[DIT085] Testing and Verification
University of Gothenburg
Group 1.1

2016-02-25

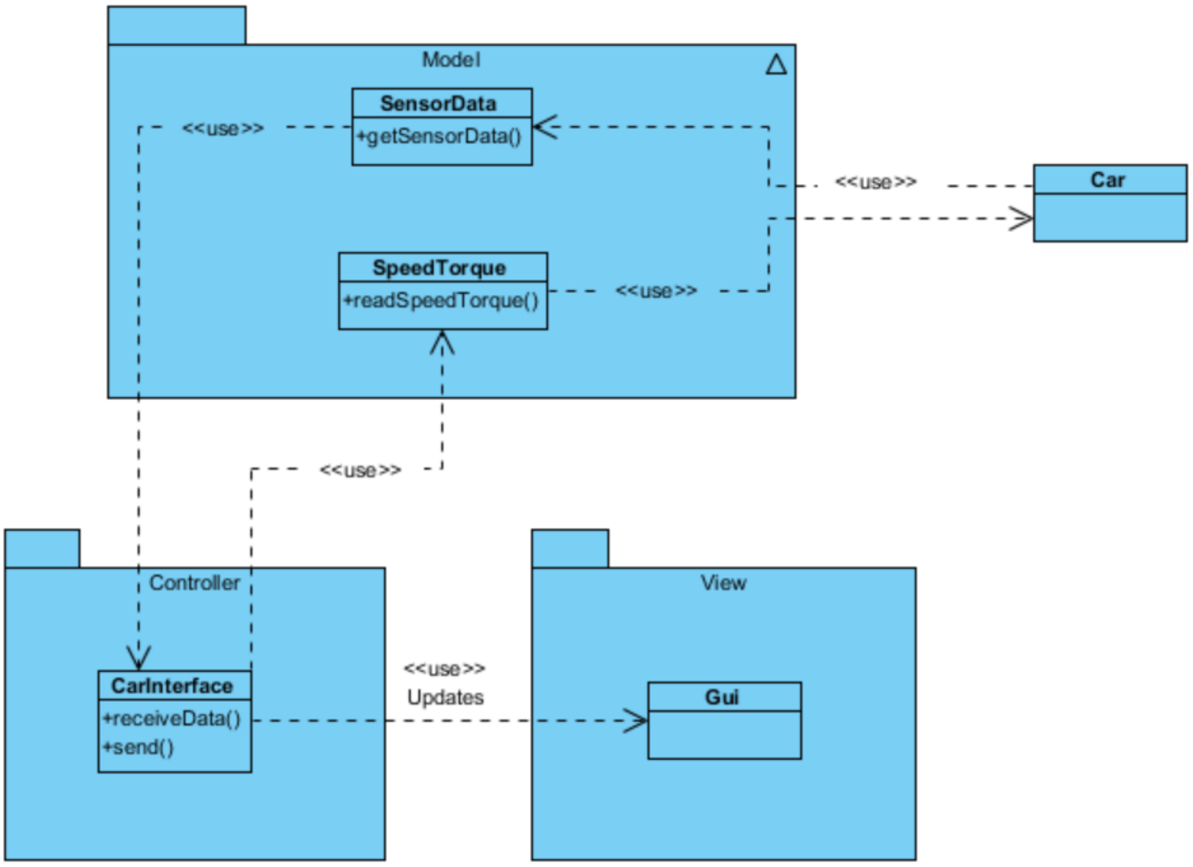
# PHASE 2

Software integration,
Integration testing, and
Measuring code coverage and
Testability

Team members:
Kristiyan Dimitrov
Martina Freiholtz
Kai Salmon

## Part 1: Architectural Design

In the instructions for Phase 1, methods for receiving speed and torque data was to be developed. This method will be used for receiving speed and angle data in Phase 2.



In the second phase of this project we elected to use a Model View Controller architecture. The **View** component is represented by the **GUI** class (a simple Swing UI), the **Controller** component is represented by the **CarInterface** class, and the **SpeedTorque** and **SensorData** classes makes up our **Model** component.

**Car** sends speed and angle date in the form of a bitstream. The stream is read by the **SpeedTorque** class (developed during Phase 1) which looks for the first valid data packet in the stream, and returns an object with speed and angle data. The object is received by the **CarInterface** (new object every second) and displayed in the **GUI**.

The user enters sensor values into the interface, which are sent to the Odroid (**Car**) every 2 seconds. **SensorData** is used to convert the values to a bitstream.

# Part 2: TDD of New Modules

## Gui

A simple Swing UI which displays inserted and received values.

The threads are started in this class, calling methods from the **SpeedTorqueObj** class (from Phase 1) and the **CarInterface** class (described in detail with test cases below).

## Car

Contains a constructor and getter and setter methods for its data stream.

## CarInterface

### receiveData

**Summary:**
receiveData is called to receive speed and angle data from the car.

**Pre-condition:**
A valid data packet is received in the form of a bitstream of valid length and with valid delimiters.

**Post-condition:**
An object containing the speed and angle values of the car is returned. If the data packet is not valid, an object with negative values is returned.

**Test cases:**

TC0: The data packet received from the car is valid.
TC1: The data packet is incomplete or of invalid length.
TC2: The data packet is corrupt.

## send

**Summary:**
Originally contained the functionality of **sendValid**, described below. The method converts sensor values to a stream using the method **getSensorData** in class **SensorData**, and calls method **sendValid** with the stream as an argument. The getSensorData method has already been tested in Phase 1, and should always return a valid stream (such as in test case 0).

**Pre-condition:**
The method takes three double values as arguments. Since the values are being manually entered by the user, they are assumed to be valid.

**Post-condition:**
The **sendValid** method is called, and nothing is returned from **send**.

**Test cases:**

TC0: The sensor values result in a data packet of the correct length and with correct delimiters.

## sendValid

**Summary:**
In order for us to test the functionality of sending a corrupt or empty stream (in the event that **getSensorData** returned an invalid stream), our solution is to separate the untested functionality and place it into the method **sendValid**, which takes a stream as an argument, tests if it's valid, and sends either the valid stream or an empty stream to the car.

**Pre-condition:**
A bitstream (data packet) is entered as an argument.

**Post-condition:**
Either an empty stream or the original stream is returned.

**Test cases:**

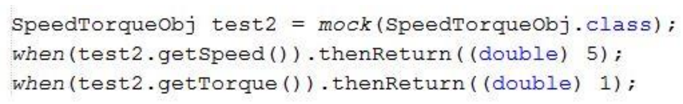
TC0: The data packet is corrupt.
TC1: The data packet is empty.
TC2: The data packet is valid.

# Part 3: Integration Testing

## 3.1 Integration Testing Using Mockito

Mockito is a mocking tool which allows us to create dummy classes and emulate their return value or exception throwing without receiving any compiler errors. Its main usage, for us, was in the CarInterfaceTest class. We manipulate the behavior of the SpeedTorqueObj class to return imaginary values with the “when” method built into Mockito. By creating an active stub and mocking the methods getSpeed and getTorque, we return a set of dummy values which, if the implementation was done correctly, should match the real values returned by the Car through the CarInterface.

Example from the code (class **CarInterfaceTest**, starting from **row 53**):



```
SpeedTorqueObj test2 = mock(SpeedTorqueObj.class);
when(test2.getSpeed()).thenReturn((double) 5);
when(test2.getTorque()).thenReturn((double) 1);
```

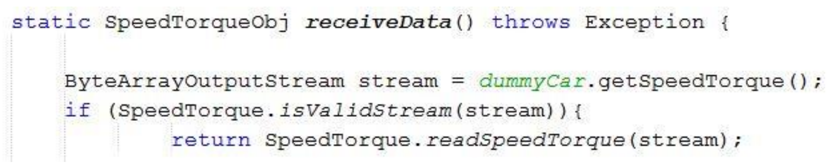
## 3.2 Scenarios and Code Coverage

The initial Coverage of the system gave us green signatures for roughly 90% of the code. Rare cases of red and yellow code occurred in the SensorData class (from Phase 1). CarInterface was, of course, uncovered.

To cover the code of the new modules, the following scenarios were implemented as tests in **CarInterfaceTest** (the following code snippets are taken from the **CarInterface** class):

1. **A valid data packet is received from the car**

   17.9% coverage after implementation. The received data packet is tested and, deemed valid, an object with the values of speed and angle is returned.



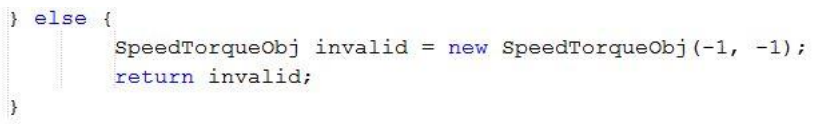
```
static SpeedTorqueObj receiveData() throws Exception {

    ByteArrayOutputStream stream = dummyCar.getSpeedTorque();
    if (SpeedTorque.isValidStream(stream)){
            return SpeedTorque.readSpeedTorque(stream);
```

Row 36

**2. A data packet of invalid length is received from the car**

29.9% coverage after implementation. The data packet is invalid, and an object with negative values is returned.



```
} else {
        SpeedTorqueObj invalid = new SpeedTorqueObj(-1, -1);
        return invalid;
}
```

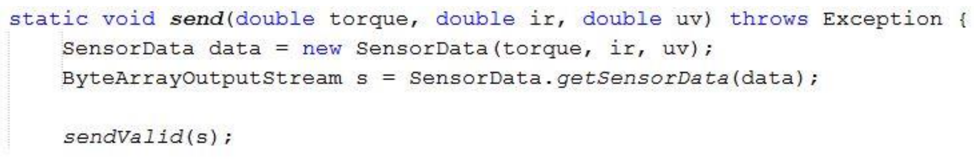
Row 41

**3. A corrupt data packet is received from the car**

29.9% coverage after implementation. Just like in scenario 2, the packet is an invalid stream and the same lines of the code are covered.

**4. A valid data packet is sent to the car**

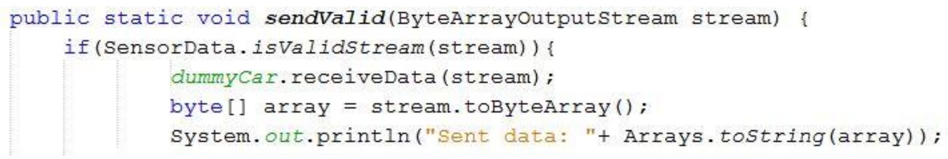
82.1% coverage after implementation. We cover all of the **send** method and most of the **sendValid** method.



```
static void send(double torque, double ir, double uv) throws Exception {
    SensorData data = new SensorData(torque, ir, uv);
    ByteArrayOutputStream s = SensorData.getSensorData(data);

    sendValid(s);
```

Row 56

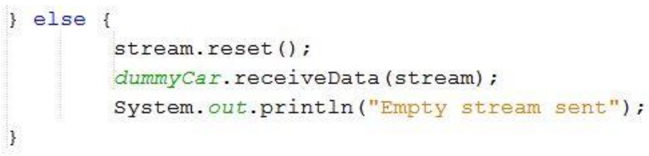


```
public static void sendValid(ByteArrayOutputStream stream) {
    if(SensorData.isValidStream(stream)){
            dummyCar.receiveData(stream);
            byte[] array = stream.toByteArray();
            System.out.println("Sent data: "+ Arrays.toString(array));
```

Row 80

**5. An empty data packet is sent to the car**

95.5% coverage after implementation. The packet is deemed invalid by a method tested in Phase 1, and we cover the corresponding action in the **sendValid** method (send empty stream).



```
} else {
        stream.reset();
        dummyCar.receiveData(stream);
        System.out.println("Empty stream sent");
}
```

Row 85

6. **A corrupt data packet is sent to the car**

   95.5% coverage after implementation. This is also discovered to be an invalid packet, with the same result as in the scenario above.

## Member Contribution

### Development

| | |
|---|---|
| Car: | Kristiyan Dimitrov and Kai Salmon |
| CarInterface: | Kristiyan Dimitrov, Martina Freiholtz and Kai Salmon |
| Gui: | Kai Salmon |
| CarInterfaceTest: | Kristiyan Dimitrov and Martina Freiholtz |
| Coverage Testing: | Kristiyan Dimitrov |

### Documentation

Kristiyan Dimitrov, Martina Freiholtz and Kai Salmon