# Testing and verification of embedded systems

Part 1

Jesper Holmblad, Fredrik Mårtensson, Mattias Hallefält

September 29, 2020

# Contents

**1 Introduction** **2**

**2 Interface and Test design** **3**
2.1 Assumptions . . . . . 3
2.2 Interfaces . . . . . 3

**3 Test Driven Development** **7**
3.1 Implementation . . . . . 8
3.2 MoveForward . . . . . 8

**4 Testing** **10**
4.1 Functional testing . . . . . 10
4.2 Structural testing . . . . . 12

**5 Appendix** **13**

**References** **15**

# 1 Introduction

This report is written as a part of the course *Testing and verification of Embedded systems (TVES)* at Halmstad University. The task is to develop testing methods and using *Test-Driven Development (TDD)*. The project is based on information given from the course and the coursebooks: *Introduction to software testing [1],Practical model-based testing: A tools approach [2].* The project consists of planning and developing interfaces and functions through Test-Driven Development (TDD) in connection with testing methods. The project consists of predetermined specification of an automatic parking system being able to park/unpark a car on a 500 meter road with the help of two sensors. In addition to the defined functions that are specified in the project description, extended functions are included in the report. The information must be interpreted and tested through JUnit tests.

# 2 Interface and Test design

## 2.1 Assumptions

Several assumptions to create a clearer definition of the project are made. The assumptions that exist affect the following objects / methods described in table 1

| **Object/method** | **Description** |
|---|---|
| Sensor | Check 1 meter in width at a time |
| Sensor | Both sensors at the same position |
| Sensor | Sensors placed in the front of the car |
| Sensor | Sensor read in centimeter |
| Sensor | If sensor is bad it will continue to send bad reads and be ignored |
| Filter | Standard deviation of input over 10 should be considered faulty |
| Park | The car will park as soon as it finds an available spot |
| Unpark | Check that we are parked before unpark |
| Actuator | Car can move to the left |
| Car | Width of car is 170 cm |
| Car | Car length is the number of meters required to park |

Table 1: Assumptions

## 2.2 Interfaces

Before the choice of test design technique, a framework was implemented for the project. This includes interfaces and abstract classes to be able to communicate between devices within a car. In the classes, A and I present at the beginning of a class to describe Abstract or Interface. The implemented classes are: Car, ACar, CarState, Sensor and Actuator. The UML diagram in figure 1 observes the connections and variables that exist between the different objects.

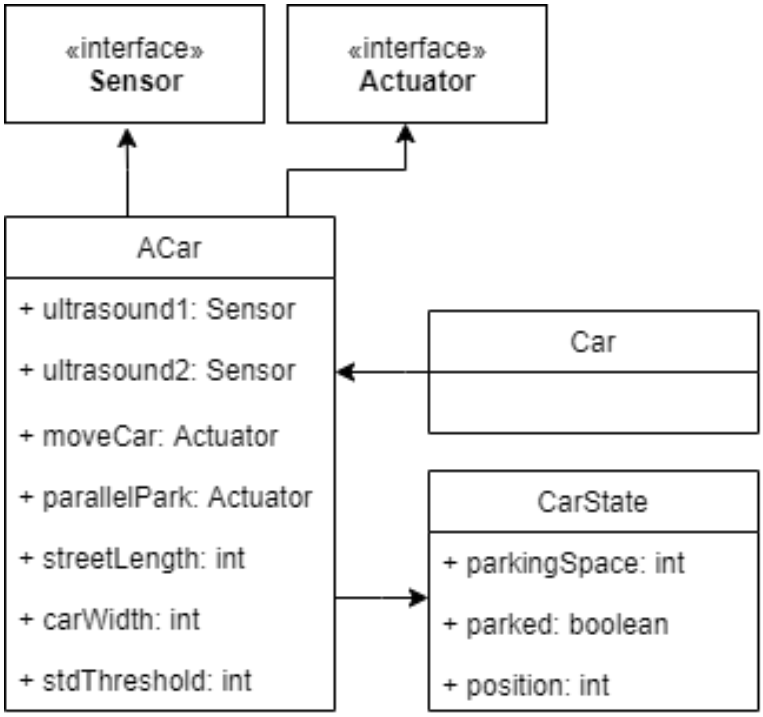



Figure 1: UML diagram of project

In combination with the implementation of interfaces, a description, pre- and post-conditions, return state was declared. An example is shown in Listing 1.

Listing 1: Method specification - MoveForward

```
This method moves the car 1 meter forward, queries the two
   sensors through the isEmpty method described below and
   returns a data structure that contains the current position
   of the car, and the situation of the detected parking places
   up to now. The car cannot be moved forward beyond the end of
   the street.

Pre-conditions:
* position < streetLength
* moveCar != null
* !parked

Post-condition: None

Test-cases:
* moveForward_noErrors_CarState
* moveForward_parked_notPossible
* moveForward_noSpace_CarState
* moveForward_noInit_noInit
* moveForward_brokenActuator_unexpectedState
* moveForward_workingSensors_CarState

@return State of the car
```

The classes and interfaces implemented is described and motivated below.

- **CarState** exists to separate the state of the *Car* from its methods. This way one can return the *CarState* object without giving the recipient access to all the methods. This should technically not be a problem since only copies of the state are to be returned but one cannot be too careful, less is more.
- **ACar** is an abstraction of *Car* that contains the core of the project. Here the requirements can be enforced without getting in depth with the implementation. This is an abstract class instead of an interface so that fields can be declared for loose couplings with other classes.
- **Car** class contains the finished implementation of the methods: moveForward, isEmpty, moveBackward, park, unPark, whereIs as required by the specifications.
- **Sensor**: This interface exists to create a loose coupling for ACar. This interface contains the method read because all sensors are readable.

- **Actuator**: This interface exists to create a loose coupling for ACar. This interface contains the method activate because all actuators can be activated.
- **StatusWrapper**: To avoid throwing exceptions that could crash the code one can use StatusWrapper to pass the exception silently with the return. This is mainly for the sole purpose of testability as it allows more verbose return statements.

The following methods are implemented in addition to the project requirements to improve and facilitate Functionality.

- **Car.readAndFilter**: reads and filters the data returned by the sensor argument. The acceptance criteria of the filter is any value within the span: $mean \pm 2 \times std$. Where mean is the mean value and *std* is the standard deviation from the read values from the sensor. If the *std* is too large, the values from the sensor is ignored.
- **CarState.hashCode**: Turns the fields of CarState into a single 32 bit integer. This was added to simplify the comparator.
- **CarState.compareTo**: Gives a number based on the hash function. Only used to check if 2 CarStates contain the same information which is needed by one of the test cases.

# 3 Test Driven Development

TDD consists of developing tests and code in several iterations in order to reach completeness of structural coverage. This ensures that each step reaches the expected result with high reliability. The approach to implementing TDD followed the steps in figure 2. The use of TDD verifies that the product is made right and meets the expected requirements. TDD provides clear bug reporting in the early stages. TDD also adds the ability to refactor code and minimise the number of bugs that would occur without testing.

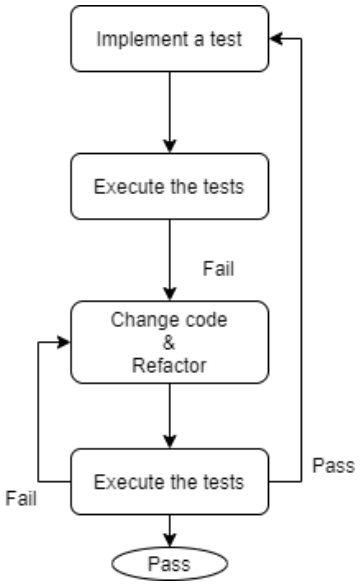



Figure 2: Flowchart of TDD

The five shown steps are repeated and are based on Functional testing and the selected properties to be tested through the decision table. Each part tests a solution to confirm that each part does what it is defined to implement.

The first step is to implement test cases based on the Test design (decision table) to satisfy the specified requirements given in the product description. Then a test is performed for the implementation to ensure the tests fail. Then change the code so it passes the tests. During the change of the code a refactor is used to simplify the code. This is repeated until all code meets the requirements specification. In figure 6 the total number of tests that is supposed to fail is created. In total the JUnit covers 31 different tests (23 Functional and 8 structural).

## 3.1 Implementation

## 3.2 MoveForward

The method for MoveForward exist within ACar which is an abstract class that is extended in Car as shown in figure 1. The abstract class serves as both a template and an interface in this case. Following the diagram of TDD shown in figure 2 the first steps of implementing the method moveForward are the following:

1. Implement a test, shown in listing 2, to move the car when parked. The expected return should be NOT_POSSIBLE.
2. Observe the test fail with listing 3.
3. If fail: Changed and refactored code, shown in listing 4.
4. Observe the test pass.
5. Repeat step one of TDD and implement a second test, shown in listing 5, and continue with code from listing 4. The expected return should be NOT_POSSIBLE.
6. If fail: Changed and refactored code, shown in listing 6.
7. Observe the test pass.
8. Repeat step 1-4 with new tests until all conditions are met.

Listing 2: Test: moveForward_parked_notPossible

```
@Test
public void moveForward_parked_notPossible() {
    wroom.setPosition(wroom.getStreetLength()-1);
    wroom.setMoveCar(workingActuator);
    wroom.setParked(true);
    assert (wroom.moveForward().getStatus().equals(autopark.
       StatusWrapper.NOT_POSSIBLE));
}
```

Listing 3: Code: MoveForward - Fail

```
public StatusWrapper<CarState> moveForward() {
    return new StatusWrapper<>(new CarState(this));
}
```

Listing 4: Code: MoveForward - Succeed

```
public StatusWrapper<CarState> moveForward() {
    if (this.isParked()) {
        // To satisfy test: moveForward_parked_notPossible
        return new StatusWrapper<>(new CarState(this), autopark.
            StatusWrapper.NOT_POSSIBLE, "Car is parked");
    }
    return new StatusWrapper<>(new CarState(this));
}
```

Listing 5: Test: moveBackward_noSpace_notPossible

```
@Test
public void moveBackward_noSpace_notPossible() {
    wroom.setParked(false);
    wroom.setMoveCar(workingActuator);
    wroom.setPosition(0);
    assert (wroom.moveBackward().getStatus().equals(autopark.
        StatusWrapper.NOT_POSSIBLE));
}
```

Listing 6: Code: MoveForward - Succeed

```
public CarState moveForward() {
    if (this.isParked()) {
        // To satisfy test: moveForward_parked_notPossible
        return new StatusWrapper<>(new CarState(this), autopark.
            StatusWrapper.NOT_POSSIBLE, "Car is parked");
    }
    if (!(this.getPosition() < this.getStreetLength())) {
        // To satisfy test: moveForward_noSpace_notPossible
        return new StatusWrapper<>(new CarState(this), autopark.
            StatusWrapper.NOT_POSSIBLE, "End of streetLength");
    }
    return new StatusWrapper<>(new CarState(this));
}
```

# 4 Testing

The decision to use Functional testing and Structural testing is based upon a quote from the course lectures: "No perfect Functional testing technique exists: a combination of classification tree (or DT) with others should provide an effective mix". This implies one should use multiple testing methods. The chosen methods are: Decision table for Functional coverage and path node test for Structural coverage. The Functional testing is used to cover validation and Structural testing to cover verification.

## 4.1 Functional testing

Decision table is used in Functional testing to test the Functionality of the methods that have been provided through the project requirements. Based on the test design in table 2-7 it is possible to observe which parts of each method are tested to cover the largest possible range of combinations.

| | | | | | | |
|---|---|---|---|---|---|---|
| C1: | position <streetLength | y | y | y | y | n |
| C2: | moveCar != null | y | y | y | n | - |
| C3: | !parked | y | y | n | - | - |
| C4: | moveCar.activate() | y | n | - | - | - |
| A1: | return CarState | x | - | - | - | - |
| A2: | return NOT_POSSIBLE | - | - | x | - | x |
| A3: | return NO_INIT | - | - | - | x | - |
| A4: | return UNEXPECTED_STATE | - | x | - | - | - |

Table 2: Decision table for moveForward: C contains the Conditions that should be met and A is expected Action that should be returned.

| | | | | | | |
|---|---|---|---|---|---|---|
| C1: | 0 <position | y | y | y | y | n |
| C2: | moveCar != null | y | y | y | n | - |
| C3: | !parked | y | y | n | - | - |
| C4: | moveCar.activate() | y | n | - | - | - |
| A1: | return CarState | x | - | - | - | - |
| A2: | return NOT_POSSIBLE | - | - | x | - | x |
| A3: | return NO_INIT | - | - | - | x | - |
| A4: | return UNEXPECTED_STATE | - | x | - | - | - |

Table 3: Decision table for moveBackward: C contains the Conditions that should be met and A is expected Action that should be returned.

| | | | | | |
|---|---|---|---|---|---|
| C1: | ultrasound1 != null | y | y | y | n |
| C2: | ultrasound2 != null | y | y | n | - |
| C3: | sensorRead >= carWidth | y | n | - | - |
| A1: | return True | x | - | - | - |
| A2: | return False | - | x | - | - |
| A3: | return NO_INIT | - | - | x | x |

Table 4: Decision table for isEmpty: C contains the Conditions that should be met and A is expected Action that should be returned.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| C1: | !parked | y | y | y | y | y | n |
| C2: | parallelPark != null | y | y | y | y | n | - |
| C3: | 5 <= parkingSpace | y | y | y | n | - | - |
| C4: | parked | y | y | n | - | - | - |
| C5: | parallelPark.activate() | y | n | - | - | - | - |
| A1: | return True | x | - | - | - | - | - |
| A2: | return NOT_POSSIBLE | - | - | - | x | - | x |
| A3: | return NO_INIT | - | - | - | - | x | - |
| A4: | return UNEXPECTED_STATE | - | x | x | - | - | - |

Table 5: Decision table for park: C contains the Conditions that should be met and A is expected Action that should be returned.

| | | | | | | |
|---|---|---|---|---|---|---|
| C1: | parked | y | y | y | y | n |
| C2: | moveCar != null | y | y | y | n | - |
| C3: | !parked | y | y | n | - | - |
| C4: | moveCar.activate() | y | n | - | - | - |
| A1: | return True | x | - | - | - | - |
| A2: | return NOT_POSSIBLE | - | - | - | - | x |
| A3: | return NO_INIT | - | - | - | x | - |
| A4: | return UNEXPECTED_STATE | - | x | x | - | - |

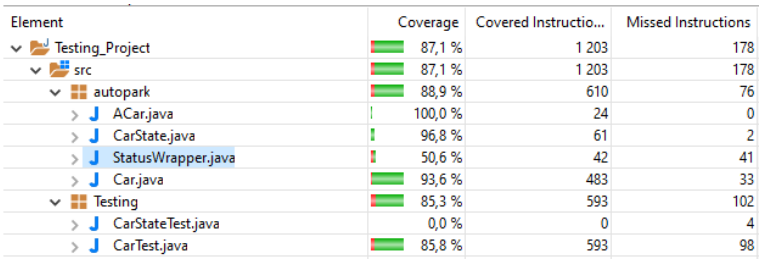
Table 6: Decision table for unPark: C contains the Conditions that should be met and A is expected Action that should be returned.

| | | | |
|---|---|---|---|
| C1: | this.compareTo(CarState) == 0 | y | n |
| A1: | return CarState | x | x |

Table 7: Decision table for whereIs: C contains the Conditions that should be met and A is expected Action that should be returned.

## 4.2 Structural testing

Structural testing focuses on path coverage. For simplicity only node coverage is done. With the help of EclEmma, which is a plugin for Eclipse, it is possible to cover the parts of the code that are not reached. This improves the quality of the unit test by increasing the code coverage. In figure 3 the test coverage is observed without applying structural testing. In figure 4 the coverage after Structural testing is applied. Within figure 5 three states is observed as three different colours. The errors that is reported gives an understanding on which parts that the Functional testing does not cover. Before Structural testing the 23 test created for Functional testing coverts 88.9% and with Structural testing it is reached to 99.9% coverage. The JUnit tests can be observed in figure 7.



| Element | Coverage | Covered Instructio... | Missed Instructions |
|---|---|---|---|
| Testing_Project | 87,1 % | 1 203 | 178 |
| src | 87,1 % | 1 203 | 178 |
| autopark | 88,9 % | 610 | 76 |
| ACar.java | 100,0 % | 24 | 0 |
| CarState.java | 96,8 % | 61 | 2 |
| StatusWrapper.java | 50,6 % | 42 | 41 |
| Car.java | 93,6 % | 483 | 33 |
| Testing | 85,3 % | 593 | 102 |
| CarStateTest.java | 0,0 % | 0 | 4 |
| CarTest.java | 85,8 % | 593 | 98 |

Figure 3: Completeness of project before Structural testing

| | | | |
|---|---|---|---|
| Testing_Project | 91,5 % | 1 499 | 139 |
| src | 91,5 % | 1 499 | 139 |
| autopark | 99,9 % | 685 | 1 |
| ACar.java | 100,0 % | 24 | 0 |
| CarState.java | 100,0 % | 63 | 0 |
| StatusWrapper.java | 100,0 % | 83 | 0 |
| Car.java | 99,8 % | 515 | 1 |
| Testing | 85,5 % | 814 | 138 |
| CarStateTest.java | 0,0 % | 0 | 4 |
| CarTest.java | 85,9 % | 814 | 134 |

Figure 4: Completeness of project after Structural testing

```
@Override
public StatusWrapper<Boolean> isEmpty() {
    // Check pre conditions
    if (!(ultrasound1 instanceof Sensor)) {
        return new StatusWrapper<>(false, autopark.StatusWrapper.NO_INIT,
                "ultrasound1 is not an instance of Sensor");
    }
    if (!(ultrasound2 instanceof Sensor)) {
        return new StatusWrapper<>(false, autopark.StatusWrapper.NO_INIT,
                "ultrasound2 is not an instance of Sensor");
    }

    // Read values
    Optional<Double> read1 = readAndFilter(ultrasound1);
    Optional<Double> read2 = readAndFilter(ultrasound2);
```

Figure 5: Example of ElcEmma producing error. Green is no error, yellow that some state/states is missing on testing and red that the code is never reached.

# 5 Appendix

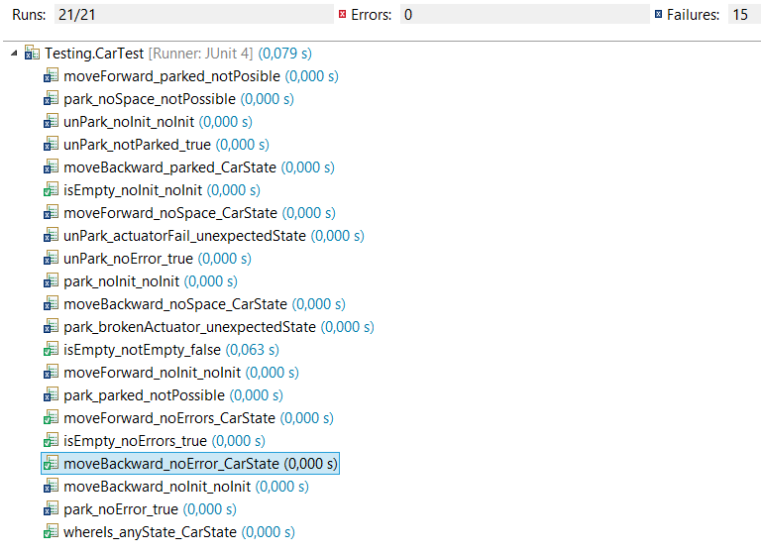



Figure 6: All failing tests that should be implemented

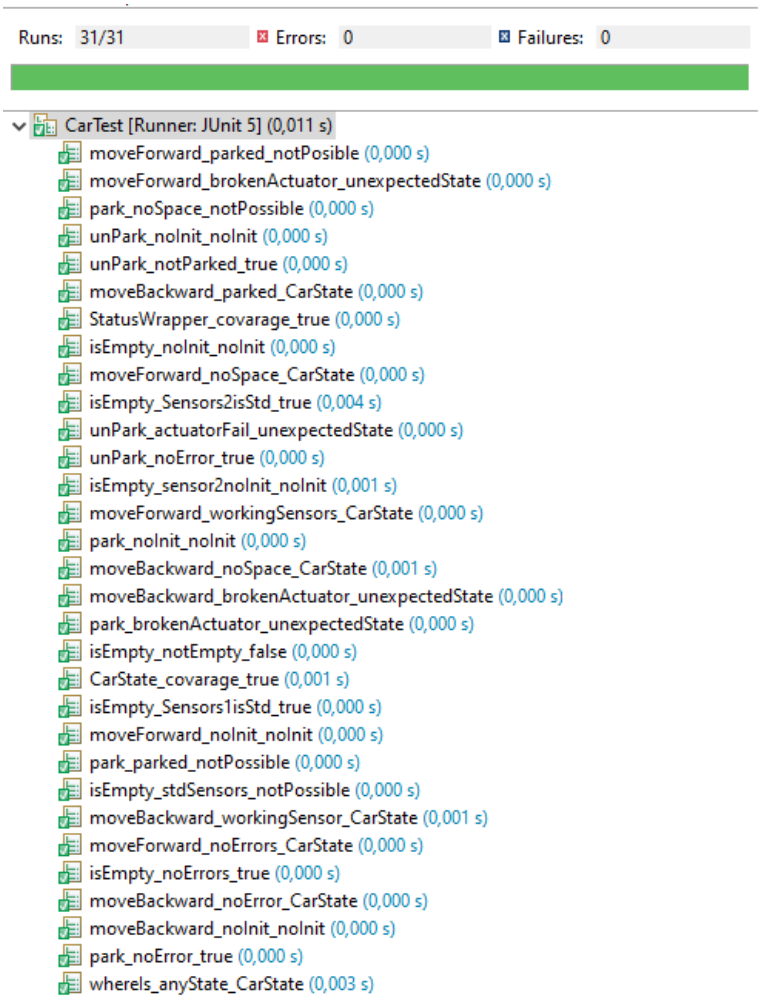



Figure 7: Completeness of project after Structural testing

# References

[1] Ammann P, Offutt J. Introduction to software testing. Cambridge University Press; 2016.

[2] Legeard B, Utting M. Practical model-based testing: A tools approach. Morgan Kaufmann Publishers; 2006.