a relationships), with rigorous verification along with it.

Such is the nature of supplier code; you aren't writing the whole application, so it takes getting used to.

3 Constraints and Assumptions

3.1 General

- Create *no* static methods in the production (non-test) classes.
- Mark *each* instance variable and *each* method as either public or private; follow the UML where it gives guidance and make smart decisions where it doesn't.
- Use the *exact* method names shown. Look carefully at the parameter data types and the return data types; they give you clues.
- The toString functions should do their best to summarize the state of the object instance in questions; include newlines in the strings to make the result displayable and attractive.
- Create JUnit unit tests for each object's methods. Since there is no UI, this is the way to ensure that any future clients of your code are getting exactly what they expect.

3.2 BakeryOrder

- You aren't responsible for incrementing order numbers to make them sequential; assume they are generated in some other system. The order number is supplied to the BakeryOrder via the constructor, as noted in the UML Class Diagram.
- Pies cost $19.99, cakes $35.99, and cupcakes $23.99 a dozen.
- There are no flavor choices available; all pies, cakes, and cupcakes are the same.

3.3 BakeShop

- BakeShop's submitOrder maintains the appropriate counts and totals and returns the price of the order just processed. We aren't *storing* these orders; handle what's needed at submit time.
- BakeShop's getAvgOrderPrice calculates the average order price, returning 0 if no orders exist.
- If you fire a baker, you don't need to worry about the existing order workload; you'll simply not accept any more orders that day (preconditions will take care of that) if you're shorthanded.

3.4 Preconditions

Throw an IllegalArgumentException if any of these preconditions are violated:

- Orders can't be submitted if they exceed bakery capacity; each baker can handle no more than 25 items, a total of pies, cakes, and cupcake dozens.
- The number of pies, cakes, and cupcake dozens passed as arguments must be non-negative.
- Order numbers must be non-negative integers.
- You try to fire more bakers than you currently have employed.
- A BakeShop is instantiated with a negative number of bakers

4 Code Implementation

Follow the provided Course Style Guide. This includes JavaDoc notation.

5 Testing

The testing required here is significant. You may spend as much time writing your tests as you do writing the entire rest of the code. Your test code needs to test everything (or nearly everything, the specific results of toString aside) your object can do; don't treat it lightly. Don't forget to test constructors and preconditions. And don't forget test code is *still code* and can have bugs, so be suspicious if everything passes miraculously the first time, i.e., test that you can induce failures and get the appropriate error messages shown.

6 Submitting Your Work

Use BlueJ to create a .jar file. Be sure and specify "include source" so your code will be included. Submit the .jar file.

7 Hints

Don't duplicate code; avoid this wherever possible. As an example, don't forget it's legal for one constructor to call another constructor; in fact, this is a good practice and helps avoid duplication and the creation of extra code paths to test.

8 Grading Matrix and Achievement Levels

Achievement	**Max Points**
Compile/runtime errors	50%
BakeryOrder	60%
BakeShop	70%
JUnit tests c'tor & methods	80%
JUnit tests for preconditions	90%
Documentation/Style	100%