◀ Project 2, Part 1: Wolf Travel Tours

Project 2, Part 2: Wally's Service Garage

Part 1 of this assignment laid out the requirements for the Wally's Service Garage application that you must create. Part 2 details the design that you must implement. For this part, you must create a complete Java program consisting at multiple source code files and JUnit test cases. You must electronically submit your files to NCSU GitHub by the due date.

Process Points 1 Deadline: Friday, July 12 at 11:45PM

For Process Points 1, you are expected to have a code skeleton implemented so that your project will compile against the teaching staff tests. Nothing has to run or pass.

Process Points 2 Deadline: Wednesday, July 17 at 11:45PM

For Process Points 2, you are expected to have enough of your own tests that the teaching-staff tests will run. They don't need to pass, but you need 80% coverage across all classes, no JUnit-related PMD alerts, and all of your tests passing.

Part 2 Due Date: Friday, July 19 at 11:45PM

Late Deadline: Sunday, July 21 at 11:45PM

Project 2 Part 2 MUST be completed individually or with your assigned partner.

Set Up Your Work Environment

Before you think about any code, prepare your work environment to be compatible with the teaching staff grading criteria. The teaching staff relies on NCSU GitHub and Jenkins for grading your work. Follow these steps:

1. Create an Eclipse project named Project2: Right click in the package explorer, select New > Java Project. Name the project Project2. (The project naming is case sensitive. Be sure that the project name begins with a capital P.)
2. If you have not already done so, clone your remote NCSU GitHub repository corresponding to this project. Note that each major project will have its own repository.

Note

If you were assigned a partner for Project 2 Part 2, you may only work with your assigned partner. Otherwise, you are expected to complete the project individually. All work must be strictly your own or you and your partner's own work.

Working on a Pair

If you are working with a partner do the following to make sure that you both are working on the same project and to avoid collisions.

1. Identify who will be partner A and partner B.
2. Partner A should create the Eclipse project as described above. Then partner A will clone the team repo and share Project 2 to the locally copy of the clone repo. Then Partner A should commit/push the shared project to NCSU's GitHub. Verify that the project is there by navigating to the repo on the GitHub website. The project should be listed as Project2 and should contain the src/ folder, a .classpath, and .project files (at a minimum).
3. Partner B should clone the repo AFTER Partner A has pushed the project. Then Partner B will import the existing project into their Eclipse workspace through the GitHub repositories view.
4. Communicate frequently to avoid merge conflicts. If you have a merge conflict see the NCSU Git Guide for help.

Design

Your program must implement our design, which we describe here. We are providing the Manageable interface, a simple console-based user interface that you can use for integration testing of the back-end, and a custom dialog class. You must create all other files, including the GUI front end.

The classes in your project are listed below under their respective packages.

*edu.ncsu.csc216.garage.ui

```
*[RepairGarageConsole](assets/part2/RepairGarageConsole.java). A console-based user interface that we supply for you initia

*`RepairGarageGUI`. The graphical user interface for the program. **You must write the code for this.**

*[UserDialog](assets/part2/UserDialog.java). A custom JDialog that we supply and that you can _optionally_ use as part of t

*`edu.ncsu.csc216.garage.model.dealer`
```

*Manageable. Interface that describes the behaviors of the model that the view (GUI) can use. **You cannot change this code in any way**. RepairGarageGUI can have only one instance variable from the back-end, and its declared type must be Manageable.

```
*`ServiceManager.` Concrete class that implements `Manageable`.
```

*edu.ncsu.csc216.garage.model.service_garage

```
*`ServiceBay.` Concrete class representing a service bay in the garage. Service bays of this type are considered "regular,'

    *`HybridElectricBay`. Concrete class that extends `ServiceBay` and represents a service bay that can accommodate hybric

*`NoAvailableBayException`. Thrown when a vehicle attempts to choose a service bay but all appropriate service bays are occ

*`BayOccupiedException`. Thrown when a vehicle attempts to choose a service bay that is occupied.

*`BayCarMismatchException`. Thrown when a vehicle attempts to choose a service bay that is empty but is of the wrong type f
```

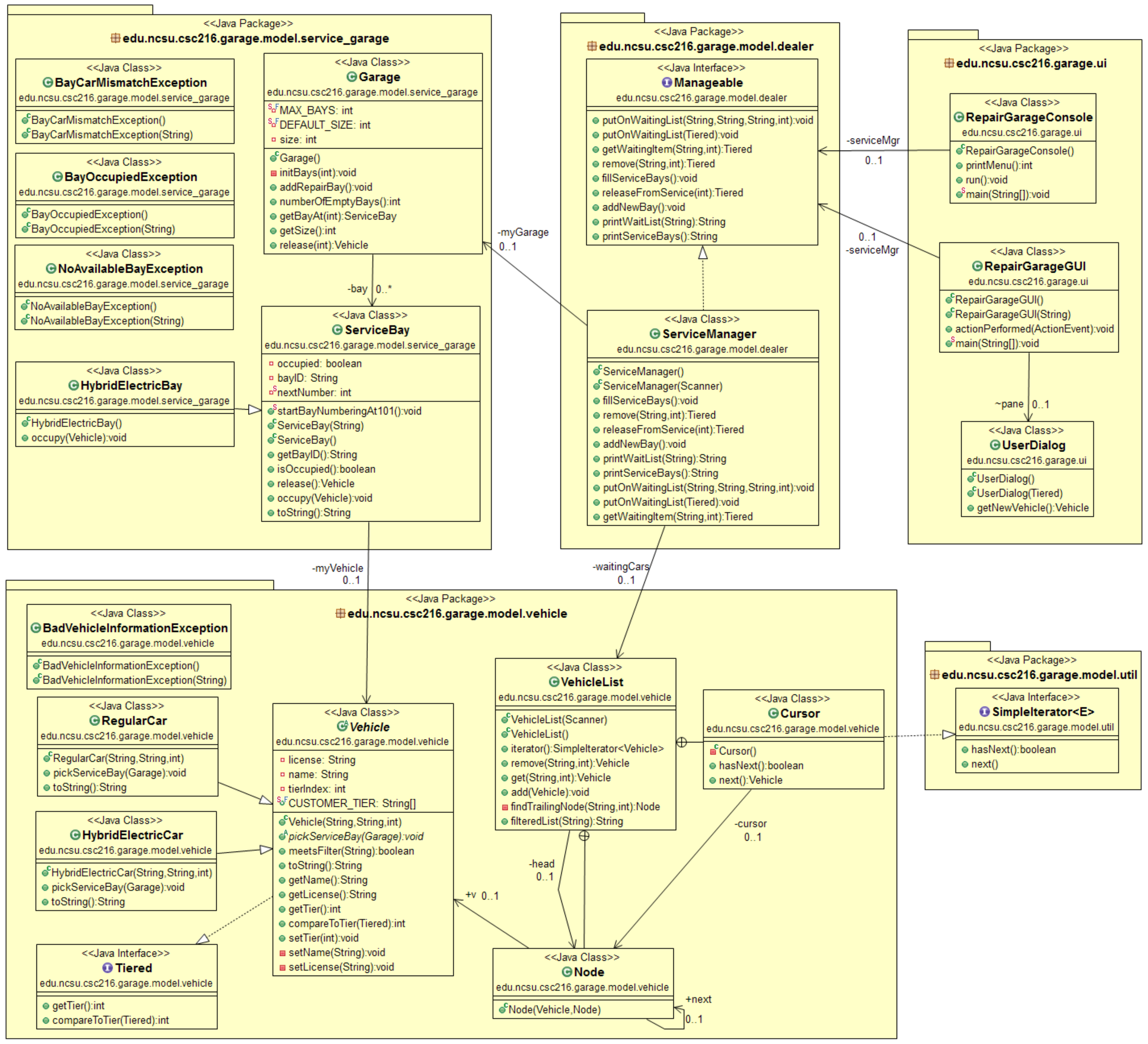
*edu.ncsu.csc216.garage.model.vehicle

```
*`Tiered`. Interface that describes behaviors of objects that can be ordered by tiers (0, 1, 2 and so on, with higher tiers

*`Vehicle.` Abstract class representing a vehicle appropriate for servicing in the garage.

    *`RegularCar`. Concrete class representing an ordinary gasoline or diesel vehicle.

    *`HybridElectricCar`. Concrete class representing a hybrid or electric car.

*`VehicleList`. List of vehicles waiting for service.

*`BadVehicleInformationException`. Thrown when a user attempts to add/edit a vehicle with invalid information (tier, type,
```

*edu.ncsu.csc216.garage.model.util

```
*`SimpleIterator<E>`. Interface describing behaviors of a generic type of iterator with two methods: `next()` and `hasNext(
```

The figure below is the UML class diagram for the design. Your project must keep the same directory structure as discussed above and illustrated below. You can modify the names of parameters and private fields. The public methods and constants (names, return types, parameter types and order) ***must be exactly*** as shown below for the teaching staff tests to run. (Note that this diagram hides the private ServiceGarageGUI and ServiceGarageConsole members.)


<<Java Package>>
edu.ncsu.csc216.garage.model.service_garage
<<Java Class>>
BayCarMismatchException
edu.ncsu.csc216.garage.model.service_garage
BayCarMismatchException()
BayCarMismatchException(String)
<<Java Class>>
BayOccupiedException
edu.ncsu.csc216.garage.model.service_garage
BayOccupiedException()
BayOccupiedException(String)
<<Java Class>>
NoAvailableBayException
edu.ncsu.csc216.garage.model.service_garage
NoAvailableBayException()
NoAvailableBayException(String)
<<Java Class>>
HybridElectricBay
edu.ncsu.csc216.garage.model.service_garage
HybridElectricBay()
occupy(Vehicle):void
<<Java Class>>
Garage
edu.ncsu.csc216.garage.model.service_garage
MAX_BAYS: int
DEFAULT_SIZE: int
size: int
Garage()
initBays(int):void
addRepairBay():void
numberOfEmptyBays():int
getBayAt(int):ServiceBay
getSize():int
release(int):Vehicle
-bay
0..*
<<Java Class>>
ServiceBay
edu.ncsu.csc216.garage.model.service_garage
occupied: boolean
bayID: String
nextNumber: int
startBayNumberingAt101():void
ServiceBay(String)
ServiceBay()
getBayID():String
isOccupied():boolean
release():Vehicle
occupy(Vehicle):void
toString():String
-myGarage
0..1
<<Java Package>>
edu.ncsu.csc216.garage.model.dealer
<<Java Interface>>
Manageable
edu.ncsu.csc216.garage.model.dealer
putOnWaitingList(String,String,String,int):void
putOnWaitingList(Tiered):void
getWaitingItem(String,int):Tiered
remove(String,int):Tiered
fillServiceBays():void
releaseFromService(int):Tiered
addNewBay():void
printWaitList(String):String
printServiceBays():String
<<Java Class>>
ServiceManager
edu.ncsu.csc216.garage.model.dealer
ServiceManager()
ServiceManager(Scanner)
fillServiceBays():void
remove(String,int):Tiered
releaseFromService(int):Tiered
addNewBay():void
printWaitList(String):String
printServiceBays():String
putOnWaitingList(String,String,String,int):void
putOnWaitingList(Tiered):void
getWaitingItem(String,int):Tiered
-serviceMgr
0..1
0..1
-serviceMgr
<<Java Package>>
edu.ncsu.csc216.garage.ui
<<Java Class>>
RepairGarageConsole
edu.ncsu.csc216.garage.ui
RepairGarageConsole()
printMenu():int
run():void
main(String[]):void
<<Java Class>>
RepairGarageGUI
edu.ncsu.csc216.garage.ui
RepairGarageGUI()
RepairGarageGUI(String)
actionPerformed(ActionEvent):void
main(String[]):void
~pane
0..1
<<Java Class>>
UserDialog
edu.ncsu.csc216.garage.ui
UserDialog()
UserDialog(Tiered)
getNewVehicle():Vehicle
-myVehicle
0..1
-waitingCars
0..1
<<Java Package>>
edu.ncsu.csc216.garage.model.vehicle
<<Java Class>>
BadVehicleInformationException
edu.ncsu.csc216.garage.model.vehicle
BadVehicleInformationException()
BadVehicleInformationException(String)
<<Java Class>>
RegularCar
edu.ncsu.csc216.garage.model.vehicle
RegularCar(String,String,int)
pickServiceBay(Garage):void
toString():String
<<Java Class>>
HybridElectricCar
edu.ncsu.csc216.garage.model.vehicle
HybridElectricCar(String,String,int)
pickServiceBay(Garage):void
toString():String
<<Java Interface>>
Tiered
edu.ncsu.csc216.garage.model.vehicle
getTier():int
compareToTier(Tiered):int
<<Java Class>>
Vehicle
edu.ncsu.csc216.garage.model.vehicle
license: String
name: String
tierIndex: int
CUSTOMER_TIER: String[]
Vehicle(String,String,int)
pickServiceBay(Garage):void
meetsFilter(String):boolean
toString():String
getName():String
getLicense():String
getTier():int
compareToTier(Tiered):int
setTier(int):void
setName(String):void
setLicense(String):void
+v 0..1
<<Java Class>>
VehicleList
edu.ncsu.csc216.garage.model.vehicle
VehicleList(Scanner)
VehicleList()
iterator():SimpleIterator<Vehicle>
remove(String,int):Vehicle
get(String,int):Vehicle
add(Vehicle):void
findTrailingNode(String,int):Node
filteredList(String):String
-head
0..1
<<Java Class>>
Cursor
edu.ncsu.csc216.garage.model.vehicle
Cursor()
hasNext():boolean
next():Vehicle
-cursor
0..1
<<Java Class>>
Node
edu.ncsu.csc216.garage.model.vehicle
Node(Vehicle,Node)
+next
0..1
<<Java Package>>
edu.ncsu.csc216.garage.model.util
<<Java Interface>>
SimpleIterator<E>
edu.ncsu.csc216.garage.model.util
hasNext():boolean
next()

The VehicleList class has two private inner classes: Node and Cursor. You can rename these inner classes as you wish. They are used in the teaching staff solution for a linked list node and an implementation of the SimpleIterator<E> interface, respectively, where cursor is the current location of the iterator within the list.

Important: You cannot declare any private data or methods in the UML diagram to public or protected. You cannot add any public or protected data or methods. You are allowed add private member data or private methods to any class.

In the rest of this Design section are explanations of methods and data that require more explanation than what is shown in the UML class diagram and Part 1 of this assignment. Ordinary setters and getters are not listed.

Extra documentation for edu.ncsu.csc216.garage.model.service_garage``

ServiceBay
startBayNumberingAt101(). Static method. Resets the service bay numbering to start from 1 (so the next 3 bays added for this project would have numbers/IDs E01, 102, and 103). This is a convenience mostly for JUnit testing, but it can optionally be called by the Garage constructor.
ServiceBay(). Creates a new empty service bay according to the current bay numbering, then increments that number. The prefix is "1". [UC6, S2], [UC6, S3]
ServiceBay(String prefix). Same as null constructor but the prefix is the first non-whitespace character in the argument or "1" if there is no such character. [UC6, S3]
occupy(Vehicle v). Occupies the service bay with the given vehicle. Throws a BayOccupiedException if the service bay is already occupied. [UC7, S1]
release(). Removes the vehicle currently in the service bay and returns it. [UC8, S2]
toString(). Formats the string representation of the service bay. [UC7, S4]

HybridElectricBay extends ServiceBay
HybridElectricBay(). Creates a new empty bay for servicing hybrid/electric vehicles according to the current bay numbering, then increments that number. The prefix is "E". [UC6, S1], [UC6, S3]
occupy(Vehicle v). Occupies the service bay with the given vehicle. Throws a BayCarMismatchException if the vehicle is not a hybrid/electric car. [UC7, S1]

Garage.
Garage(). Creates a list of 8 empty service bays. [UC1, S7]
addRepairBay(). Adds a repair bay. At least 1/3 of the bays in the garage are dedicated to hybrid/electric vehicles. [UC6]
numberOfEmptyBays(). Number of open service bays that are currently empty. [UC7, S1]
getBayAt(int x). Service bay at the given index. [UC7, S1], [UC8, S2]
getSize(). Total number of open service bays. [UC6, S1] [UC6, S2]

release(int x). See ServiceBay.release().

NoAvailableBayException. Thrown when a vehicle attempts to enter the garage for service but no appropriate bays are open. [UC7, S1]
BayOccupiedException. Thrown when a vehicle attempts to occupy a non-empty, open service bay. [UC7, S1]
BayCarMismatchException. See ServiceBay.occupy().

Extra documentation for edu.ncsu.csc216.garage.model.vehicle````

Vehicle. Abstract class.
Vehicle(String license, String owner, int status). Constructor that creates a vehicle out of a license, owner name, and tier status. [UC1, S1], [UC2]. Throws a BadVehicleInformationException if the license or owner are not valid. [UC2, E2], [UC2, E3], [UC2, E4]
Vehicle(String info, int status). Constructor that creates a vehicle out of string representing the license followed by the name, then the tier status level.
pickServiceBay(Garage). Abstract. Throws NoAvailableBayException. The vehicle picks a service bay. [UC7, S1]
meetsFilter(String filter). True if filter is a prefix to the owner's last name. The check is case insensitive. A filter of null or the null string (e.g., "") would return true. [UC3, S3], [UC3, E1], [UC4]
toString(). String representation of the vehicle. [UC1, S6], [UC2, S4], [UC3]

RegularCar. Concrete class representing a vehicle that cannot be serviced in HybridElectricBays.
pickServiceBay(Garage). Goes through the list of service bays, starting at the front, searching for an empty one. [UC7, S1]
toString(). String representation of the vehicle, prefixed by "R ". [UC1, S6], [UC2, S4], [UC3]

HybridElectricCar. Concrete class representing a vehicle that can be serviced in HybridElectricBays.
pickServiceBay(Garage)). Goes through the list of service bays, starting at the end, searching for an empty one. [UC7, S1]
toString(). String representation of the vehicle prefixed by "E ". [UC1, S6], [UC2, S4], [UC3]

Tiered. Interface that describes behaviors of objects with tier status. It has two methods:
compareTo(Tiered). Compare the tier status of this object with another. Returns 0 if the two match, a negative number if the tier status of this object is less than the other's, a positive number if the tier status of this object is greater. Required for ordering. [UC1, S5], [UC2, S4]

VehicleList. Maintains its vehicles in a linked list.
VehicleList(Scanner). Creates a list of vehicles from a Scanner. The Scanner would have been initialized by the calling code as a Scanner on an input text file. [UC1, S3]
VehicleList(). Creates an empty list of vehicles. [UC1, E1]
iterator(). Returns a SimpleIterator initialized to point to the first element in the list. [UC7, S1].

remove(String filter, int position). Removes the vehicle that appears in the filtered list in the given position. (See Vehicle.meetsFilter().) [UC5]
add(Vehicle). Adds the given vehicle to the list of those awaiting service. [UC2, S4]
filteredList(String filter). String representation of all vehicles that meet the filter. Each substring corresponding to a vehicle is terminated by a newline. [UC3, S3]

Special constraints for edu.ncsu.csc216.garage.model.dealer``

ServiceManager. Implements Manageable and contains both a Garage and a VehicleList.
ServiceManager(). Creates a service manager with no vehicles awaiting service. [UC1, E1]
ServiceManager(Scanner s). Initializes the list of vehicles awaiting service with data from the Scanner. [UC1, S1], [UC1,S2], [UC1, S4]
fillServiceBays(). Fills the open, empty service bays with vehicles on the waiting list. [UC7]
remove(String, int). See VehicleList.remove(String, int).
releaseFromService(int). See Garage.release(int).
addNewBay(). See Garage.addRepairBay().
printWaitList(String). See VehicleList.filteredList(String).
printServiceBays(). String representation of open service bays. [UC1], [UC6], [UC7], [UC8]
putOnWaitingList(String, String, String, int). Puts a vehicle on the list of vehicles awaiting service. The first parameter deterimines regular or hybrid/electric. The second is the license, third is owner name, and fourth is the tier status. (See Vehicle(String, String, int).) [UC1, UC2]
putOnWaitingList(Tiered). Puts a vehicle on the list of vehicles awaiting service. [UC1] or [UC2]

UML Diagram Notations

UML uses standard conventions for display of data, methods, and relationships. Many are illustrated in the UML diagram above. Here are some things you should note:

- in front of a class member means private.
+ in front of a class member means public.
# in front of a class member means protected.
Static members (data or methods) are underlined.
Methods that are declared but not defined (abstract methods or those declared in interfaces) are in italics.
The names of abstract classes are in italics.
Dotted arrows with triangular heads point to interfaces from the classes that implement them.
Solid arrows with triangular heads go from concrete classes to their parents.

Solid arrows with simple heads indicate *has-a* relationships (composition). The containing class is at the tail of the arrow and the class that is contained is at the head. The arrow is decorated with the name and access of the member in the containing class. The arrow is also decorated with the "multiplicity" of the relationship, where 0..1 means there is 1 instance of the member in the containing class. A multiplicity of 0..* means there are many, usually indicating a collection such as an array or collection class.
A circle containing an "X" or cross sits on a the border of a class that contains an inner (nested) class, with a line connecting it to the inner class.

Our UML diagram has some additional graphic notation:

A red square (empty or solid) in front of a name means private. Solid squares are methods and empty squares are data members.
A green circle (empty or solid) in front of a name means public.
A yellow diamond (empty or solid) in front of a name means protected.
SF in front of a name means static, final.
Methods embellished with C are constructors. Private methods embellished with a C are constructors in private inner classes.
Note that while all classes require a constructor, not all constructor require implementation. In some cases the default constructor is sufficient.

Access Modifiers

Note: You can modify the names of private variables, parameters, and methods. However, you MUST have the non-private data and methods (names, return types, parameter types and order) exactly as shown for the teaching staff tests to run. YOU MAY NOT ADD ANY ADDITIONAL PUBLIC OR PROTECTED CLASSES, METHODS, OR STATES.

Implementation Process

Every Java file for the backend code (model) for this assignment must be in the package edu.ncsu.csc216.garage.model.*, where * is a sub-package for different parts of the system as described above. Every Java file for the user interface must be in the package edu.ncsu.csc216.garage.ui.

Compile a skeleton. The class diagram provides a full skeleton of what the implemented issue tracker program should look like. Start by creating an Eclipse project named **Project2.** Copy in provided code and create the skeletons of the classes you will implement. Ensure that the packages, class names, and method signatures match the class diagram *exactly!* If a method has a return type, put in a place holder (for example, return 0; or return null;) so that your code will compile. Push to GitHub and make sure that your Jenkins job shows a yellow ball. A yellow ball on Jenkins means that 1) your code compiles and 2) that the teaching staff tests compile against your code, which means that you have the correct code skeleton for the design.

Compiling Skeleton

A compiling skeleton is due ***at least one week before*** the final project deadline to earn the associated process points.

Comment your code. Javadoc your classes and methods. When writing Javadoc for methods, think about the inputs, outputs, preconditions, and post conditions.

Fully Commented Classes

Fully commented classes and methods on at least a skeleton program are due ***at least one week before*** the final project deadline to earn the associated process points.

Commit with meaningful messages.

Meaningful Commit Messages

The quality of your commit messages for the entire project history will be evaluated for meaning and professionalism ***as part of the process points***.

Run your CheckStyle and PMD tools locally and make sure that all your Javadoc is correct. Make sure the tools are configured correctly (see configuration instructions) and set up the tools to run with each build so you are alerted to style notifications shortly after creating them.
Practice test-driven development. Start by writing your test cases. This will help you think about how a client would use your class (and will help you see how that class might be used by other parts of your program). Then write the code to pass your tests.
Save coding the GUI until the rest of your code is complete.
    Start functional testing through RepairGarageConsole.
    Save coding the GUI for last. And give yourself some time – GUI coding is both tricky and tedious!

Implementation

We suggest you attack Project 2 Part 2 one piece at a time. The details below provide the suggested implementation order and details about the implementation.

The sections here describe implementation hints and restrictions.

toString() methods

The toString() methods for Vehicle and ServiceBay need to follow the exact pattern demonstrated in the GUI images from Part 1 of this assignment:

ServiceBay.toString() returns a string in the form <bay id>: <license> <owner name> or EMPTY
Be sure to count leading or trailing blanks on the string tokens to get the correct spacing. Examples:

E01: EMPTY 102: NC1234 Doe, John

RegularCar.toString() returns a string in the form R <Tier> <license> <owner name>. HybridElectricCar.toString() uses the same format, except the initial character is E instead of R.
Be sure to count leading or trailing blanks on the string tokens to get the correct spacing. Examples:

R Gold NC123456 Doe, Jane E Platinum 79-27DC Carter, June W.

In each case, the returned strings have no leading or trailing whitespace.

List implementations

For this project, you must create two lists, one for the service bays and the other for the vehicles in the awaiting service.

You must implement the list of service bays in the garage as an array-based list of ServiceBays. You must create your own array-based list – _you cannot use any Java Collection Classes such as ArrayList or ArrayList._ Since the garage can contain at most 30 service bays, the array can be limited to 30 elements. `Garage.getBayAt()` enables client code to go through every open service bay in the garage.

You must implement the list of vehicles awaiting service as a linked list of Vehicles. You must create your own linked list – _you cannot use any Java Collection Classes such as LinkedList or LinkedList._ `SimpleIterator` enables client code to go through each vehicle in the waiting list. This is helpful, for example in filling the garage's open service bays.

Connecting the GUI to the model

The interface Manageable provides the *major* methods that the GUI or console can call on the model. Since two methods of Manageable return Tiered objects, you may also call methods of Vehicle, when obtained via Manageable. The console is already completed. The GUI should follow the same setup for its model instance variable. The GUI and console are responsible for associating file data with the model (in particular, initializing a scanner to the file and creating a new ServiceManager` via the scanner).

And a little help with the GUI

You can create any GUI that offers the functionality that is described in the project. It does not have to look the same as the one for the teaching staff solution, but your GUI should be as close as possible to the prototypes provided in Part 1. If you're having trouble getting started, look at the GUI code for the previous two projects. Some simple explanations of how to use GUI components are in the online course materials and the Java Swing Tutorial:

- Trail: Creating a GUI with JFC/Swing (Java Tutorials)
  - Using Swing Components (Java Tutorials)

We do offer the UserDialog code for anyone who wants to use it. Here is the call that our code uses inside its actionPerformed method, where e is the name of the ActionEvent and btnAdd is the *Add New Vehicle* button.

```
if (e.getSource().equals(btnAdd)) {
        UserDialog pane = new UserDialog();
        pane.setVisible(true);
        Vehicle v = pane.getNewVehicle();
        if (v != null)
                serviceMgr.putOnWaitingList(v);
        }
```

Testing

For Part 2 of this project, you must do white box testing via JUnit AND report the results of running your black box tests from Part 1.

White box testing

Your JUnit tests should follow the same package structure as the classes that they test. You need to create JUnit tests for *all* of the concrete classes that you create (even inner classes). At a minimum, you must exercise every method in your solution at least once by your JUnit tests. Start by testing all methods that are not simple getters, simple setters, or *simple* constructors for all of the classes that you must write and check that you are covering all methods. If you are not, write tests to exercise unexecuted methods. You can test the common functionality of an abstract class through a concrete instance of its child.

When testing void methods, you will need to call other methods that do return something to verify that the call made to the void method had the intended effects.

For each method and constructor that throws exceptions, test to make sure that the method does what it is supposed to do and throws an exception when it should.

At a minimum, you must exercise at least 80% of the statements/lines in all ***non-GUI*** classes. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. You must have 95% method coverage to achieve a green ball on Jenkins (assuming everything else is correct).

We recommend that you try to achieve 100% condition coverage (where every conditional predicate is executed on both the true and false paths for all valid paths in a method).

Black box testing and submission

Use the provided black box test plan template to describe your tests. Each test must be repeatable and specific; all input and expected results values must be concrete. All inputs required to complete the test must be specified either in the document or the associated test file must be submitted. Remember to provide instructions for how a tester would set up, start, and run the application for testing (What class from your design contains the main method that starts your program? What are the command line arguments, if any? What do the input files contain?). The instructions should be described at a level where anyone using Eclipse could run your tests.

If you are planning for your tests to read from or write to files, you need to provide details about what the files so that the teaching staff could recreate them or find them quickly in your project.

Follow these steps to complete submission of your black box tests:

1. Run your black box tests on your code and report the results in the Actual Results column of your BBTP document.
2. Save the document as a pdf named **BBTP_P2P2.pdf**.
3. Create a folder named **project_docs** at the top level in your project and copy the pdf to that folder.
4. Push the folder and contents to your GitHub repository.

Deployment

For this class, deployment means submitting your work for grading. Submitting means pushing your project to NCSU GitHub.

Before considering your work complete, make sure:

1. Your program satisfies the style guidelines.
2. Your program behaves as specified in this document. You should test your code thoroughly. Be sure to know what messages should be displayed for each major scenario.
3. Your program satisfies the gradesheet. You can estimate your grade from your Jenkins feedback.
4. You generate javadoc documentation on the most recent versions of your project files.

5. You push any updated project_docs, doc, and test-files folders to GitHub.

Deadline

The electronic submission deadline is precise. Do not be late. You should count on last minute failures (your failures, ISP failures, or NCSU failures). Push early and push often!

◀ Project 2, Part 1: Wolf Travel Tours