Project 2 - Minivan Door

Due Sep 30 by 12pm **Points** 80

Overview

In this assignment, you will write another complete class that is intended to reinforce your skills in writing instance variables, accessors, and mutator methods, while also increasing the use of if statements to provide conditional logic. You will also write tests to verify the proper operation of your code.

Your task is to develop software that monitors and controls the operation of the power sliding door of a minivan.

Requirements

The state of the door system is governed entirely by the following operational rules:

The door can be locked or unlocked at any time, by using a lock/unlock button on the dashboard. When the door is closed, the door can only be successfully opened if: 1) the gear shift lever is in park; and 2) the door is unlocked. In such a state, any of the following mechanisms will successfully open the door: 1) an open button on the dashboard; 2) a handle on the outside of the door; or 3) a handle on the inside of the door. However, the inside handle mechanism will not open the door if a separate child-safety mechanism has been engaged, regardless of any other conditions. The child-safety mechanism can only be engaged or disengaged while the door is open. The door can be closed at any time. The gear can be changed at any time.

You will write a class that represents these features and manages the sliding door. In addition to maintaining the current state of the sliding door system, your class will also generate an event log containing a history of all attempts to change the state of the system (whether successful or not). To test your system, rather than writing an informal tester class as in the previous assignment, you will develop a comprehensive test suite using the JUnit framework, which will need to test the entirety of your implementation.

System Description

You will develop a class called `MinivanDoor` in the `edu.vt.cs5044` package that models the behavior and state of the sliding door of the minivan. The methods of this class will be invoked by other systems, displays, and sensors within the minivan to indicate certain requests or activities. The methods of your implementation must behave according to the operational rules above. Your class will provide the following public methods, plus any private instance variables and private methods as needed:

- Constructor:
  - `MinivanDoor()` initializes the minivan door system to its default state:
    - door is closed
    - door is unlocked

    - child-safety is disengaged
    - gear shift lever is in park
    - event log is empty
- Accessors:
    - `isOpen()` request the state of the door
    - `isLocked()` request the state of the door's lock
    - `isChildSafe()` request the state of the child-safety feature
    - `getGear()` request the state of the gear shift lever
    - `getLastLogEntry()` request the most recent entry from the event log (returns `null` if the event log is empty)
    - `getLogEntryCount()` request the number of entries in the event log
    - `getLogEntryAt(int index)` request the specified log entry event (returns `null` if the index is invalid)
- Mutators*:
    - `setGear(Gear gear)` indicate an activation of the gear shift lever
    - `setChildSafe(boolean engage)` indicate an activation of the child-safety (on or off)
    - `pushLockButton()` indicate an activation of the dashboard lock button
    - `pushUnlockButton()` indicate an activation of the dashboard unlock button
    - `pushOpenButton()` indicate an activation of the dashboard open button
    - `pullInsideHandle()` indicate an activation of the inside handle
    - `pullOutsideHandle()` indicate an activation of the outside handle
    - `closeDoor()` indicate an activation of the door closure sensor

(*Each mutator call must always result in exactly **one** `LogEntry` value appended to the event log, even if no state is actually mutated due to the operational rules. The event log must be stored as an `ArrayList<LogEntry>` field.)

You will also develop a comprehensive JUnit test class called `MinivanDoorTest` to confirm proper operation of your `MinivanDoor` implementation.

Downloads

- **MinivanDoor.java** source file with method placeholders
- **MinivanDoorTest.java** JUnit test file with sample test methods
- **Gear.java** enumeration type for gear shift values
- **LogEntry.java** enumeration type for log entry values

Getting Started With Eclipse

Create a new Java Project for this assignment, and within the src folder of that project create a new Package named `edu.vt.cs5044` for your code. Within this package you need to copy all of the provided files, **except** the `MinivanDoorTest` file.

The `MinivanDoor` class already has an empty constructor, one private instance variable, and placeholders for the public methods listed above. There is only enough code to satisfy the compiler at this point, so the

methods certainly don't yet work as intended. However, we have at least something that compiles, in order to demonstrate how JUnit works (see below).

You'll also need to add Eclipse support for JUnit for this project, so right-click your project, select Build Path | Add Libraries, then highlight JUnit and click Next. Select the version as "JUnit 4" (Note: this is NOT the default version) and click Finish to add the built-in JUnit library.

Next, it will be very useful later in this project to have a separate source folder structure for you test file. To create this, right-click your project and select New | Source Folder. (If "Source Folder" is not an option, select "Other" instead and type "source" to search for it.) Name the new source folder "test" and click Finish.

In the project at this point, you should see both the "test" and the "src" folders, along with nodes for the JRE System Library and JUnit 4.

Finally, right-click "test" folder, add a new Package named `edu.vt.cs5044` there, and copy the provided MinivanDoorTest file into this package.

Working With JUnit

Background

For the first time, you'll develop a formal test suite via JUnit, so you can much more readily self-check your code as you write it. You've probably already seen how self-checking with a tester class can allow you to find your own errors as early as possible during the development process. Now we'll use JUnit to help automate this process completely, making it even easier to develop and run test code.

JUnit is optimized for software test cases to be developed as individual methods. Each test method has the `@Test` annotation, a nice (often long) descriptive name beginning with "test" by convention, and a simple structure with three main steps: 1) Setup the initial conditions; 2) perform the one action to be tested, which is typically a single mutator call; and 3) confirm the resultant behavior via assertions.

Provided Test File

Open the `MinivanDoorTest` file provided above, and refer to it for the next few sections.

It may look a bit odd at first, but this file contains three (working and complete) sample test methods. One validates the constructor, one tests locking the door, and one tests attempting to open the locked door via the button. Notice that each test method requires the `@Test` annotation immediately above its signature. That tells JUnit to treat this as a test method. There's also another method with the the `@Before` annotation. JUnit will call that method each time (and before) any of the `@Test` methods is executed. So here, every test method will automatically have access to a default (newly-constructed) `MinivanDoor` object. Note also that JUnit test classes should not declare constructors, nor should they contain a main method.

Initial Conditions

In some cases, such as two of the provided examples, the `@Before` method is entirely sufficient, so no additional setup code is needed within the `@Test` methods. However, many `@Test` methods (as in the third

provided sample) will need to manipulate that default object to prepare other initial conditions required for the test. Often this setup is achieved by calling other test methods, but any setup code is acceptable.

Action To Be Tested

Performing the **one** action you want to test is typically pretty easy, since it's usually just a single mutator method call. Just make sure it's very clear in every test method exactly what action is being tested. Also, it's important to resist the temptation to test multiple actions within a single test method. You should always create a new test any time you want to test an additional action. The first test method can act as the setup for the second action, if applicable. You'll naturally end up using lots of copy-and-paste when developing test cases, and that's perfectly acceptable.

Assertions (Confirming Results)

To confirm the action actually did what it was supposed to do, JUnit provides a family of `assert__()` methods that let you state the expected results in code. Keep in mind that an assertion is just a claim about something that you expect. If an assertion is valid during execution, the code simply continues normally. Reaching the end of the test method without any failed assertions means your code passed that test. However, as soon as any assertion fails, that test method immediately stops executing, reporting exactly which assertion failed.

There are actually dozens of `assert__()` methods available in the JUnit library, but the ones you will probably find most useful for this assignment are `assertEquals()`, `assertTrue()`, `assertFalse()`, and `assertNull()`. There are examples of each in the provided file. In assertions such as `assertEquals()`, where two values are required, the *expected* value goes first, while the *actual* value goes second.

Let's Do This!

Finally, to actually run your tests, right-click on your `MinivanDoorTest` class in the package explorer view and choose "Run As | JUnit Test" to execute all the tests. Eclipse will show the results in a JUnit pane, as a tree of tests (with tiny ✓ or x badges) along with a green or red bar. A red bar means that at least one test method had an assertion that failed. At this point we expect all our tests to fail, since we haven't implemented anything yet, so a red bar it is. But now we have a goal, to make that bar green!

Double-click one of the test methods that failed, and the editor will jump to and highlight the exact assertion that failed. The "Failure Trace" at the bottom of the JUnit pane will also show more details about the expected values. For example, double-click the constructor defaults test in the tree. You'll see the `getGear()` assertion line highlighted in your test file, and this line in the failure trace below:

```
java.lang.AssertionError: expected:<PARK> but was:<null>
```

This is as expected, since we only have placeholder code for `getGear()` at this point. Note that the placeholder code did actually happen to pass a few of the assertions, but that was purely coincidental!

Planning Your Development

Read through the description for each method, thinking about how you might implement and test the behavior, then plan an order in which you want to implement the methods. It's completely natural to adjust your plan as you go along, but at least just choose one to get started for now. The constructor and most of the basic accessors should be relatively easy to test and implement first.

Once you've selected a method to implement, don't implement it quite yet! First, try to write at least one test method *before* you start coding the implementation. The new tests will almost certainly fail when you first run them, but your code will begin to pass as you start to develop the method. This strategy goes by the names of "Test-First Coding" and "Test-Driven Development" (which are slightly different from each other, but not in a way that matters to us). Sometimes you'll find you need to start coding a new method to get your current method to pass every test, but try to continue this process by writing at least one test case for that new method before starting its implementation.

Additional Hints and Tips

ArrayList

As noted above, the event log is to be held as an `ArrayList<LogEntry>` which is a collection we'll cover in detail very soon. For now, just create a private field like this:

```
private final List<LogEntry> eventLog;
```

Then, in the constructor, you must initialize this field to a new empty `ArrayList` that will hold the LogEntry objects:

```
eventLog = new ArrayList<LogEntry>();
```

To populate the list, we use the add() method, for example:

```
eventLog.add(LogEntry.DOOR_LOCKED);
```

There are many ways to query the list, but we'll only require the following two:

```
int logSize = eventLog.size();
LogEntry value = eventLog.get(i);
```

In the get() method above, note that `i` is an index location within the list, where the first entry is at index location *zero*.

Enumeration Types

You have been provided source code for two pre-defined enumeration types (specifically `LogEntry` and `Gear`) to be used within several of the required methods. Please use this code exactly as provided, without modification. See Horstmann 5.4 for a little more background on the usage of enumeration types. The basic idea is that we've defined a set of distinct constant values, in a way that leverages Java's natural type-safety

system. This ensures that our code can only ever use valid values, as enforced by the compiler, thus eliminating typos and many other common mistakes.

Code Path Test Coverage

Your test file will need to achieve 100% coverage of you `MinivanDoor` implementation for full credit in Web-CAT. As you test your code, you may find that Web-CAT indicates you have certain conditions that are not being completely tested by your own tests. You'll need to include tests for each unique way an overall condition can be true or false. This may require you to test the same action you've already tested, but starting from a different set of initial conditions, to exercise every possible branch.

Note that you do NOT need to achieve 100% coverage on the enumerated type files (Gear and LogEntry). Even if you exercise every value, you'll still notice a single missed line of coverage, due to the way enum classes are handled internally. However, if you'd like to cover those last lines, you can add the following test method to your test file as a workaround:

```
@Test
public void testDummyCaseForEnumCoverage()
{
    assertNotNull(Gear.valueOf(Gear.PARK.name()));
    assertNotNull(LogEntry.valueOf(LogEntry.NO_ACTION_TAKEN.name()));
}
```

This test method serves only to complete the coverage the enum files.

Test Coverage of Short-Circuits

Due to short-circuit of logical operations, if you have N boolean expressions joined by AND or OR, you will only need N+1 test cases for full coverage. For example, testing a conditional with 2 components such as `(isThis && !isThat)` requires 3 distinct test cases:

- Overall false:
  - when `isThis` is false, regardless of the `!isThat` value
  - when `isThis` is true, and `!isThat` is false
- Overall true:
  - when `isThis` is true, and `!isThat` is true.

Web-CAT will highlight any such conditions you haven't fully tested, but it can't tell you exactly which of these possibilities you have left untested.

Impossible Test Coverage

Redundancies in your conditionals can make it impossible to cover all branches with test cases, even if the code works exactly as intended. A very simple example is the following:

```
if (isReady) {
    ...
} else if (!isDone && !isReady) {
```

```
    ...
}
```

All branches of the second conditional above cannot be tested, because `!isReady` at that point can never be false. To resolve this, you must remove any redundant components from the conditionals.

Submission to Web-CAT

Submit your solution to **Web-CAT (http://web-cat.cs.vt.edu/)**, ideally via the Eclipse plug-in. Please feel free to improve and resubmit your code as many times as you like (before the deadline) in order to improve your score.

Important Notes:

- Your submission will be evaluated both by Web-CAT (40 points) and by human (40 points). As such, Web-CAT's automated score will show at most 40/80 points.
- Your JUnit test class will be evaluated by Web-CAT to ensure your code passes your own tests, and that your tests provide complete coverage of you implementation code. However, you test code will NOT be evaluated for style. Web-CAT will also use its own internal test suite (that we've developed) to evaluate the correctness of your code with respect to the requirements.
- Carefully review your feedback. In the summary, you will see a table listing each of your source files with a green/red bar by each non-test file. If the bar for any file is not *completely* green, then some code in that file was not exercised by your own tests. Click the file name to view your source code and look for lines that are highlighted in pink. Hover your mouse over highlighted lines for additional information.
- See the **project grading rubric** for full details on the grading criteria applied to this assignment.