Deliverables

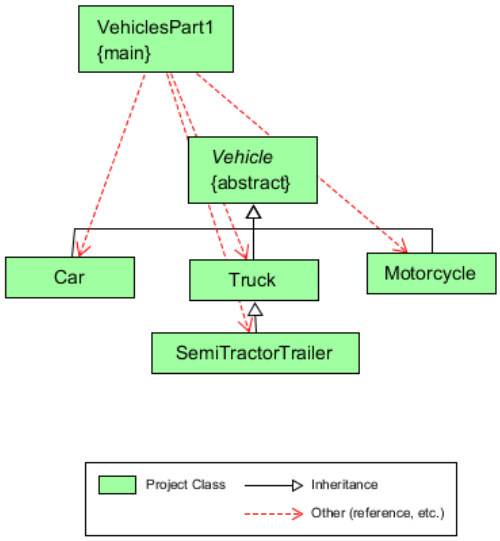
Your project files should be submitted for grading by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will indicate level of coverage your tests have achieved. The files you submit to skeleton code assignment may be incomplete in the sense 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 files to the Completed Code assignment no later than 11:59 PM on the due date. If you are unable to submit to the grading system, you should e-mail your project Java files in a zip file to your TA before the deadline. Your grade will be determined, in part, by the tests that you pass or fail in your test files and by the level of coverage attained in your source files, as well as our usual correctness tests.

Files to submit to for grading:

- Vehicle.java
- Car.java, CarTest.java
- Truck.java, TruckTest.java
- SemiTractorTrailer.java, SemiTractorTrailerTest.java
- Motorcycle.java, MotorcycleTest.java
- VehiclesPart1.java, VehiclesPart1Test.java

Specifications

Overview: This project is the first of three parts that will involve calculating the annual use tax for vehicles where the amount is based on the type of vehicle, its value, and various tax rates. You will develop Java classes that represent categories of vehicles: car, truck, semi-tractor trailer (a subclass of truck), and motorcycle. These categories will be implemented as follows: an abstract Vehicle class which has three subclasses Car, Truck, and Motorcycle. The Truck class has a subclass SemiTractorTrailer. The driver class for this project, VehiclesPart1.java, should contain a main method that creates one or more instances of each of the non-abstract classes in the Vehicle hierarchy. As you develop each non-abstract class, you should add code in the main method to create and print one or more instances of the class. Thus, after you have created all the classes, your main method should create and print one or more objects (e.g., at least one for each of the types Car, Truck, SemiTractorTrailer, and Motorcycle). You can use VehiclesPart1in conjunction with interactions by running the program in the canvas (or debugger with a breakpoint) and single stepping until the each of the instances is created. You can then enter interactions for the instances in the usual way. However, a more efficient way to test your methods would be to create the JUnit test file (required for this project) for each class and write an appropriate number of test methods to ensure the classes and



methods meet the specifications. All of your files should be in a single folder. You should create a jGRASP project upfront and then add the source and test files as they are created. You should generate (or regenerate) the UML class diagram each time you add a class to the project (see page 9).

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

- **Vehicle.java**

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

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

  (1) **Fields**: Four *instance* variables for the vehicle's owner of type String, yearMakeModel of type String, value of type double, and alternative fuel of type boolean. These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of Vehicle. The last field should be a *static* variable, vehicleCount, of type int with *protected* access. This class variable is used to track the number of vehicles that are created from the classes in the Vehicle hierarchy. These are the only fields that this class should have.

  (2) **Constructor**: The Vehicle class has a constructor that accepts four parameters representing the values for the respective *instance* variables, assigns the values, and then increments the vehicleCount class variable. Since this class is abstract, the constructor cannot be called with the *new* operator to create an instance of Vehicle. Instead, the constructor will be called from the constructors of the subclasses of Vehicle using *super* and the parameter list.

  (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.
  - `getOwner`: Accepts no parameters and returns a String representing the vehicle's owner.
  - `setOwner`: Accepts a String representing the owner, sets the field, and returns nothing.
  - `getYearMakeModel`: Accepts no parameters and returns a String representing the yearMakeModel field.
  - `setYearMakeModel`: Accepts a String representing yearMakeModel, sets the field, and returns nothing.
  - `getValue`: Accepts no parameters and returns a double representing the value of the vehicle.
  - `setValue`: Accepts a double representing value, sets the field, and returns nothing.
  - `getAlternativeFuel`: Accepts no parameters and returns the boolean value for the alternative fuel field.

- `setAlternativeFuel`: Accepts a boolean, sets the alternative fuel field, and returns nothing.
- `getVehicleCount`: Accepts no parameters and returns an int representing the count. Since vehicleCount is *static*, this method should be *static* as well.
- `resetVehicleCount`: Accepts no parameters, resets vehicleCount to zero, and returns nothing. Since vehicleCount is *static*, this method should be *static* as well.
- `useTax`: An *abstract* method that accepts no parameters and returns a double representing the total amount for the vehicle's use tax. Note this method has no body in Vehicle; it will be overridden in its subclasses.
- `toString`: Returns a String describing the Vehicle. This method will be inherited by the subclasses although it may be overridden in the subclass as appropriate. If it is overridden, then it may be called from the toString method in the subclasses of Vehicle using super.toString(). For examples of the toString result, see the Car class below. The first two lines in each result should be produced by this toString method. Note that you can get the class name within the class by calling: `this.getClass().getName()`
- `equals`: Overrides the `equals` method inherited from the Object class. Accepts a parameter of type Object, and returns true if this vehicle owner, yearMakeModel, and value are all equal to the parameter's vehicle owners, yearMakeModel, and value; otherwise returns false. You may use the following code.

```
/**
 * @param obj the other object
 * @return boolean
 */
public boolean equals(Object obj) {

   if (!(obj instanceof Vehicle)) {
      return false;
   }
   else {
      Vehicle other = (Vehicle) obj;
      return (owner + yearMakeModel + value).
            equals(other.owner + other.yearMakeModel
                  + other.value);
   }
}
```

- `hashCode`: Overrides the hashCode method inherited from the Object class. Accepts no parameters and returns 0. This method is required by Checkstyle if you overrideYou may use the following code.

```
/** @return 0 */
public int hashCode() {
   return 0;
}
```

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

- **Car.java**

Requirements: Derive the Car class from the Vehicle class.

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

(1) **Fields**: Four public constants of type double: TAX_RATE = 0.01, ALTERNATIVE_FUEL_TAX_RATE = 0.005, LUXURY_THRESHOLD = 50_000, and LUXURY_TAX_RATE = 0.02. These are the only fields that should be declared in this class.

(2) **Constructor**: The Car class must contain a constructor that accepts four parameters representing the four instance fields in the Vehicle class. Since this class is a subclass of Vehicle, the super constructor should be called with field values for Vehicle. Below are examples of how the constructor could be used to create a Car object:

```
Car car1 = new Car("Jones, Sam", "2017 Honda Accord", 22000, false);
Car car2 = new Car("Jones, Jo", "2017 Honda Accord", 22000, true);
Car car3 = new Car("Smith, Pat", "2015 Mercedes-Benz Coupe",
                   110000, false);
Car car4 = new Car("Smith, Jack", "2015 Mercedes-Benz Coupe",
                   110000, true);
```

(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.

- `useTax`: Accepts no parameters and returns a double representing the total use tax calculated as follows. If the car uses alternative fuel, it's calculated as value * ALTERNATIVE_FUEL_TAX_RATE; otherwise it's calculated as value * TAX_RATE. Furthermore, if the value is exceeds LUXURY_THRESHOLD, then value * LUXURY_TAX_RATE is added to get the total use tax.
- `toString`: The toString method required in the Car class should invoke the toString method in Vehicle and then append a line to the result. The line to be appended consists of the tax or taxes used in the calculation of the use tax. Below are examples of the toString result for car1, car2, car3, and car4 as declared above.

```
Jones, Sam: Car 2017 Honda Accord
Value: $22,000.00 Use Tax: $220.00
with Tax Rate: 0.01

Jones, Jo: Car 2017 Honda Accord (Alternative Fuel)
Value: $22,000.00 Use Tax: $110.00
with Tax Rate: 0.005
```

```
Smith, Pat: Car 2015 Mercedes-Benz Coupe
Value: $110,000.00 Use Tax: $3,300.00
with Tax Rate: 0.01 Luxury Tax Rate: 0.02

Smith, Jack: Car 2015 Mercedes-Benz Coupe (Alternative Fuel)
Value: $110,000.00 Use Tax: $2,750.00
with Tax Rate: 0.005 Luxury Tax Rate: 0.02
```

Code and Test: As you implement the Car 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 Vehicle (parent class of Car). The test methods in CarTest should be used to test the methods in both Vehicle and Car. Remember, Car *is-a* Vehicle which means Car inherited the instance methods defined in Vehicle. Therefore, you can create instances of Car in order to test methods of the Vehicle class. You can test the *equals* method by creating three Car objects where the first two are equal and the third is not equal. Then assert that car1.equals(car2) is true and assert that car1.equals(car3) is false. To cover the hashCode method, you can assert that car1.hashCode() equals car2.hashCode(). You may also consider developing VehiclesPart1 (page 8) in parallel with this class to aid in testing.

- **Truck.java**

Requirements: Derive a Truck class from the Vehicle class.

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

(1) **Field**: an instance variable for tons of type double, which should be declared with the *protected* access modifier; four public constants of type double: TAX_RATE = 0.02, ALTERNATIVE_FUEL_TAX_RATE = 0.01, LARGE_TRUCK_TONS_THRESHOLD = 2.0, and LARGE_TRUCK_TAX_RATE = 0.03. These are the only fields that should be declared in this class.

Constructor: The Truck class must contain a constructor that accepts five parameters representing the four instance fields in the Vehicle class and one instance field from Truck. Since this class is a subclass of Vehicle, the super constructor should be called with field values for Vehicle. The instance variable for tons should be set with the last parameter of the Truck constructor. Below is an example of how the constructor could be used to create a Truck object.

```
Truck truck1 = new Truck("Williams, Jo", "2012 Chevy Silverado",
                         30000, false, 1.5);
Truck truck2 = new Truck("Williams, Sam", "2010 Chevy Mack",
                         40000, true, 2.5);
```

(2) **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.

- `getTons`: Accepts no parameters and returns a double representing the tons field.

- `setTons:` Accepts a double representing the tons, sets the field, and returns nothing.
- `useTax`: Accepts no parameters and returns a double representing the total use tax calculated as follows. If the truck uses alternative fuel, it's calculated as value * ALTERNATIVE_FUEL_TAX_RATE; otherwise it's calculated as value * TAX_RATE. Furthermore, if the tons is exceeds LARGE_TRUCK_TONS_THRESHOLD, then value * LARGE_TRUCK_TAX_RATE is added to get the total use tax.
- `toString`: The toString method required in the Truck class should invoke the toString method in Vehicle and then append a line to the result. The line to be appended consists of the tax or taxes used in the calculation of the use tax. Below are examples of the toString result for truck1 and truck2 as declared above.

```
Williams, Jo: Truck 2012 Chevy Silverado
Value: $30,000.00 Use Tax: $600.00
with Tax Rate: 0.02

Williams, Sam: Truck 2010 Chevy Mack (Alternative Fuel)
Value: $40,000.00 Use Tax: $1,600.00
with Tax Rate: 0.01 Large Truck Tax Rate: 0.03
```

Code and Test: As you implement the Truck 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 Truck in a JUnit test method in the TruckTest 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 VehiclesPart1 (page 8) in parallel with this class to aid in testing.

- **SemiTractorTrailer.java**

Requirements: Derive the SemiTractorTrailer class from the Truck class.

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

(1) **Field**: an instance variable for axles of type int, which should be declared with the *protected* access modifier; one public constant of type double: PER_AXLE_TAX_RATE = 0.005. These are the only fields that should be declared in this class.

(2) **Constructor**: The SemiTractorTrailer class must contain a constructor that accepts six parameters representing the four instance fields in the Vehicle class, one instance field from Truck, and one instance field from SemiTractorTrailer. Since this class is a subclass of

Vehicle, the super constructor should be called with field values for Truck. The instance variable for axles should be set with the last parameter of the SemiTractorTrailer constructor. Below is an example of how the constructor could be used to create a Truck object:

```
SemiTractorTrailer semi1 = new SemiTractorTrailer("Williams, Pat",
                                  "2012 International Big Boy",
                                  45000, false, 5.0, 4);
```

(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 method.

- `getAxles`: Accepts no parameters and returns a int representing the axles field.
- `setAxles:` Accepts a int representing the axles, sets the field, and returns nothing.
- `useTax`: Accepts no parameters and returns a double representing the total use tax calculated as follows. Calls the useTax method in Truck and then adds value * PER_AXLE_TAX_RATE * axles, then returns the result as the total use tax for a SemiTractorTrailer.
- `toString`: The toString method required in the SemiTractorTrailer class should invoke the toString method in Truck and then append " Axle Tax Rate: " followed by the rate calculated as PER_AXLE_TAX_RATE * axles. Below are examples of the toString result for truck1 and truck2 as declared above.

```
Williams, Pat: SemiTractorTrailer 2012 International Big Boy
Value: $45,000.00 Use Tax: $3,150.00
with Tax Rate: 0.02 Large Truck Tax Rate: 0.03 Axle Tax Rate: 0.02
```

Code and Test: As you implement the SemiTractorTrailer 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 in SemiTractorTrailerTest. You may also consider developing VehiclesPart1 below in parallel with this class to aid in testing. For more details, see **Code and Test** above for the Car and Truck classes.

- **Motorcycle.java**

Requirements: Derive the Motorcycle class from the Vehicle class.

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

(1) **Field**: an instance variable for engine size of type double, which should be declared with the *protected* access modifier; four public constants of type double: TAX_RATE = 0.005, ALTERNATIVE_FUEL_TAX_RATE = 0.0025, LARGE_BIKE_CC_THRESHOLD = 499, and LARGE_BIKE_TAX_RATE = .015. These are the only fields that should be declared in this class.

(2) **Constructor**: The Motorcycle class must contain a constructor that accepts five parameters representing the four instance fields in the Vehicle class and one instance field from Motorcycle. Since this class is a subclass of Vehicle, the super constructor should be called with field values for Vehicle. The instance variable for engine size should be set with the last parameter of the Motorcycle constructor. Below is an example of how the constructor could be used to create a Motorcycle object.

```
Motorcycle bike1 = new Motorcycle("Brando, Marlon",
                                  "1964 Harley-Davidson Sportster",
                                   14000, false, 750);
```

(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.

- `getEngineSize`: Accepts no parameters and returns a double representing the engine size field.
- `setEngineSize`: Accepts a double representing the engine size, sets the field, and returns nothing.
- `useTax`: Accepts no parameters and returns a double representing the total use tax calculated as follows. If the motorcyle uses alternative fuel, it's calculated as value * ALTERNATIVE_FUEL_TAX_RATE; otherwise it's calculated as value * TAX_RATE. Furthermore, if the engine size exceeds LARGE_BIKE_CC_THRESHOLD, then value * LARGE_BIKE_TAX_RATE is added to get the total use tax.
- `toString`: The toString method required in the Motorcycle class should invoke the toString method in Vehicle and then append a line to the result. The line to be appended consists of the tax or taxes used in the calculation of the use tax. Below is an example of the toString result for bike1 as declared above.

```
Brando, Marlon: Motorcycle 1964 Harley-Davidson Sportster
Value: $14,000.00 Use Tax: $280.00
with Tax Rate: 0.005 Large Bike Tax Rate: 0.015
```

Code and Test: As you implement the Motorcycle 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 in MotorcycleTest. You may also consider developing VehiclesPart1 below in parallel with this class to aid in testing.

- **VehiclesPart1.java**

Requirements: Driver class with main method.

Design: The VehiclesPart1 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 Vehicle and Car, you can add statements to main that create and print an instance of Car. [Since Vehicle is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print one or more instances of Car, Truck,

SemiTractorTrailer, and Motorcycle. Since printing the objects does not show the actual of the fields, you should also run VehiclesPart1 in the canvas (or debugger with a breakpoint) to examine the objects while the program is running. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create Car car1, as described above and your main method is stopped between steps after Car car1 has been created, you can enter the following in interactions to calculate the use tax for the Car object.

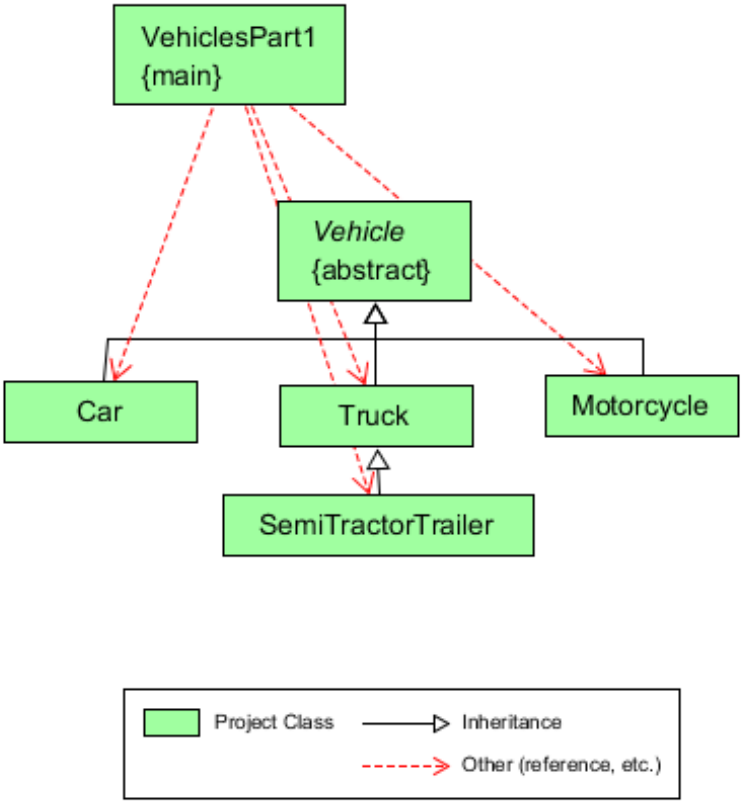
```
car1.useTax()
220.0
```

Code and Test: After you have implemented the VehiclesPart1 class, you should create the test file VehiclesPart1Test.java in the usual way. The only test method you need is one that creates an instance of VehiclesPart1 (to cover its default constructor), and then checks the class variable v*ehicleCount* that was declared in Vehicle and inherited by each subclass. In the test method, you should declare and create an instance of VehiclesPart1, reset v*ehicleCount*, call your main method, then assert that v*ehicleCount* is eight (assuming that your main creates eight objects from the Vehicle hierarchy). The following statements accomplish the test.

```
VehiclesPart1 vp1 = new VehiclesPart1();
Vehicle.resetVehicleCount();
VehiclesPart1.main(null);
Assert.assertEquals("Vehicle.vehicleCount should be 8. ",
                    8, Vehicle.getVehicleCount());
```

UML Class Diagram

As you add your classes to the jGRASP project, you should generate the UML class diagram for the project. To layout the UML class diagram, right-click in the UML window and then click **Layout** > **Tree Down**. Click in the background to unselect the classes. You can then select the VehiclesPart1 class and move it around as appropriate. Below is an example. Note that the dependencies represented by the red dashed arrows indicate that VehiclesPart1 references each of the subclasses of Vehicle (i.e., the main method in VehiclesPart1creates instances of each subclass and prints then out).



Canvas for VehiclesPart1

Below is an example of a jGRASP viewer canvas for VehiclesPart1 that contains two viewers for each of the variables car1, truck1, semi1, and bike1. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. A viewer for the class variable Vehicle.vehicleCount is near the bottom of the canvas. 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 *vehicleCount*. To display types with the labels, click **View** on the canvas top menu bar then turn on **Show Types in Viewer Labels** with the check box. Since you are printing the objects in main, you can see the toString results in the Run I/O window. So a single canvas with all objects in the Basic viewer may be more useful than the toString viewer.