
This document contains an overview of the important tests for the Car, Section and Track classes as part of a Test-Driven Development (TDD) approach.
Each test was written first and then the code was adjusted so that all tests were successful.

1.) Important Additional Tests

To further validate the robustness and functionality of the program, the following tests can be implemented:

- ***Gear Transition Speed Test***:

  > Checks whether the speed resets or adjusts correctly when the gear changes from high to low.

- ***Test for consistency of the dice value***:

  > Checks whether rolling the dice always results in a number between 1 and 6 and that there are no exceptions or unexpected values.

- ***Section Maximum Speed Test***:

  > Verify that the car's speed does not exceed the maximum speed of a section.

- ***Test for gear change exception***:

  > Checks whether ArgumentException is thrown correctly when invalid gear values (e.g. above 6) are set.

- ***Speed test after multiple accelerations without changing gear***:

  > Checks whether the speed behaves consistently when 'Accelerate' is called up several times.

2.) Tests for the Car class

- ***ItShouldStandStill_GivenCreated***:

  > Ensures that the Speed of the car is 0, when created.

  ▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldStandStill_GivenCreated()
{
    // ARRANGE - Erstellen eines neuen Autos
    Car car = new();

    // ACT - Abfrage der Anfangsgeschwindigkeit des Autos
    int actualSpeed = car.Speed;

    // ASSERT - Überprüfen, ob die Geschwindigkeit 0 ist
    Assert.AreEqual(0 , actualSpeed); // Erwartung: Geschwindigkeit ist 0.
}
```

- ***ItShouldStore_GivenGearBetweenOneAndSix***:

  > Ensures that a valid `Gear` is stored as expected.

  ▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldStore_GivenGearBetweenOneAndSix()
{
    // ARRANGE - Erstellen eines neuen Autos
    Car car = new()
    {
        // ACT - Setzen des Gangs auf einen gültigen Wert (z. B. 6)
        Gear = 6
    };

    // ASSERT - Überprüfen, ob der Gang korrekt gespeichert wurde
    Assert.AreEqual(6 , car.Gear); // Erwartung: Gang ist 6.
}
```

- ***ItShouldThrowAnExpection_GivenGearOutsideRange***:

  > Ensures that a `ExpectedException(typeof(ArgumentException)` is thrown, if an invalid `Gear` was set.

  ▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
[ExpectedException(typeof(ArgumentException) , "Gear should be between 0 and 6")]
0 references
public void ItShouldThrowAnExpection_GivenGearOutsideRange()
{
    // ARRANGE - Erstellen eines neuen Autos
    Car car = new()
    {
        // ACT - Setzen eines ungültigen Gangwerts (z. B. 7)
        Gear = 7 // Erwartung: ArgumentException.
    };

    // ASSERT - Überprüfung erfolgt durch das ExpectedException-Attribut
}
```

- ***ItShouldHaveASpeedOfZero_GivenNoAcceleration***:

  > Ensures that the `Speed` of a car is 0, if not accelerated.

▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldHaveASpeedOfZero_GivenNoAcceleration()
{
    // ARRANGE - Erstellen eines neuen Autos und Setzen des Gangs
    Car car = new()
    {
        Gear = 3
    };

    // ACT - Keine Beschleunigung durchführen

    // ASSERT - Überprüfen, ob die Geschwindigkeit 0 ist
    Assert.IsTrue(car.Speed == 0); // Erwartung: Geschwindigkeit ist 0.
}
```

- ***ItShouldHaveASpeedBetween30And180_GivenGear3AndAccelerated***:

> Ensures that `Accelerate()` works as intended, if a valid `Gear` was chosen and the new `Speed` of the car is in a valid range.

▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldHaveASpeedBetween30And180_GivenGear3AndAccelerated()
{
    // ARRANGE - Erstellen eines neuen Autos und Setzen des Gangs
    Car car = new()
    {
        Gear = 3
    };

    // ACT - Beschleunigung durchführen
    car.Accelerate();

    // ASSERT - Überprüfen, ob die Geschwindigkeit im erwarteten Bereich liegt
    Assert.IsTrue(car.Speed >= 30 && car.Speed <= 180); // Erwartung: Geschwindigkeit zwischen 30 und 180.
}
```

- ***ItShouldHaveASpeedOf60_GivenGear3AndDiceShowsTwoDots***:

> Ensures that the Acceleration and `Speed` was computed correctly.

▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldHaveASpeedOf60_GivenGear3AndDiceShowsTwoDots()
{
    // ARRANGE - Erstellen eines Autos mit einem FakeDice und Setzen des Gangs
    FakeDice fakeDice = new() { Dots = 2 }; // Setzt die Würfelaugen auf 2
    Car car = new(fakeDice)
    {
        Gear = 3
    };

    // ACT - Beschleunigung durchführen
    car.Accelerate();

    // ASSERT - Überprüfen, ob die Geschwindigkeit korrekt berechnet wurde (3 * 10 * 2 = 60)
    Assert.AreEqual(60 , car.Speed); // Erwartung: Geschwindigkeit ist 60.
}
```

- ***ItShouldCallDiceRoll_GivenAccelerateIsCalled***:

> Ensures that `Dice.RollWasCalled` is `true` if the car has accelerated.

▶ $\color{lightgreen}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldCallDiceRoll_GivenAccelerateIsCalled()
{
  // ARRANGE - Erstellen eines Autos mit einem FakeDice
  FakeDice fakeDice = new();
  Car car = new(fakeDice);

  // ACT - Beschleunigung durchführen
  car.Accelerate();

  // ASSERT - Überprüfen, ob Roll() aufgerufen wurde
  Assert.IsTrue(fakeDice.RollWasCalled); // Erwartung: Roll() wurde aufgerufen.
}
```

• 3.) Tests for the Section class

- ***ItShouldHaveALengthAndAMaxSpeed_GivenObjectCreated***:

> Ensures that `MaxSpeed` and `Length` are set correctly.

▶ $\color{green}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldHaveALengthAndAMaxSpeed_GivenObjectCreated()
{
  // ARRANGE - Setzen der Testdaten für MaxSpeed und Länge
  var someSpeed = 60;
  var someLength = 400;

  // ACT - Erstellen eines neuen Section-Objekts mit den Testdaten
  Section section = new(someSpeed , someLength);

  // ASSERT - Überprüfen, ob MaxSpeed und Länge korrekt gesetzt wurden
  Assert.AreEqual(someSpeed , section.MaxSpeed);  // Erwartung: MaxSpeed ist 60.
  Assert.AreEqual(someLength , section.Length);   // Erwartung: Länge ist 400.
}
```

- ***ItShouldConnectASectionAfterTheCurrentSection_GivenAddAfterMeIsCalled***:

> Ensures that connecting Sections with `AddAfterMe()` works as expected.

▶ $\color{green}{click\ for\ Screenshot}$

```
[TestMethod]
0 references
public void ItShouldConnectASectionAfterTheCurrentSection_GivenAddAfterMeIsCalled()
{
  Section
    section = new(60 , 400),
    nextSection = new(60 , 400);

  section.AddAfterMe(nextSection);

  Assert.AreEqual(nextSection , section.NextSection);
  Assert.AreEqual(section , nextSection.PreviousSection);
}
```

- ***ItShouldConnectASectionBeforeTheCurrentSection_GivenAddBeforeMeIsCalled*** :

  > Ensures that connecting Sections with `AddBeforeMe()` works as expected.

  ▶ $\color{green}{click\ for\ Screenshot}$

```
[TestMethod]
public void ItShouldConnectASectionBeforeTheCurrentSection_GivenAddBeforeMeIsCalled()
{
  Section
    section = new(60 , 400),
    previousSection = new(60 , 400);

  section.AddBeforeMe(previousSection);

  Assert.AreEqual(previousSection , section.PreviousSection);
}
```

- ***ItShouldInsertASectionBetweenTwoSections_GivenTwoConnectedSectionsAndAddAfterMeIsCalled***:

  > Ensures that connecting Sections with `AddAfterMe()` reconnects given Sections as expected.

  ▶ $\color{green}{click\ for\ Screenshot}$

```
[TestMethod]
public void ItShouldInsertASectionBetweenTwoSections_GivenTwoConnectedSectionsAndAddAfterMeIsCalled()
{
  Section
    sectionOne = new(60 , 400),
    sectionTwo = new(60 , 500),
    insertSection = new(50 , 300);

  sectionOne.AddAfterMe(sectionTwo);
  sectionOne.AddAfterMe(insertSection);

  Assert.AreEqual(sectionTwo , sectionOne.NextSection!.NextSection);
}
```

- ***ItShouldInsertASectionBetweenTwoSections_GivenTwoConnectedSectionsAndAddbeforeMeIsCalled***:

  > Ensures that connecting Sections with `AddBeforeMe()` reconnects given Sections as expected.

  ▶ $\color{green}{click\ for\ Screenshot}$

```
[TestMethod]
public void ItShouldInsertASectionBetweenTwoSections_GivenTwoConnectedSectionsAndAddbeforeMeIsCalled()
{
  Section
    sectionOne = new(60 , 400),
    sectionTwo = new(60 , 500),
    insertSection = new(50 , 300);

  sectionOne.AddAfterMe(sectionTwo);
  sectionTwo.AddBeforeMe(insertSection);

  Assert.AreEqual(sectionTwo , sectionOne.NextSection!.NextSection);
}
```

- 3.1) Additional tests for Section

  - ***ItShouldThrowException_GivenNegativeMaxSpeed***:

> Checks whether negative MaxSpeed values throw an exception.

- ***ItShouldThrowException_GivenNegativeLength***:

  > Checks whether negative length values throw an exception.

- *Minimum Length Test* (***ItShouldThrowException_GivenLengthLessThanMinimum***):

  > Tests whether an exception is thrown if Length is too low.

- *Limit for MaxSpeed* (***ItShouldThrowException_GivenMaxSpeedExceedsLimit***):

  > Checks whether MaxSpeed does not exceed the limit.

4.) Tests for the Track class

If a track class contains multiple section instances, the following tests may be useful:

- *Total length test* (***ItShouldReturnTotalLength_GivenMultipleSections***):

  > Checks whether GetTotalLength() calculates the correct total length of the sections.

- *Maximum speed test* (***ItShouldReturnMaxSpeed_GivenMultipleSections***):

  > Checks whether GetMaxSpeed() returns the highest allowed speed in the track.

- *Empty Section List Test* (***ItShouldThrowException_GivenEmptySectionList***):

  > Ensure an exception is thrown when track is created without sections.

- *Test for null sections in the list* (***ItShouldThrowException_GivenNullSectionInList***):

  > Checks whether no null objects are accepted as a section.

5.) Tests for the TrackBuilder class

- *Conected Track Build test* (***ItShouldBuildAConnectedTrack_GivenSectionInformation***):

  > Checks if the TrackBuilder builds (connects) a Track with given Sections as expected.

  ▶ click for Screenshot

```
[TestMethod]
public void ItShouldBuildAConnectedTrack_GivenSectionInformation()
{
  (int, int)[ ] sectionInfos = [ (10, 10) , (20, 20) , (30, 30) ];

  TrackBuilder builder = new(sectionInfos);

  Section
    startSection = new(sectionInfos[ 0 ].Item1 , sectionInfos[ 0 ].Item2),
    secondSection = new(sectionInfos[ 1 ].Item1 , sectionInfos[ 1 ].Item2),
    thirdSection = new(sectionInfos[ 2 ].Item1 , sectionInfos[ 2 ].Item2);

  startSection.AddAfterMe(secondSection);
  secondSection.AddAfterMe(thirdSection);
  Track manuallyBuiltTrack = new([ startSection , secondSection , thirdSection ]);

  Assert.AreEqual(10 , manuallyBuiltTrack.StartSection!.Length);
  Assert.AreEqual(10 , manuallyBuiltTrack.StartSection.MaxSpeed);
  Assert.AreEqual(startSection , manuallyBuiltTrack.StartSection);
  Assert.AreEqual(secondSection , manuallyBuiltTrack.StartSection.NextSection);

  Assert.AreEqual(10 , builder.RaceTrack!.StartSection!.Length);
  Assert.AreEqual(10 , builder.RaceTrack.StartSection.MaxSpeed);

  Assert.AreEqual(manuallyBuiltTrack.StartSection.Length , builder.RaceTrack.StartSection.Length);
  Assert.AreEqual(manuallyBuiltTrack.StartSection.MaxSpeed , builder.RaceTrack.StartSection.MaxSpeed);
  Assert.AreEqual(manuallyBuiltTrack.StartSection.NextSection!.Length , builder.RaceTrack.StartSection.NextSection!.Length);
  Assert.AreEqual(manuallyBuiltTrack.StartSection.NextSection.MaxSpeed , builder.RaceTrack.StartSection.NextSection.MaxSpeed);
  Assert.AreEqual(manuallyBuiltTrack.StartSection.NextSection.NextSection!.Length , builder.RaceTrack.StartSection.NextSection.NextSection!.Length);
  Assert.AreEqual(manuallyBuiltTrack.StartSection.NextSection.NextSection.MaxSpeed , builder.RaceTrack.StartSection.NextSection.NextSection.MaxSpeed);
}
```

- # 6.) Summary

These tests cover the most important requirements and error conditions of the Car, Section and Track classes.
They validate the basic logic and robustness of the program and ensure that the application remains stable even with invalid inputs.