TCSS 305 Programming Practicum

Programming Assignment #04

Revision History
Rev.0 – (11:50 pm on Wed, Dec 1st, 2021) Initial version.

Please post your general questions about this assignment on the `Ed Discussion > Programming Assignments > #NN` and respond to your peer's questions on this assignment.

Study Groups:

- Please form 2-3 people study groups to work on this assignment together and register your group on the `Canvas > People` page.
- You can exchange ideas, discuss the requirements and alternative solutions, ask questions to each other about a specific code segment that doesn't compile or run as you expected, and share helpful online or printed resources with your study group. However, you cannot share your working source codes and final technical solution with your study group members.

Guidelines for Submissions:

[1] Submit your report containing your answers in a single file, named "`PRGA#04-lastname_firstname.pdf`."

You may paste all the required snapshots into your Word document, convert it into pdf document (`File > Save As > File Format: pdf`), and upload your pdf file into Canvas. Canvas will not allow you to submit your report in other file types.

[2] Attach each IntelliJ IDEA project as a separate zip file, named "`PRGA#04-lastname-<projectname>.zip`" to your submission's Assignment Comments after submitting your report.

- In IntelliJ, choose from the main menu `File | Export | Project to Zip file ...` to export the IntelliJ IDEA project folder containing your source codes, as well as your test codes, other required files for your program (such as image and data files, if any) as a compressed single zip file.
- See the instructions about choosing proper file names on the first page.
- Please verify your project zip file before submitting it. We should be able to compile and run your program without any problems after unzipping your zip file into another folder and importing it again into IntelliJ IDEA as a new project.

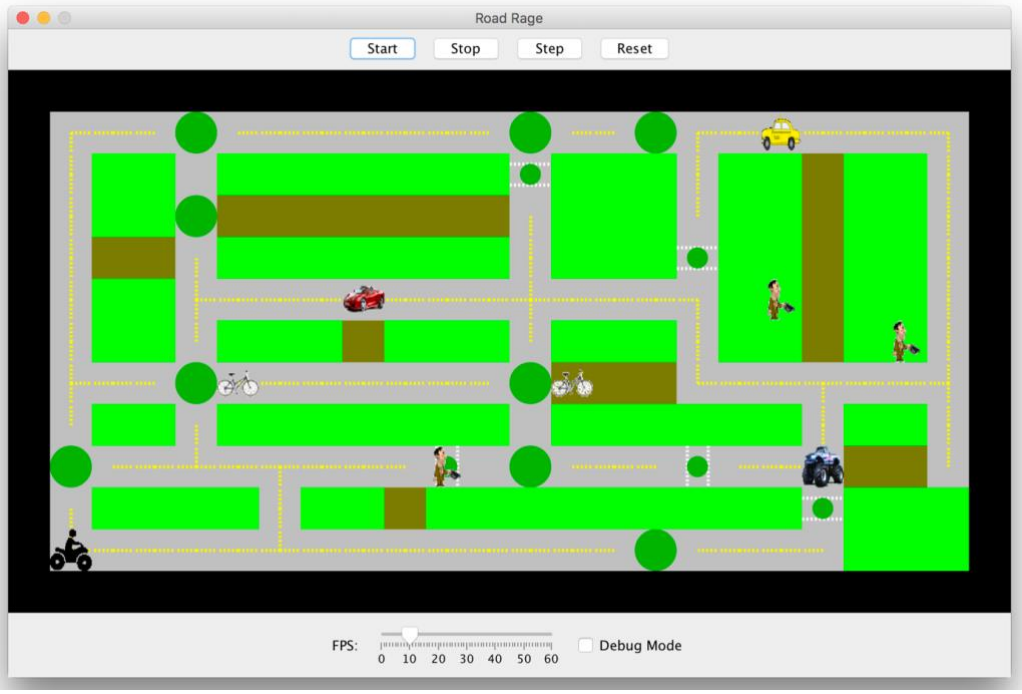
[3] DON'T FORGET to turn in the corresponding survey (if one posted) for this assignment after submitting your assignment.
[4] You can make up to **three (3) attempt/submissions** before the deadline; only your latest submission will be graded unless you request otherwise by leaving a note in the *Assignment Comments* box.
[5] Keep your responses brief, concise, clear, and focused on the essence of the question or problem.
[6] Use multiple sections with focused subtopics instead of presenting everything in one long paragraph.
[7] Add visuals (hand-drawn or computer-drawn pictures, drawings, models, etc.) to your answers to highlight important concepts. Add short labels to such entities to explain what they are about.
[8] The text should be appropriately formatted with headers, indents, colors, underlines/bolds, etc., and use bulleted/numbered lists to make your answers easily readable.
[9] All web links should show the full address of the web site and be clickable.

I. Problem Description:

You are going to develop a software application (*starting with the given code base)* that simulates the movements of vehicles, bicycles, and people in a segment of the city center (whose map is given below) composed of streets, bicycle paths, green areas, and traffic lights, and vehicles, bicycles, and people.

This assignment aims to improve your understanding of

(1) Java Graphics programming, (Core Java: Chapter 10, Section 10.3)
(2) Java GUI programming, (Core Java: Chapter 10 & 11)
(3) as well as inheritance, polymorphism, interfaces, and abstract classes.

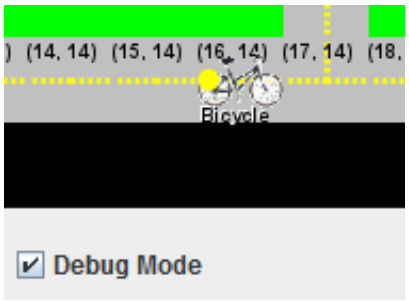


On the above picture, the black frame indicates the border of the area of interest. The green areas are indicated by light green color, bicycle paths are indicated by dark green color, grey areas marked with yellow dashed lanes are the streets, and the colored circles represent the traffic lights, respectively.

The given starter code for the project includes a complete (and correct) graphical user interface (GUI).

- The GUI shows pictures for the map terrain and vehicles. The GUI loads the city map and vehicle starting positions from the provided file `city_map.txt`.

- There are buttons to start and stop the city's animation, as well as to step through it one frame at a time and to reset the vehicles to their initial layout.
- The rate of city animation is controllable via a slider at the bottom of the interface.
    - Each time the GUI redraws, it moves each of the vehicles. If any two vehicles end up on the same square, the GUI tells the vehicles that they have collided with each other, which may "kill" them temporarily. "Dead" vehicles appear upside down on the screen.
- Also, debug information can be shown using a checkbox.
    - If debug information is enabled, all (x, y) grid positions on the map are labeled, and every vehicle's `toString` output is shown next to the vehicle.
    - You can put any information you like into your vehicles' `toString` methods to help you debug their behavior.



Your task is to write the classes to model the various vehicles in the city. The classes will have many similarities, so you should use inheritance to design the classes effectively.

- You must write `Vehicle` sub-classes (and a common parent class called `AbstractVehicle`) for this assignment.
- Each of the `Vehicle` classes must implement the provided `Vehicle` interface described in this document.
- Note that a Human is a "Vehicle" for the purposes of this assignment.
- Vehicle behavior includes some randomness. If you need to create an object of type Random, then your project code should only create a single object of type `Random` that all `Vehicle` objects can share.

2. Vehicle Descriptions:

Truck

- Constructor: `public Truck(int theX, int theY, Direction theDir)`
- Images: `truck.gif` (alive), `truck_dead.gif` (dead)
- Movement behavior:
    - Trucks travel only on streets and through lights and crosswalks.
    - Trucks randomly select to go straight, turn left, or turn right. As a last resort, if none of these three directions is legal (all not streets, lights, or crosswalks), the truck turns around.
    - Trucks drive through all traffic lights without stopping!
    - Trucks stop for red crosswalk lights but drive through yellow or green crosswalk lights without stopping.
- Collision behavior: A truck survives a collision with anything, living or dead.

Car

- Constructor: `public Car(int theX, int theY, Direction theDir)`
- Images: `car.gif` (alive), `car_dead.gif` (dead)
- Movement behavior:
    - Cars can only travel on streets and through lights and crosswalks.
    - A car prefers to drive straight ahead on the street if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible; as a last resort, it turns around.

  - Cars stop for red lights; if a traffic light is immediately ahead of the car and the light is red, the car stays still and does not move. It does not turn to avoid the light. When the light turns green, the car resumes its original direction.
  - Cars ignore yellow and green lights.
  - Cars stop for red and yellow crosswalk lights but drive through green crosswalk lights without stopping.
- Collision behavior: A car dies if it collides with a living truck and stays dead for 15 moves.

Taxi

- Constructor: `public Taxi(int theX, int theY, Direction theDir)`
- Images: `taxi.gif` (alive), `taxi_dead.gif` (dead)
- Movement behavior:
  - Taxis can only travel on streets and through lights and crosswalks.
  - A taxi prefers to drive straight ahead if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible; as a last resort, it turns around.
  - Taxis stop for red lights; if a traffic light is immediately ahead of the taxi and the light is red, the Taxi stays still and does not move until the light turns green. It does not turn to avoid the light. When the light turns green the taxi resumes its original direction. Taxis ignore yellow and green lights.
  - Taxis stop for (temporarily) red crosswalk lights. If a crosswalk light is immediately ahead of the taxi and the crosswalk light is red, the Taxi stays still and does not move for 3 clock cycles or until the crosswalk light turns green, whichever occurs first. It does not turn to avoid the crosswalk light. When the crosswalk light turns green, or after 3 clock cycles, whichever happens first, the taxi resumes its original direction. A Taxi will drive through yellow or green crosswalk lights without stopping.
- Collision behavior: A taxi dies if it collides with a living truck and stays dead for 15 moves.

All-Terrain Vehicle (ATV)

- Constructor: `public ATV(int theX, int theY, Direction theDir)`
- Images: `atv.gif` (alive), `atv_dead.gif` (dead)
- Movement behavior:
  - ATVs can travel on any terrain except walls.
  - They randomly select to go straight, turn left, or turn right.
  - ATV's never reverse direction (they never need to).
  - ATV's drive through all traffic lights and crosswalk lights without stopping!
- Collision behavior: An ATV dies if it collides with a living truck, car, or taxi, and stays dead for 25 moves.

Bicycle

- Constructor: `public Bicycle(int theX, int theY, Direction theDir)`
- Images: `bicycle.gif` (alive), `bicycle_dead.gif` (dead)
- Movement behavior:
  - Bicycles can travel on streets and through lights and crosswalk lights, but they prefer to travel on trails.
  - If the terrain in front of a bicycle is a trail, the bicycle always goes straight ahead in the direction it is facing. Trails are guaranteed to be straight (horizontal or vertical) lines that end at streets, and you are guaranteed that a bicycle will never start on a trail facing terrain it cannot traverse.

  - If a bicycle is not facing a trail, but there is a trail either to the left or to the right of the bicycle's current direction, then the bicycle turns to face the trail and moves in that direction. You may assume that the map is laid out so that only one trail will neighbor a bicycle at any given time.
  - If there is no trail straight ahead, to the left, or to the right, the bicycle prefers to move straight ahead on a street (or light or crosswalk light) if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible. As a last resort, if none of these three directions is legal (all not streets or lights or crosswalk lights), the bicycle turns around.
  - Bicycles ignore green lights.
  - Bicycles stop for yellow and red lights; if a traffic light or crosswalk light is immediately ahead of the bicycle and the light is not green, the bicycle stays still and does not move unless a trail is to the left or right. If a bicycle is facing a red or yellow light and there is a trail to the left or right, the bicycle will turn to face the trail.
- Collision behavior: A bicycle dies if it collides with a living truck, car, taxi, or ATV. It stays dead for 35 moves.

Human

- Constructor: `public Human(int theX, int theY, Direction theDir)`
- Images: **`human.gif`** (alive), **`human_dead.gif`** (dead)
- Movement behavior:
  - Humans move in a random direction (straight, left, or right), always on grass or crosswalks.
  - A human never reverses direction unless there is no other option.
  - If a human is next to a crosswalk it will always choose to turn to face in the direction of the crosswalk. (The map of terrain will never contain crosswalks that are so close together that a human might be adjacent to more than one at the same time.)
  - Humans do not travel through crosswalks when the crosswalk light is green. If a human is facing a green crosswalk, it will wait until the light changes to yellow and then cross through the crosswalk. The human will not turn to avoid the crosswalk.
  - Humans travel through crosswalks when the crosswalk light is yellow or red.
- Collision behavior: A human dies if it collides with any living vehicle except another human and stays dead for 45 moves.

3. Implementation Guidelines:

3.1 Interactions of Vehicles:

The GUI's interaction with your various vehicle classes is the following:

- When the GUI initially loads, it creates several instances of each of your vehicle classes (*This is why there are compile errors in the GUI as initially provided to you; it is trying to create instances of classes you have not yet written.*)
- The GUI draws each vehicle on the map by asking for its `getImageFileName` string, and then loading an image file from the current directory. So, if a Human object returns "**`human.gif`**" from its `getImageFileName` method, it will be drawn by showing that image.
- When you click the **Start** or **Step** buttons, the GUI updates the state of the overall map and the state of every vehicle. The **Start** button causes repeated updates, while the **Step** button causes a single update.

  - No vehicles will start on walls and no vehicles should at any time move onto walls.
- On each update, the GUI calls the `chooseDirection` method on each living vehicle to see which way it prefers to move.
  - The `chooseDirection` method parameter informs the vehicle about what terrain is around it. The vehicle uses this information to pick the direction it would like to move.
  - Note that a vehicle may prefer to move in a direction that it is not currently able to move in. In particular, when stopped at a traffic light, the `chooseDirection` method can return the direction that would take the vehicle through the traffic light, while the `canPass` method (described next) reports that the vehicle cannot actually pass through the traffic light.
- After each vehicle reports its preferred direction of movement, the GUI controller checks each vehicle to see whether it can move in the preferred direction by calling the `canPass` method, passing the appropriate type of terrain and streetlight status.
  - If the vehicle can traverse the given terrain with the given streetlight status, the GUI controller updates the vehicle's X and Y coordinates by calling its `setX` and `setY` methods and modifies its direction by calling its `setDirection` method.
  - If the vehicle cannot traverse the given terrain with the given streetlight status, it sits still for the round.
  - NOTE: Every live vehicle should move on every update cycle unless stopped at a streetlight or crosswalk light.
- On each update, the GUI notifies any dead vehicles that an update has occurred by calling their `poke` method.
  - Each type of vehicle has a predefined number of pokes after which it should revive itself; for example, a dead taxi should revive itself after 10 pokes.
  - Live vehicles are never poked.
  - Vehicles should not move on the update in which they revive. Instead, they should revive after the correct number of pokes and face in a random direction. On the next update after they revive they should move.
- At predefined intervals, the GUI changes the map's lights in a cycle (green, yellow, red, green).

Each of your vehicle classes must have a constructor as specified in the descriptions above unless necessary:

- Do not change the number, type or order of the parameters.
- You may change parameter names if you choose.

3.2 Methods to be Implemented:

Each vehicle must also implement the following methods of the instructor-provided `Vehicle` interface.

`public Direction chooseDirection(Map<Direction, Terrain> theNeighbors)`
A query that returns the direction in which this vehicle would *like* to move, given the specified information. Different vehicles have different movement behaviors, as described previously.

`theNeighbors` is a Map containing the types of terrain that neighbor this vehicle. The keys are instances of the `Direction` enumeration, such as `Direction.WEST`, and the values are instances of the `Terrain` enumeration, such as `Terrain.STREET` and `Terrain.GRASS`. The values are guaranteed to be non-null. To access the terrain in a particular direction, retrieve it by specifying the direction you wish to check.

For example, to see whether the square to the west contains a street, one could write code such as the following:

```
if (theNeighbors.get(Direction.WEST) == Terrain.STREET) {// something}
```

OR to see whether the square to the left contains a street, one could write code such as the following:

```
if (theNeighbors.get(getDirection().left()) == Terrain.STREET) {// something}
```

`public boolean canPass(Terrain theTerrain, Light theLight)`
A query that returns whether this vehicle can pass through the given type of terrain, when the street lights are in the given state. Different vehicles respond in different ways, as described previously. For example, a **`Bicycle`** can pass the `Terrain.STREET` terrain type and can also pass the `Terrain.LIGHT` terrain type if the light status represented by **`theLight`** is `Light.GREEN`.

`public void collide(Vehicle theOther)`
A command that notifies this vehicle that it has collided with the given other **`Vehicle`** object. Different vehicles respond in different ways, as described previously.
Collisions should only have an effect when they occur between two vehicles that are alive.
When the GUI notices that two vehicles have collided (whether they are both alive or not), it calls this method on each vehicle; the order in which this happens is not defined. Each vehicle should handle only the update of *its own* status and not the update of the other vehicle's status.

`public String getImageFileName()`
A query that returns the name of the image file that the GUI will use to draw this **`Vehicle`** object on the screen. For example, a living **`Taxi`** object should return the string `"taxi.gif"` and a dead **`Human`** object should return the string `"human_dead.gif"`.

`public int getDeathTime()`
A query that returns the number of updates between this vehicle's death and when it should revive. For example, a taxi should lie dead for 15 updates and then revive (on the 15th update). Calling `getDeathTime()` on a **`Taxi`** object should always return 15.

`public Direction getDirection()`
A query that returns the direction this vehicle is facing, one of `Direction.NORTH`, `Direction.SOUTH`, `Direction.EAST`, or `Direction.WEST`.

`public int getX()`
`public int getY()`
Queries that return the *x* and *y* coordinates of this vehicle.

`public boolean isAlive()`
A query that returns whether this vehicle is alive; that is, if it has not collided with a more powerful vehicle and gotten killed. Killed vehicles revive themselves after a certain number of turns, as described previously.

`public void poke()`
A command called by the graphical user interface once for each time the city animates one turn. This allows dead vehicles to keep track of how long they have been dead and revive themselves appropriately. Live vehicles are *never* poked.

When a dead vehicle revives, it must set its direction to be a random direction.

The static method `Direction.random()` is useful for this purpose .

`public void reset()`
A command that instructs this **`Vehicle`** object to return to the initial state (including position, direction, and being alive) it had when it was constructed.

`public void setDirection(Direction theDir)`
A command that sets the movement direction of this vehicle. This should only be called by the GUI controller, and (possibly) your `poke()` and/or `reset()` methods to set direction at revival or reset.

It should never be called by `canPass()` or `chooseDirection()`

It will be much better if you switch its image properly when a vehicle changes its direction.

`public void setX(int theX)`
`public void setY(int theY)`
Commands that set the *x* and *y* coordinates of this Vehicle. These should only be called by the GUI controller, and (possibly) your `reset()` method. It should never be called by `canPass()` or `chooseDirection()`

Submissions instructions are TBA.

/.

3.3 Enumeration Types:

The following enumeration types are provided to you (there are additional methods in some of them that are used internally by the provided code, but that you will likely not need to use).

```
public enum Direction {
    NORTH, WEST, SOUTH, EAST;

    // returns a random direction
    public static Direction random()

    // returns the direction 90 degrees counter-clockwise from this direction
    public Direction left()

    // returns the direction 90 degrees clockwise from this direction
    public Direction right()

    // returns the direction opposite this direction
    public Direction reverse()
}

public enum Light { GREEN, YELLOW, RED }

public enum Terrain { GRASS, STREET, LIGHT, WALL, TRAIL, CROSSWALK }
```

You should use these enumeration types to implement the behavior of your vehicles.

- The **`Light`** and **`Terrain`** enumerations contain constant values, so they would be used in your code in tests such as "`if (light == Light.GREEN)`".
- The Direction enumeration also contains a few methods, such as the static method `Direction.random` and the instance methods `left`, `right`, and `reverse`.

You should use these methods in your code whenever possible to describe the vehicles' movement behavior.

- For example, to check whether your vehicle's neighbor in its "left" direction (the direction it would be facing if it turned left) is a street, you would write code like the following:

```
if (theNeighbors.get(getDirection().left()) == Terrain.STREET)
{
  // do something
}
```

Note: You can learn more about Enum types in the Core Java and Effective Java books. You can also learn more in the Oracle tutorial on Enum types.

3.4 Inheritance Hierarchy Guidelines:

Since one of the learning goals related to this assignment is to increase your knowledge of inheritance, you must use inheritance to reduce redundancy among your vehicle classes.

Specifically, you must implement a parent class for the concrete vehicle sub-types. The parent class should contain *as much of the vehicle behavior as possible*; that is, all the behavior that can be made common among the sub-classes.

These are the *required* features of your inheritance hierarchy:

- The parent class must not be instantiable and must implement the `Vehicle` interface.
- The parent class must contain all common instance fields shared by the vehicle sub-types.
- The parent class must have a protected constructor that initializes these fields appropriately.
    - Note that this constructor need not have the same signature as the individual vehicle class constructors (it can take more, or fewer, parameters as you choose.
    - Think about what data would be useful to pass from the child class constructors to the parent class constructor.)
- The parent class and all other classes must be properly encapsulated: instance fields *must* be private.
- The parent class *must not* explicitly contain information about all the child classes;
    - For example, the parent class should not store the names of every child class's image files or the number of moves until every type of vehicle revives.
    - It is acceptable for information specific to a particular instance to be passed from the child classes to the parent class constructor upon construction of the object (and stored in a field).
    - It is not acceptable, under *any* circumstances, for any of your classes to perform an `instanceOf` test (or equivalent) to determine proper behavior.
- `toString` must return a reasonable `String` representation for each vehicle, which means that you must override `toString`. This would ideally be done once in the parent class or in each child class individually if there is a good reason for doing it differently for each child class.
- You may NOT add any new *public* or *package-level* methods or any new abstract methods to the `Vehicle` interface or to the `AbstractVehicle` class (except public `toString`).
    - You may add *protected* non-abstract methods to the `AbstractVehicle` class if such methods are used by more than half of the concrete `Vehicle` child classes (so that the new method represents a default behavior for a majority of `Vehicle` types).
    - You may, of course, add private methods to any of the `Vehicle` classes if you wish.
- Your `Vehicle` child classes may contain *only* fields used only by that sub-class, constants used by only that sub-class, constructor(s) and implementations of their respective `canPass` and `chooseDirection` methods. All other data and behavior should be moved upward into the parent class.
- A child class may declare data fields or helper methods *if they are needed solely by that vehicle type*, but such fields or methods *must* be declared private; you may not add public, protected, or package-level methods or fields to the Vehicle subclasses.
- The **javadoc** comments for methods implemented in the `Vehicle` child classes must clearly indicate the specific behavior for each `Vehicle` child class. You *should not* simply inherit javadoc from the parent class because each vehicle child class implements these methods in a unique way.

3.5 Hints:

It is a challenge to move so much behavior upward into your parent class, especially when much of it seems to be different depending on which child class is being used. The following hints may help:

- *To implement collision behavior in the parent class*: Since each vehicle knows its death time, you can compare them and have your vehicle "die" if it collides with another vehicle with a smaller death time.

- *To implement the image file name behavior in the parent class*: Calling the method

  ```
  getClass().getSimpleName().toLowerCase()
  ```

  yields a convenient `String` that is the same as the current object's class name, in lowercase; for example, the above code, when run on an object of the `Bicycle` class, would yield the string "`bicycle`".

- *To implement reset in the parent class*: When the object is constructed, remember its initial position and direction for restoration later if `reset` method is called.

- *To make your parent class generalized so that it can work with all child classes*: Think about what exactly is unique about each vehicle type.
  - Each has a unique set of terrain it can pass, and each has a unique way of choosing which direction it would like to move. You'll represent this by putting the `canPass` and `chooseDirection` methods in the child classes.

  - In addition, you should put a constructor in each child class that initializes the object by calling the parent class's constructor. The parent class needs to implement behavior such as the collisions, so consider passing your vehicle's death time and other needed information upward as arguments to the parent class's constructor.

- Are there any fields which will never change after the `Vehicle` is instantiated? If so, consider making those fields **`final`**. Are there any classes in your project which will never be extended? If so, consider making such classes **`final`**. Are there any methods which will never be overridden? If so, consider making them **`final`**.

4. Unit Testing:

A unit test for class `Human` has been provided (`HumanTest.java`) with tests of the `canPass` and `chooseDirection` methods (and for several inherited methods). When you complete your `Human` class and `AbstractVehicle` class, these tests should pass when run on your code.

- Since these tests were developed as 'black box' tests, they may or may not provide complete code coverage of your `Human` class, depending on how you write your code. It is suggested that you should NOT use the code coverage tool with the provided tests. Just ensure that my tests pass when run on your code.

You are **required** to write unit tests for one of the `Vehicle` classes that you choose as well as the inherited methods implemented in `AbstractVehicle` class.

- The unit tests that you write for your selected `Vehicle` class will be 'white box' tests and therefore should achieve full code coverage of all methods which you choose to implement in the class.

5. Submission:

- Submit your source codes by committing them onto SVN. A README file located in the main folder should explain to anyone other than you the required details about how to load and run your code, organization of your source files, required settings (if any), etc. so that we can load and evaluate your codes on our machines.

- Submit your text answers to the essay questions via Canvas using the given textboxes after the assignment. You may also want to enrich your submission by inserting links, uploading images (such as your design sketches, snapshot of sample codes, etc.) in some cases where it is not easy or convenient to type in the answers. You may also be required your program files onto Canvas.
- Please follow the up-to-date instructions about submission on the Canvas.

Disclamation: You are free to make reasonable assumptions/corrections and continue your work if you find any ambiguities or errors in the above description above, or any type of inconsistency between the description above and the provided program code. Please document your assumptions/corrections in your submission as part of your essay questions.

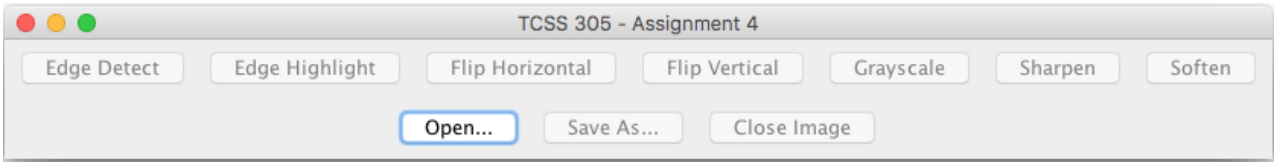
In this assignment, you will review the requirements, design, and partial implementation (provided in the form of skeleton and partially completed codes) for a given problem; and

- complete the application to satisfy the requirements described in Part II – Program Requirements.
- run (or implement using JUnit) end-to-end tests to verify the completeness and correctness of your implementation.

This assignment is designed to extend your understanding of graphical user interface components in Java using AWT and Swing. The graphical user interface (GUI) based application displays and manipulates images. The image processing classes have been already provided for you in the given zip file.

II. Program Requirements

Req1. When the program initially loads, it should have the following appearance.



Req1.1. The window title shall be "TCSS 305 – Programming Assignment **3** (<netid username>)" – replace <…> it with your username.

Req1.2. There shall be buttons centered along the top of the window labeled "`Edge Detect`", "`Edge Highlight`", "`Flip Horizontal`", "`Flip Vertical`", "`Grayscale`", "`Sharpen`", "`Soften`"; "`Open...`", and "`Save As...`" (Each with 3 dots called an ellipses); and another button labeled "`Close Image`".

Req1.3. All buttons except the "`Open...`" button shall be initially disabled.

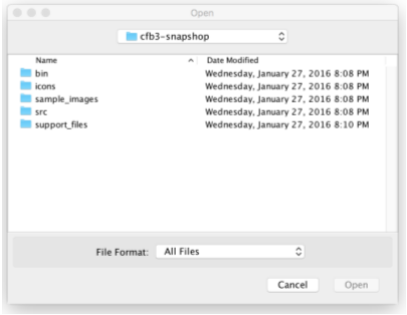
Req1.4. You should add proper icons to the buttons selected from the ones provided in the project folder or from the Internet.

Req2. When the user clicks "`Open...`", a file selection dialog shall appear to let the user select an image file that contains the image to be loaded.

Req2.1. The file selection dialog should open in the current directory as shown below (use relative addressing).

Req2.2. If the user later pushes the " `Open...`" or "`Save As...`" button again, it should open the same folder where the file chooser was left previously.

Req2.3. If the user cancels the file chooser, the contents of the window should be left unchanged.



Req.3. When the user selects a file, the system shall load the image data contained in that file and displays that image in the center of the window.

Note that, the window resizes to exactly fit the image and buttons; if the image is too large, this may make some of the buttons appear outside the screen area. This is expected behavior, and you do not need to handle it in any special way.

Req3.1. When an image is loaded, all buttons shall become enabled so that user can click on any of them.

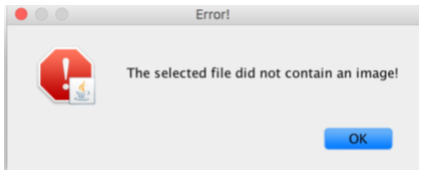
Req3.2. Your window shall be resizable. As the window is resized, any loaded image shall remain centered both horizontally and vertically.

Req4. Clicking each filter button shall cause a modification of the image: flipping, sharpening, softening, *etc*.

Req.4.1. These operations do not modify the original image file, they just change the onscreen appearance.

Req5. Applying more than one filter to an image shall have a cumulative effect; but should not modify the original file.

Req6. If the user chooses a file that does not contain valid image data, the program shall display an error message dialog (as shown on the next page).



Req6.1. The message should also display the name of the selected file.

Req7. If the "`Open...`" button is used while an image is already open, a new image may be selected, and it shall replace the displayed current image. Only one image can be open and displayed for editing at a time.

Req.8. If the user chooses "`Close Image`", the displayed image shall close, all buttons shall disable except the "`Open...`" button, and the GUI shall resize to its initial size and appearance.

Req.9. When the GUI window is closed by clicking the window's close (i.e., x at the top corner) icon, the application shall terminate and exit.

III. Design Guidelines:

1. You will write a class named `SnapShopGUI` in the **`gui`** package that contains the implementation of your graphical user interface. You may also write additional helper classes if you find it necessary.

> Your class may extend `JFrame`, but this is not required (if you do not extend `JFrame`, you will have to store a `JFrame` as an instance field). Do one or the other, but not both.

2. The constructor to your class must require no parameters, and must not display the GUI on the screen; instead, your class must contain a method named `start ()` that performs all necessary work to create and show the GUI on the screen.

> Note that this does not mean that all the work should be performed in the `start ()` method itself; the `start()` method should call other methods to create various parts of the user interface (buttons, panels), set up event handlers, etc.
>
> Note that the main method in the `SnapShopGUI` class calls the GUI constructor and then calls `start ()`.

3. You do not need to write your own code to apply different types of transformation filters on the currently displayed image. Do this by using the instructor-provided filter classes. Note that a single filter object can (and should) be used multiple times; you *must not* create more than one of each type of filter object.

4. You should implement your action listeners using Lambda Expressions and inner classes instead of regular classes (implementing interfaces) and (i.e., `actionPerformed ()`) methods. They are more compact and precise and will increase the readability of your code.

5. The following UML Class Diagram gives a summary of the supplied transformation filter classes in the project folder and their relationships. The `SnapShopGUI` is constructed by the `SnapShopMain` class. The GUI uses various filters (implementing the `Filter` interface) to filter the `PixelImages` that are composed of `Pixels`.

6. Provided Files: The following classes and interfaces are provided for your use. You must NOT modify these files when writing your program – unless you have a strong rationale for doing that.

> Note that the provided classes have other methods beyond what is listed, but the listed methods are all you need.

Interface `Filter`:

```
public interface Filter {
  // Applies this filter to the given image.
  public void filter (PixelImage theImage);
  // Returns a text description of this filter, such as "Edge Highlight".
  public String getDescription ();
}
```

The following classes implement the `Filter` interface. All have a no-argument constructor.

```
public class EdgeDetectFilter implements Filter
public class EdgeHighlightFilter implements Filter
public class FlipHorizontalFilter implements Filter
public class FlipVerticalFilter implements Filter
public class GrayscaleFilter implements Filter
public class SharpenFilter implements Filter
public class SoftenFilter implements Filter
```

Class **`Pixel`**:

```
public class Pixel {
  public Pixel () // constructs a black pixel
  public Pixel (int theRed, int theGreen, int theBlue) // RGB values, 0-255
  public int getRed (), getGreen (), getBlue ()
  public void setRed (int theRed), setGreen (int theGreen), setBlue (int theBlue)
}
```

Class `PixelImage`:

```
public class PixelImage {
  // Loads an image from the given file and returns it.
  // Throws an exception if the file cannot be read.
  public static PixelImage load (File theFile) throws IOException

  // Saves this image to the given file.
  // Throws an exception if the file cannot be written.
  public void save (File theFile) throws IOException

  // Methods to get and set the 2D grid of pixels that comprise this image.
  // The first dimension is the rows (y), the second is the columns (x).
  public Pixel [][] getPixelData()
  public void setPixelData (Pixel [][] theData)
}
```

Class `SnapShopMain`:

```
public class SnapShopMain {
  // Runs your program by constructing a SnapShopGUI object.
  // (You must write the SnapShopGUI class.)
  public static void main (String... theArgs)
}
```

IV. Implementation Guidelines

1. Create your project by downloading the `prga#03-netid-snapshop.zip` file from Canvas, importing it into your workspace. Make sure that when we unzip the file, we should get the folder named `prga#03-netid-snapshot`. Replace netid with your own UW netid. Included with the project are some image files that you can use for testing.

2. Display an image on the GUI by setting it to be the icon of an onscreen `JLabel`. A `JLabel`'s icon is set by calling its `setIcon` method. The `setIcon ()` method accepts an `ImageIcon` object as a parameter. An `ImageIcon` object can be constructed by passing a `PixelImage` as the parameter.

Note that you should not create a new label every time you change the image, but instead should change the icon of the label you already have).

Here is an example that creates a new label and sets its icon:

```
PixelImage image = PixelImage.load(new File("apple.jpg"));
JLabel label = new JLabel ();
label. setIcon (new ImageIcon(image));
```

3. Use a `JFileChooser` for your file selection dialog. A save dialog can be shown by calling `showSaveDialog ()` on a `JFileChooser` object, and an open dialog can be shown by calling `showOpenDialog ()`. You can ask for the file the user selected by calling the `getSelectedFile ()` method on the `JFileChooser`.

Note that your program should create only one `JFileChooser` object and reuse it repeatedly.

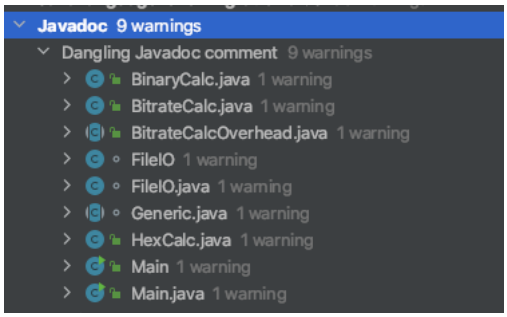
4. If image loading fails, catch the `IOException` and use a `JOptionPane` to display an error message dialog like the one shown above.

5. When an image is selected and loaded, call the `pack` method on your `JFrame` to make it resize itself to fit the image and buttons. If the image is too large, `pack` may make your window larger than the screen; and that is acceptable. (We won't test the program with images that are that large.) After packing the `JFrame`, set a minimum size on the `JFrame` equal to the current size of the `JFrame` so that it cannot be resized too small to display the image or the buttons.

6. Try to design and implement an efficient way of creating filter objects and their corresponding buttons. Ideally, it should take you *exactly* one line of code to create each filter and its corresponding button and the button's `ActionListener` (So seven such lines of code for the seven filter/button/listener combinations.) The goal here is to make it possible to easily add (or remove) a filter and its corresponding button without making changes to other parts of the project code.

7. You should use IntelliJ IDEA Ultimate Edition w/required plugins for your implementation.

8. Your codes should be well documented by using all three types of Java comments effectively.



Hint: *See [Core Java 11/e] Section 4.9 Documentation Comments for examples of the third type of comment (i.e., Javadoc comments.)*

- Use Javadoc comments that describe your program's classes, interfaces, methods, and fields.
- Use regular comments on a need basis to explain hard-to-understand code segments.
- Mark the code segments that satisfies the given requirements clearly in your code with an appropriate comment that includes the requirements number (e.g, `//#Req1.2 is implemented below/here.`)

Hint: *Take advantage of the IntelliJ IDEA's Inspect Code feature to see the status of your JavaDoc comments. You should completely take care of these warnings before you submit your code.*

V. Grading:

1. Part of your program's score will come from
   - External correctness: For this assignment, external correctness is measured by the output generated (correct calculations/processing, correct behavior, correct GUIs, completeness of the implementation of the requirements as described, *etc*.), and can be determined manually or by running automated tests on your code by the instructor.
   - Internal correctness: Internal correctness includes meaningful and systematically assigned identifier names, proper encapsulation, avoidance of redundancy, good choices of data representation, the use of comments on particularly complex code sections, and the inclusion of Javadoc headers on your classes.

   Internal correctness also includes whether your source code follows the stylistic guidelines. This includes criteria such as
   - the presence of Javadoc comments on *every* method and field (even the private ones!),
   - the use of variable names, spacing, indentation, and bracket placement specified in the class coding standard, and
   - (optional*) the absence of certain common coding errors that can be detected by the tools.
     **Although it is optional, it is to your advantage to use the IDE plug-in tools like CheckStyle, FindBugs, and PMD to check your code before you submit it.*

On this assignment, internal correctness also includes avoidance of redundancy, because it is easy for GUI code to become redundant especially when there are many similar elements (such as the filter buttons). In particular, redundancy in file choosers (using more than one file chooser object) and filters and their buttons (using more than one of each filter, writing the same code multiple times with only minor textual differences to create each filter button) will hurt your internal correctness grade. Try to find the most concise and elegant way possible to create this GUI without repeating nearly identical code many times.

VI. PRGA Report: Expected Deliveries:

1. A (one-page) summary table with proper headers that displays the following metrics for your object-oriented program:

   [1] number of lines (excluding the space and comment lines,)
   [2] number of lines in the shortest class,
   [3] number of lines in the longest class,

   [4] maximum number of lines in a method,
   [5] minimum number of lines in method,
   [6] maximum number of methods in a class,
   [7] minimum number of methods in a class,

   [8] number of imported Java libraries,
   [9] number of packages,

   [10]number of classes,
   [11]number of abstract classes,
   [12]number of interfaces,
   [13]number of concrete classes,

   [14]number of methods (including constructors),
   [15]number of static methods,
   [16]number of methods responsible for only input/output,
   [17]number of methods responsible for some type of processing as well as input/output,
   [18]number of methods responsible for some type of processing but no input/output,

   [19]number of static variables,
   [20]number of final static variables.

2. A [clickable] full link address and one-page snapshot of your mind map that presents the (grouped list of) a) procedural and object-oriented programming constructs; and b) primitive and object reference data types that were used in the complete program (provided codes plus newly added codes.)

3. On a single page and using at least 2-3 complete sentences for each, describe how the following Object-Oriented fundamental concepts were used or implemented in the complete program (provided codes plus newly added codes.)
   - Abstraction
   - Encapsulation
   - Inheritance
   - Polymorphism

> ***Hint:*** *Due to nature of the assignment and provided skeleton/partial codes, the final codes will have applied/exploited each of them* ***directly*** *at the end. Therefore, you should, here, describe how they have been applied/exploited in the program, whether you did it in your newly added codes, or they were part of the given design or skeleton/partial codes.*

4. A mini-snapshot of the `IntelliJ IDEA's Project Panel` that shows your Java projects' organization in terms of packages, classes, libraries, etc.

> ***Hint:*** *Place your mini-snapshots side by side on the page to save space. Arrange the size of snapshots so that they are not too large, just sufficiently large to make the content readable. In most cases, two snapshots should fit side-by-side on the same row.*

5. A snapshot of the UML Class Diagram that displays all classes, fields, methods, dependencies, etc. of your technical solution (i.e., design.) Adjust the diagram to fit onto one page and take a (readable) snapshot to paste into your document.

> ***Hint:*** *You can view the structure (i.e., structural model) of your program as a UML Class Diagram using the* `IntelliJ IDEA (Ultimate Edition) > Diagrams > UML Class Diagram` *feature or its equivalent on the Windows platform, such as PlantUML. Then display all of the above before taking a snapshot of your diagram.*

6. On one or two pages, a set of mini snapshots that present your GUI design thoroughly with some selected good examples from your menus, user prompts, and program outputs or error/warning messages that will prove that you have satisfied the requirements listed in II. Program Requirements. You should definitely include a snapshot for each image included above in the II. Program Requirements.

7. In 3-5 full sentences, give a brief explanation about how you implemented the input validation and handled OR prevented the potential input errors in the menus displayed to the user or during the input values' entry.

8. In 3-5 full sentences, give a brief explanation about how you implemented the exception handling and handled the exceptions raised during the calculations, such as division by zero or out-of-range results.

9. A list of your **five (5)** most important takeaways from this assignment. Each should be presented in 2-3 complete sentences. They can be technical, non-technical, or preferably a combination of both.

10. A [clickable] link of the full address to your *10 ±2-minute* Panopto video presentation where you go over and shortly present) the sections of your report and b) representative end-to-end test cases from entering inputs to displaying outputs while describing how your program run those cases; starting from main() method,

Hint1 1: *You should use* `Canvas > Panopto Recordings` *to record your video presentation. Configure it so that the screen image is recorded in good quality and readable afterwards.*

Hint 2: *You can make one recording, or you can divide it into multiple recordings. You can decide how you will organize your presentation or video recordings as you wish. Please write one sentence of description for each video if you have multiple video recordings.*

Hint 3: *You can use IntelliJ IDEA Debugger and set some breakpoints in your codes to describe step by step how your program runs and how the internals operate while running the test cases.*

(Tentative) Grading Rubric:

PRGA Report Delivery #	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	Total Points
Points	5	10	20	5	20	10	10	10	10	20	120

Program Codes	Completeness	Correctness	Documentation	Design	Total Points
Complete Implementation	45	45	10	50 Following the given Design and Implemention Guidelines	250

/.