Deliverables

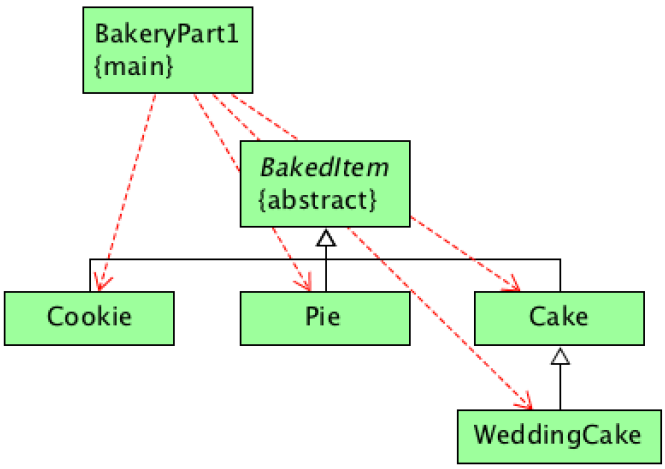
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment this week which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code. You may submit your completed code up to 24 hours after the due date, but there is a late penalty of 15 points. No projects will be accepted after the one-day late period. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The Completed Code will be tested against your test methods in your JUnit test files and against the usual correctness tests. The grade will be determined, in part, by the tests that you pass or fail and the level of coverage attained in your Java source files by your test methods.

Files to submit to Web-CAT:

- BakedItem.java
- Cookie.java, CookieTest.java
- Pie.java, PieTest.java
- Cake.java, CakeTest.java
- WeddingCake.java, WeddingCakeTest.java
- BakeryPart1.java, BakeryPart1Test.java

Specifications

Overview: This project is the first of three that will involve a bakery and reporting for baked items. You will develop Java classes that represent categories of baked items including cookies, pies, cakes, and wedding cakes. You will also develop a driver class with a main method. As you develop each class, you should create the associated JUnit test file with the required test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project upfront and then add the source and test files as they are created. All of your files should be in a single folder. Below is the UML class diagram for the required classes which shows the inheritance relationships.



You should read through the remainder of this assignment before you start coding.

- **BakedItem.java**

Requirements: Create an *abstract* BakedItem class that stores BakedItem data and provides methods to access the data.

Design: The BakedItem class has fields, a constructor, and methods as outlined below.

(1) **Fields**: *instance* variables for the BakedItem's name of type String, the flavor type String, the quantity of type int, and ingredients of type String[]; *static* (or class) variable of type int for the count of BakedItem objects that have been created (set to zero when declared and incremented in the constructor). These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of BakedItem. These are the only fields that this class should have.

(2) **Constructor**: The BakedItem class must contain a constructor that accepts four parameters representing the values to be assigned to the *instance* fields: name, flavor, quantity, and ingredients. The last parameter (e.g., ingredientIn) should be a variable length parameter (i.e., `String ... ingredientsIn`), which will be type String[] in the constructor body. Since this class is abstract, the constructor will be called from the subclasses of BakedItem using *super* and the parameter list. The count field should be incremented in this constructor.

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getName`: Accepts no parameters and returns a String representing the name.
- `setName`: Accepts a String representing the name, sets the field, and returns nothing.
- `getFlavor`: Accepts no parameters and returns a String representing the flavor.
- `setFlavor`: Accepts a String representing the flavor, sets the field, and returns nothing.
- `getQuantity`: Accepts no parameters and returns a int representing quantity.
- `setQuantity`: Accepts an int representing the quantity, sets the field, and returns nothing.
- `getIngredients`: Accepts no parameters and returns a String[] representing ingredients.
- `setIngredients`: Accepts a variable length parameter (i.e., `String ... ingredientsIn`) representing the ingredients, which will be type String[] in the method body, sets the field, and returns nothing.
- `getCount`: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.

- `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
- `toString`: Returns a String describing the BakedItem object. This method will be inherited by the subclasses and should be declared *public final* so that it cannot be overriden. The double value for the price() should be formatted (`"$#,##0.00"`), but the numeric value for quantity does not require formatting. Below is an example of the toString result when called on a Cookie object of the Cookie class described below. Note that "Cookie" is the class name which is followed by name and flavor separated by a dash. Quantity and Price are preceded by three spaces. The ingredients, which are enclosed in parentheses, begin on a new line and are separated by commas. Also, there should be at most five ingredients on a line. For example, "salt" is the sixth ingredient so it was printed on the next line.

```
Cookie: Chips Delight - Chocolate Chip   Quantity: 12   Price: $4.20
(Ingredients: flour, sugar, dark chocolate chips, butter, baking soda,
salt)
```

  Note that you can get the class name for the instance by calling this.getClass().toString().substring(6). For the example Cookie c, this.getClass().toString() returns "class Cookie" and substring(6) extracts the class name "Cookie" which begins at character 6. This approach allows the toString method in BakedItem to work for all subclasses that inherit the toString method.
- `price`: An *abstract* method that accepts no parameters and returns a double representing the price for a BakedItem. Since this is an abstract method, it has no body in BakedItem; however, each non-abstract subclass must implement this method.

Code and Test: Since the BakedItem class is abstract you cannot create instances of BakedItem upon which to call the methods. However, these methods will be inherited by the subclasses of BakedItem. You should consider first writing skeleton code for the methods in order to compile BakedItem so that you can create the first subclass described below. At this point you can begin completing the methods in BakedItem and writing the JUnit test methods for your subclass that tests the methods in BakedItem.

- **Cookie.java**

Requirements: Derive the class Cookie from BakedItem.

Design: The Cookie class has fields, a constructor, and methods as outlined below.

(1) **Fields**: a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 0.35. This is the only field that should be declared in this class.

(2) **Constructor**: The Cookie class must contain a constructor that accepts four parameters representing the four instance fields in the BakedItem class (name, flavor, quantity, and ingredients). Since this class is a subclass of BakedItem, the super constructor should be called with field values for BakedItem. Below is an example of how the constructor could be used to create a Cookie object:

```
Cookie c = new Cookie("Chips Delight", "Chocolate Chip", 12,
                      "flour", "sugar", "dark chocolate chips",
                      "butter", "baking soda", "salt");
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following price method <u>but not the toString method</u>.

- `price`: Accepts no parameters and returns a double representing the price for the Cookie calculated as follows: BASE_RATE * quantity.
- `toString`: <u>None</u> - this method is inherited from BakedItem. An example of the return value is shown below for Cookie c created above. Note that Quantity and Price are preceded by three spaces.

```
Cookie: Chips Delight - Chocolate Chip   Quantity: 12   Price: $4.20
(Ingredients: flour, sugar, dark chocolate chips, butter, baking soda,
salt)
```

Code and Test: As you implement the Cookie class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. You can now continue developing the methods in BakedItem (parent class of Cookie). The test methods in CookieTest should be used to test the methods in both BakedItem and Cookie. Remember, Cookie *is-a* BakedItem which means Cookie inherited the instance methods defined in BakedItem. Therefore, you can create instances of Cookie in order to test methods of the BakedItem class. You may also consider developing BakeryPart1 (page 8) in parallel with this class to aid in testing.

- **<u>Pie.java</u>**

Requirements: Derive the class Pie from BakedItem.

Design: The Pie class has a field, a constructor, and methods as outlined below.

(1) **Field**: an *instance* variable for crust cost of type double, which is declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 12.0. <u>These are the only fields that should be declared in this class</u>.

(2) **Constructor**: The Pie class must contain a constructor that accepts five parameters representing the four instance fields in the BakedItem class and the one instance field crust cost declared in Pie (name, flavor, quantity, crust cost, and ingredients). Since this class is a subclass of BakedItem, the super constructor should be called with field values for BakedItem. Below are examples of how the constructor could be used to create a Pie p1 (without a crust cost) and Pie p2 (with a crust cost of $2).

```
Pie p1 = new Pie("Weekly Special", "Apple", 1, 0,
                 "flour", "sugar", "apples", "cinnamon",
                 "butter", "baking soda", "salt");
```

```
Pie p2 = new Pie("Summer Special", "Key Lime", 1, 2.0,
                 "flour", "sugar", "lime juice", "lemon juice",
                 "graham crackers", "butter", "baking soda", "salt");
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following getCrustCost, setCrustcost, and price methods but not the toString method.

- `getCrustCost`: Accepts no parameters and returns a double representing crust cost.
- `setCrustCost`: Accepts double for crust cost, sets the field, and returns nothing.
- `price`: Accepts no parameters and returns a double representing the price for a Pie calculated as follows: (BASE_RATE + crustCost) * quantity.
- `toString`: NONE. When toString is invoked on an instance of Pie, the toString method inherited from BakedItem is called. Below is an example of the toString result for Pie p1 as it is declared above.

```
Pie: Weekly Special - Apple   Quantity: 1   Price: $12.00
(Ingredients: flour, sugar, apples, cinnamon, butter,
baking soda, salt)

Pie: Summer Special - Key Lime   Quantity: 1   Price: $14.00
(Ingredients: flour, sugar, lime juice, lemon juice, graham crackers,
butter, baking soda, salt)
```

Code and Test: As you implement the Pie class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. For example, as soon as you have implemented and successfully compiled the constructor, you should create an instance of Pie in a JUnit test method in the PieTest class and then run the test file. If you want to view your objects in the Canvas, set a breakpoint in your test method and then run *Debug* on the test file. When it stops at the breakpoint, step until the object is created. Then open a canvas window using the canvas button at the top of the Debug tab. After you drag the instance onto the canvas, you can examine it for correctness. If you change the viewer to "toString" view, you can see the formatted toString value. You can also enter the object variable name in interactions and press ENTER to see the toString value. *Hint: If you use the same variable names for objects in the test methods, you can use the menu button on the viewer in the canvas to set "Scope Test" to "None". This will allow you to use the same canvas with multiple test methods*. You may also consider developing BakeryPart1 (page 8) in parallel with this class to aid in testing.

- **Cake.java**

Requirements: Derive the class Cake from class BakedItem.

Design: The Cake class has a field, a constructor, and methods as outlined below.

(1) **Field**: *instance* variable for layers of type int, which should be declared with the *protected* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 8. These are the only fields that should be declared in this class.

(2) **Constructor**: The Cake class must contain a constructor that accepts five parameters representing the four instance fields in the BakedItem class and the one instance field layers declared in Cake (name, flavor, quantity, layers, and ingredients). Since this class is a subclass of BakedItem, the super constructor should be called with field values for BakedItem. Below is an example of how the constructor could be used to create a Cake c1 with one layer and Cake c2 with two layers.

```
Cake c1 = new Cake("Birthday", "Chocolate", 1, 1,
            "flour", "sugar", "cocoa powder", "vanilla", "eggs",
            "butter", "baking soda", "baking powder", "salt");

Cake c2 = new Cake("2-Layer", "Red Velvet", 1, 2,
            "flour", "sugar", "cocoa powder", "food coloring",
            "eggs", "butter", "baking soda", "baking powder",
            "salt");
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following getLayers, setLayers, and price methods but not the toString method.

- `getLayers`: Accepts no parameters and returns an int representing layers.
- `setLayers`: Accepts an int representing layers, sets the field, and returns nothing.
- `price`: Accepts no parameters and returns a double representing the price for the Cake calculated as follows: (BASE_RATE * layers) * quantity.
- `toString`: None - this class uses the toString method inherited from BakedItem. An example of the return value is shown below for Cake c1 and Cake c2 created above. Note that Quantity and Price are preceded by three spaces.

```
Cake: Birthday - Chocolate   Quantity: 1   Price: $8.00
(Ingredients: flour, sugar, cocoa powder, vanilla, eggs,
butter, baking soda, baking powder, salt)

Cake: 2-Layer - Red Velvet   Quantity: 1   Price: $16.00
(Ingredients: flour, sugar, cocoa powder, food coloring, eggs,
butter, baking soda, baking powder, salt)
```

Code and Test: As you implement the Cake class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Cookie and Pie classes. You may also consider developing BakeryPart1 (below) in parallel with this class to aid in testing.

- **WeddingCake.java**

  Requirements: Derive the class WeddingCake from Cake.

  Design: The WeddingCake class has a fields, a constructor, and methods as outlined below.

  (1) **Fields**: *instance* variables for tiers of type int which should be declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 15.0. These are the only fields that should be declared in this class.

  (2) **Constructor**: The WeddingCake class must contain a constructor that accepts six parameters representing the four values for the instance fields in the BakedItem and two for the instance fields declared in Cake and WeddingCake respectively (name, flavor, quantity, layers, tiers, and ingredients). Since this class is a subclass of Cake, the super constructor should be called with five values for Cake. Below is an example of how the constructor could be used to create a WeddingCake object.

```
WeddingCake c3 = new WeddingCake("3-Layer/3-Tier", "Vanilla", 1, 3, 3,
                  "flour", "sugar", "buttermilk", "coffee",
                  "eggs", "butter", "baking soda", "baking powder",
                  "salt");
```

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following getTiers, setTiers, and price methods but not the toString method.

  - `getTiers`: Accepts no parameters and returns an int representing tiers.
  - `setTiers`: Accepts an int representing tiers, sets the field, and returns nothing.
  - `price`: Accepts no parameters and returns a double representing the price for a WeddingCake calculated as follows: (BASE_RATE * layers * tiers) * quantity.
  - `toString`: None - this class uses the toString method inherited from BakedItem. An example of the return value is shown below for WeddingCake c3 created above. Note that Quantity and Price are preceded by three spaces.

```
WeddingCake: 3-Layer/3-Tier - Vanilla   Quantity: 1   Price: $135.00
(Ingredients: flour, sugar, buttermilk, coffee, eggs,
butter, baking soda, baking powder, salt)
```

  Code and Test: As you implement the WeddingCake class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Cookie, Pie, and Cake classes. You may also consider developing BakeryPart1 (below) in parallel with this class to aid in testing.

- **BakeryPart1.java**

Requirements: This driver class with a main method is optional but you may find it helpful.

Design: The BakeryPart1 class only has a main method as described below.

The main method should be developed incrementally along with the classes above. For example, when you have compiled BakedItem and Cookie, you can add statements to the main method that create and print an instance of Cookie. Remember, since BakedItem is abstract you cannot create an instance of it. When main is completed, it should contain statements that create and print instances of Cookie, Pie, Cake, and WeddingCake. Since printing the objects will not show all of the details of the fields, you should also run in canvas (or debug with a breakpoint) to examine the objects. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create Cookie c, Pie p1, Pie p2, Cake c1, Cake c2, and WeddingCake c3 as described in the sections above and your main method is stopped between steps after c3 has been created, you can enter the following in interactions to get the price for the WeddingCake object.

```
c3.price()
135.0
```

The output from main assuming you create print the five objects Cookie c, Pie p1, Pie p2, Cake c1, Cake c2, and WeddingCake c3 as described in the sections above is shown as below. Note that new lines were added by main to achieve the spacing between objects.

```
Cookie: Chips Delight - Chocolate Chip   Quantity: 12   Price: $4.20
(Ingredients: flour, sugar, dark chocolate chips, butter, baking soda,
salt)

Pie: Weekly Special - Apple   Quantity: 1   Price: $12.00
(Ingredients: flour, sugar, apples, cinnamon, butter,
baking soda, salt)

Pie: Summer Special - Key Lime   Quantity: 1   Price: $14.00
(Ingredients: flour, sugar, lime juice, lemon juice, graham crackers,
butter, baking soda, salt)

Cake: Birthday - Chocolate   Quantity: 1   Price: $8.00
(Ingredients: flour, sugar, cocoa powder, vanilla, eggs,
butter, baking soda, baking powder, salt)

Cake: 2-Layer - Red Velvet   Quantity: 1   Price: $16.00
(Ingredients: flour, sugar, cocoa powder, food coloring, eggs,
butter, baking soda, baking powder, salt)

WeddingCake: 3-Layer/3-Tier - Vanilla   Quantity: 1   Price: $135.00
(Ingredients: flour, sugar, buttermilk, coffee, eggs,
butter, baking soda, baking powder, salt)
```

Code and Test: After you have implemented the BakeryPart1 class, you should create the test file BakeryPart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in BakedItem and visible to each subclass. In the test method, you should reset the *count*, call your main method, then assert that *count* is five (assuming that your main creates six objects indicated above). The following statements accomplish the test.

```
// covers the default constructor (Web-CAT will check for this)
BakeryPart1 bp1 = new BakeryPart1();

// checks class variable count
BakedItem.resetCount();
BakeryPart1.main(null);
Assert.assertEquals("BakedItem.count should be 6. ",
                    6, BakedItem.getCount());
```

Canvas for BakedItemBoardingPart1

Below is an example of a jGRASP viewer canvas for BakeryPart1 that contains a viewer for the class variable BakedItem.count and two viewers for each of Cookie c, Pie p1, Pie p2, Cake c1, Cake c2, and WeddingCake c3. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. Notice that runtime types are shown in the object viewer labels. To turn on this feature, click View on the top menu, then select "Show Runtime Types in Viewer Labels" (you should see a check mark in the associated check box when this is on). The canvas was created dragging instances from the debug tab into a new canvas window and setting the appropriate viewer. Note that you will need to unfold one of the instances in the debug tab to find the static variable *count*.