[DIT085] Testing and Verification 2016-02-25
University of Gothenburg
Group 1.1

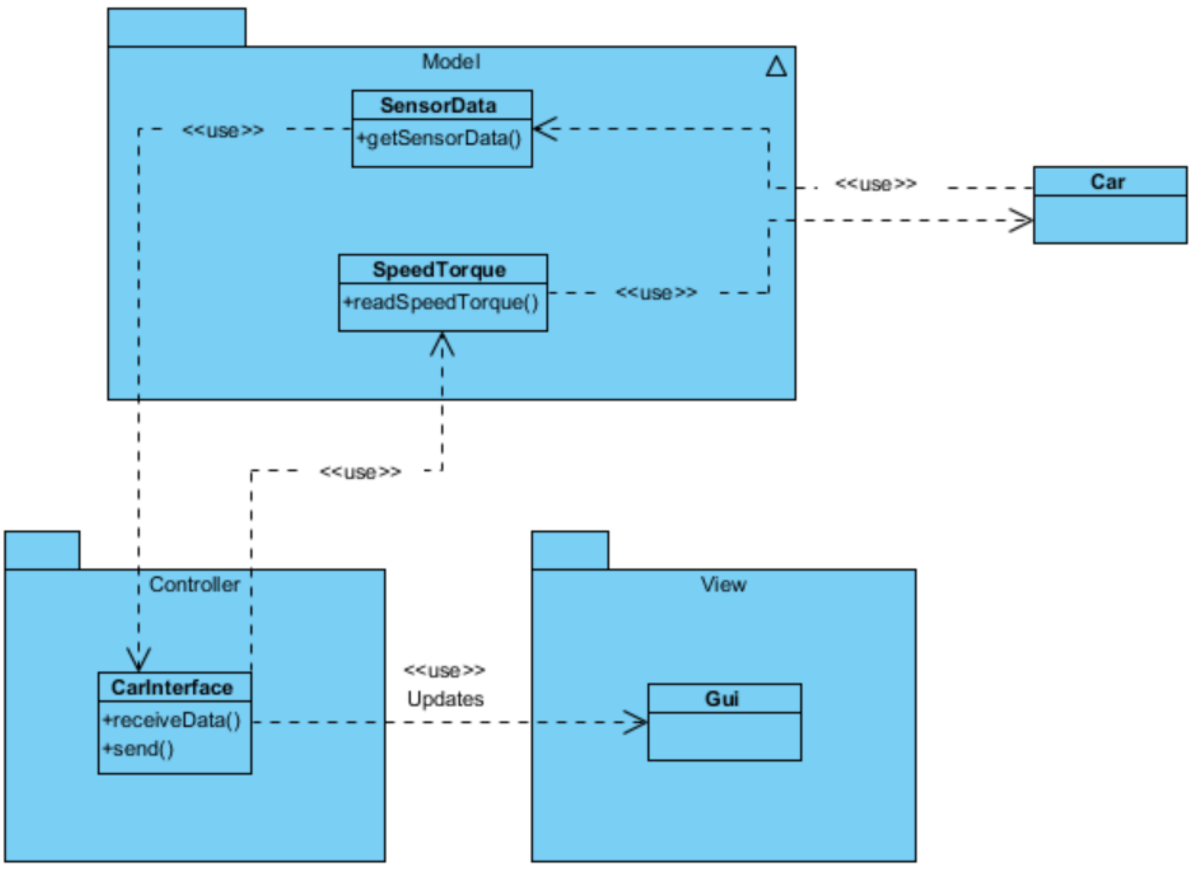
PHASE 2

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 IU), 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

Car

The Car class was made using Mockito.

CarInterface

receiveData

receiveData is called to receive speed and angle data from the car

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, sendValid

The method **send** takes double values as arguments, and is converted into a stream using the method **getSensorData** in class **SensorData**, before being sent to the car. The getSensorData method has already been tested in Phase 1, and should always return a valid stream (such as in TC0).

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.

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

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. It's 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.

3.2 Scenarios and Code Coverage

To cover the code of the new modules, the following scenarios were implemented as tests:

1. A valid data packet is received from the car
2. A data packet of invalid length is received from the car
3. A corrupt data packet is received from the car
4. A valid data packet is sent to the car
5. An empty data packet is sent to the car
6. A corrupt data packet is sent to the car

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).

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