Programming Project 1

---

**Guidelines**

A programming project is a laboratory that you complete on your own, without the help of others. It is a form of take-home exam. You may consult your text, your notes, your lab work, our online examples, and the course web pages, but no other sources for code are permitted. You are encouraged to reuse the code from your labs or our class examples, however. While you may discuss these problems with the course instructors, you may not discuss them with the TAs, tutors, classmates, friends, etc. The use of any outside help or sources is a violation of the Honor Code.

This project is due by 9am on Monday, October 16. No late work will be accepted.

The projects will be graded out of 100 points, with roughly half the points given for correctness and half the points given for style. Four points are allocated to a short writing assignment – a response to a prompt that we provide below. Thus, even imperfect programs can receive close to full credit, but poor design and style can lead to quite low final scores. A more detailed point breakdown appears at the end of this document.

---

**Program 1: Dashboard**

Welcome to T-eph-la© Self Driving Cars! Your first assignment as an Auto Auto Test Engineer is a car dashboard. Self-driving cars take into account a variety of variables before determining optimal paths of action. Your self-driving car dashboard, entitled Dashboard, will calculate the stopping distance of the car given randomized odometer mileage, speed, and car type. Our patented self-driving car stopping distance formula is based on the total stopping distance formula, but with some top secret modifications described below: https://en.wikipedia.org/wiki/Braking_distance

Total stopping distance is the sum of the perception-reaction distance plus the braking distance or:

\[
\text{stopping distance} = \text{perception reaction distance} + \text{braking distance}
\]

where

\[
\text{perception reaction distance} = \text{speed} \times \text{perceptionReactionTime}
\]

and

\[
\text{braking distance} = \frac{(\text{speed} \times \text{speed})}{(2 \times \text{frictionCoefficient} \times \text{gravity})}
\]

In your program, the randomly determined speed the car is going will be used directly in the formula. Gravity is represented as 9.81.

The randomly generated odometer mileage will determine the frictionCoefficient: cars with less than 3,333 miles have good brakes and a frictionCoefficient of 0.9, cars with mileage between 3,333 and 6,666 have a frictionCoefficient of 0.7, and cars with mileage greater than 6,666 have the tires with least grip and a frictionCoefficient of 0.5. Here at T-eph-la©, our self-driving cars are retired after 9,999 miles and no longer require testing or simulation.

perceptionReactionTime is determined by one of three randomly generated car types that we are currently testing: a van, with a perceptionReactionTime of 1.75 seconds, an SUV with a perceptionReactionTime of 1.5 seconds, and a microcar with a perceptionReactionTime of 1.25 seconds. Our researchers observed that 1.5 seconds is an average perception reaction time, but that our test cars have some impact on driver reaction times.

When your program, entitled Dashboard, initializes, there should be visual placeholders for the stopping distance, car type, speed, and odometer mileage to be displayed. (See the window on the left side of the figure below.) There should also be a start button that when pressed, generates the
appropriate values (random and arithmetically determined) to be displayed on the dashboard. (See
the window on the right.)

See the handouts page on the course website for a demonstration of the self-driving car dashboard:
http://www.cs.williams.edu/cs134/handouts.html

---

**Program 2: Radar Cars**

Here at T-eph-la©Self-Driving Cars, we believe in rotating our engineers throughout the company.
In this program, you'll be focusing on the radar beams the self-driving car uses to see the environment
around itself. Here's what we mean:

The canvas initially starts empty. When you press on the canvas, a self-driving car appears. Ini-
tially it will just be a car with two tires, a body, two windows, and a radar. Other details, like headlights
and the top frame are not required for the testing of the radar beams. If you press on the radar and
drag, two radar beams grow horizontally, one in each direction: the left radar grows left-ward, the
right radar beam grows to the right. Right now, our radar beams grow to a maximum of two-times the
car body's width. With a left and right radar beam, that’s a total width of four-times the car body’s width.

If you press the car body after the beams have grown to their maximum width, the car changes color.

If you press on the windows at any time, a single passenger appears. These cars are still in beta-testing, so only one passenger can ever be legally seated inside, but that passenger should change seats to appear in the selected window.

When you are happy with the car’s color and passenger location, you can manufacture another one by pressing somewhere else in the window. When the mouse moves out of and then back into the window, the scene resets itself to be empty.

**Program Design.** Your program should be divided into two classes: a window controller called `CarController` and a `RadarCar` class. We have actually provided a complete `CarController` class in the starter folder. You should not modify our `CarController` class in any way. Instead, you have to implement the `RadarCar` class so that it works with our `CarController`. In particular, the `RadarCar` constructor should expect two parameters: the Location of the center of the car’s body, and the canvas. The `RadarCar` class should define the following methods used by the controller to implement the functionality described above:

- **public void changeColor():** sets the color of the car body to a random color, but only if the radar beams have reached their maximum width.

- **public boolean windowContains(Location point):** returns true if either of the car’s windows contains the given point.

- **public boolean radarContains(Location point):** returns true if the car’s radar contains the given point.

- **public boolean carContains(Location point):** returns true if the car body (not the windows!) contains the point.

- **public void passenger(Location point):** moves a passenger to the given point.

- **public void growRadar():** makes the car’s left and right radar beams grow a bit, until the radar beam reaches its maximum width.

The radar beams will only grow when the mouse moves, so `RadarCar` should not extend `ActiveObject`.

**Take the wheel.** Your self-driving radar cars need not look identical to ours — you may make the size, shape, windows, and other features look however you like, and also feel free to add other graphic items to the `RadarCar` class to make the cars more attractive.

---

**Program 3: Crosswalks**

Here at T-eph-la©Self-Driving Cars we pride ourselves on safety. Our cars follow the rules of the road, and that means stopping at crosswalks to give pedestrians the right of way. The goal of your final program is to write a crosswalk-observing self-driving car simulation. The canvas will be a one-way road, with cars travelling from left to right. As shown in the image on the left below, the program begins with an empty road. The rectangle displayed on the canvas is a crosswalk. The user controls the simulation by creating cars and by signalling that a pedestrian is ready to cross or has finished crossing.

To create cars, the user clicks on the canvas. Because we want to safely test our cars’ ability to stop, we only allow the user to create a car to the left of the crosswalk – not on it or beyond it. Each time the user clicks in the region to the left of the crosswalk, a car is created so that its top left corner is at the point of the click, provided the entire vehicle can be displayed without intersecting the crosswalk. It is fine if the cars overlap visually, as in the figure on the right.
Cars move along the road, driving until they disappear off the right side of the canvas. As they drive, they pay attention to the crosswalk. After all, in Massachusetts, stopping for pedestrians is the law! (Besides, you’d never want to delay a student eagerly walking to their CS 134 class.) Of course, pedestrians need to be responsible as well. To simulate a pedestrian wanting to cross, the user can click on the crosswalk. If the user does this, the crosswalk changes color, and all cars that are required to stop do so. If a car is already beyond the crosswalk, there is no need to stop. Additionally, if a car is already in the crosswalk, it should continue. (Stopping abruptly and then blocking the crosswalk could be problematic both for the passenger and for the person trying to cross.) Any new cars created while the crosswalk is in a “stop” state must also be stopped until the crosswalk is released.

To simulate the idea of the pedestrian getting across safely, the user can once again click on the crosswalk. This changes the crosswalk back to its original color, and any stopped cars can move again. The figure on the left below shows a crosswalk signaling a stop in red. Note that the cars on or beyond the crosswalk keep moving. The figure on the right shows the same scene right after the user sets the crosswalk back to the “go” state, displayed by a green crosswalk.
**Program Design.** Your program should be divided into three classes: a window controller called `CrosswalkController`, a class that extends `ActiveObject` called `MovingCar`, and a class called `Crosswalk`. We have provided the outlines of these classes in the starter folder. The starter folder also contains various vehicle images for you to use.

**The Window Controller.** The `CrosswalkController` window controller creates the initial scene and handles the mouse interaction. We placed our crosswalk about three-quarters of the way from the left of the canvas, and we made it green in the “go” state, but the exact placement and color choice are up to you.

When the user clicks to the left of the crosswalk such that an entire car can be safely placed, a `MovingCar` should be created. If the user clicks on the crosswalk, the `Crosswalk` object needs to adjust its state.

**Moving Cars.** A `MovingCar` object needs to know how to construct itself. It also needs to move itself across the canvas, knowing that when it gets to the end, it is done. Along the way, it needs to pay attention to the crosswalk signal.

You might begin by simply constructing a car that moves, without regard to the crosswalk. Once you have that working, you can add functionality.

**The Crosswalk.** Don’t forget that the crosswalk is not simply a colorful rectangle! It can be placed into a state that indicates cars should stop; and it can subsequently be placed into a state that indicates cars can go. As with the moving cars, you can start simple here. Begin by making sure you can construct a crosswalk to appear visually on the canvas. Then add the ability to turn it a different color when it is clicked; now make sure it goes back to the original color when it is clicked again. Consider how the crosswalk knows what state it’s in.

**A reminder about images.** Note that images do not load properly if the name of your project folder or any of its enclosing folders contain spaces. That is, folder names like “Project1 ID nnnn” should be avoided. We can help sort things out if you have trouble loading images, but you can continue to make progress even if you can’t find us right away. Simply use FilledRects for cars in the meantime.

---

**Reflection**

You may have heard of the Hippocratic Oath, which is an oath historically taken by physicians. In essence, it is a pledge to “do no harm.” Software engineers are not required to take an oath, but the computing industry’s main professional society has a code of ethics that all members of the profession are expected to follow. Though different from the Hippocratic Oath, there are interesting similarities, because developing and deploying new technology with wide reach can have both positive and negative impact. These effects can be social, political, economic, etc.

Just this morning one of us overheard a conversation about self-driving vehicles. One person said that their dream was to have enough time to pack everything up in a van and travel all across the country, stopping at interesting places, exploring the natural beauty of the land. Though it was a long way off, they could imagine themselves doing that in retirement. Another person pointed out that waiting until retirement could be a problem. Driving a van all around the country would require retaining the ability to drive safely, something that sadly becomes more difficult with age. “Ah! Not to worry!,” the first person said. Being a computer scientist, they were confident they’d have a self-driving van by retirement. Reflecting on this, the second person said, “Yes! Great for the elderly, and great for families with kids.”

Consider the implications of the final statement above. What are some social, political, or economic effects of self-driving cars? What positive or negative impacts might a computer scientist working for a self-driving car company be responsible for? Write a brief (2 paragraph) but thoughtful reaction. Please type it in the file called `Response.txt`, which appears in the programming project starter folder.
Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment. Please do not submit three separate folders. Instead, place the folders for all three of your complete programs into one folder, make sure that your ID appears in the title of the main folder and each of the subfolders, and then place the main folder in our dropoff folder with the following steps:

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.
- From the “Go” menu at the top of the screen, select “Connect to Server...”.
- For the server address, type in “afp://Guest@fuji” and click “Connect”.
- A dialog box will pop up. Connect as a guest by selecting the “Guest” option.
- A selection box should appear. Select “Courses” and click “OK”.
- You should now see a Finder window with a “cs134” folder. Open this folder by double clicking on it.
- You should now see the drop-off folders for the three lab sections. Drag your “Project1IDnnnn” folder into the appropriate lab section drop-off folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK”.
- Log off of the computer before you leave.
Grading Guidelines

Points will be assigned roughly as follows:

Style (16 pts per program)

- Proper use of boolean conditions
- Proper use of ifs/whiles
- Proper use of variables (instance/local, public/private)
- Descriptive comments
- Good names
- Good use of constants
- Appropriate formatting (indenting, white space, etc.)
- Parameters used appropriately

Correctness (16 pts per program)

- Dashboard
  
  Initial layout and default values
  Start button generates and displays new random values
  Car type and mileage determine formula values
  Stopping Distance formula executes correctly
  Stopping Distance displayed

- Radar Car

  Car created correctly on press
  Radar Beam grows on radar-click then drag
  Passenger appears or moves after window clicking
  Car color changes correctly
  Containment in car body, windows, and radar are determined correctly

- Crosswalks

  Crosswalk drawn correctly at start
  Cars created in response to click in correct part of canvas
  Cars move across the canvas
  Cars know when they reach the end of the canvas
  Crosswalk toggles between “stop” and “go” states when clicked
  Cars respond appropriately to the crosswalk

Written Response (4 pts)

Extra Credit (up to 6 pts)  While not required, feel free to experiment and add embellishments to your programs once you have the basic features working. You may receive a small amount of extra credit for any extensions.