

**FH-OÖ Hagenberg/HSD**  
**SDP3, WS 2025**  
**Übung 5**



---

Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku

---

Mat.Nr: 2410306027 / 2410306014

Punkte:

---

Übungsgruppe: 1

korrigiert:

---

**Beispiel 1 (24 Punkte) Fahrsimulation:** Entwerfen Sie aus der nachfolgenden Spezifikation ein Klassendiagramm, instanzieren Sie dieses und implementieren Sie die Funktionalität entsprechend:

Für einen KFZ-Prüfstand ist eine Software zu entwickeln. Dabei sollen Sensorwerte zur Ermittlung der Raddrehzahlen eingelesen und verarbeitet werden. Die Aufbereitung der Daten umfasst die Berechnung der Momentangeschwindigkeit (Tachometeranzeige) und der zurückgelegten Wegstrecke (Kilometerzähler; grobe Näherung genügt).

Modellieren Sie nun einen PKW als Konkretisierung eines KFZ. Der PKW hat einen Tachometer und einen Kilometerzähler. Weiters verfügt der PKW über einen Raddrehsensor, der bei jedem Abruf die momentane Raddrehzahl eines der vier Räder zurückliefert. Bei jedem Abruf der Raddrehzahl sollen die Anzeigen für den Tachometer und die Kilometeranzeige automatisch aktualisiert werden. Verwenden Sie dazu ein passendes Design Pattern.

Der Raddrehsensor wird hier in der Simulation durch eine Datei nachgebildet. Jede Zeile der Datei entspricht dabei einer momentanen Drehzahl. Kapseln Sie den Sensor in einer Klasse. In der Schnittstelle dieser Klasse befindet sich eine Methode `GetRevolutions()`, die einen Drehzahlwert von der Datei einliest und zurückgibt.

Die Klasse PKW verfügt über eine Methode `Process()`, die vom Testtreiber fix im 500ms-Takt aufgerufen wird. `Process()` liest jedes Mal einen neuen Drehzahlwert via `GetRevolutions()` ein und legt diesen Wert lokal in einem Member ab. Weiters ist eine Methode `GetCurrentSpeed()` vorzusehen, die anhand folgender technischer Daten die Momentangeschwindigkeit ermittelt:

Einheit der Raddrehzahl: U/min

Raddurchmesser: 600mm

Ermitteln Sie die Momentangeschwindigkeit entweder in m/s oder km/h.

Bsp: 500 U/min

$$v = (500 / 60) \times 0,6\text{m} \times \pi \times 3,6 = 56,549 \text{ km/h}$$

Die Tachometer- und Kilometeranzeige holen sich im Zuge einer Benachrichtigung den aktuellen Geschwindigkeitswert. Die Kilometeranzeige berechnet auf Basis des 500ms-Taktes die gefahrene Wegstrecke:

$$(56,549 / 3,6) / 2 = 7,854\text{m} \text{ (grobe Näherung)}$$

Simulieren Sie den Tachometer mit Hilfe der analogen Anzeige (AnalogDisplay) und die Kilometeranzeige mit Hilfe der digitalen Anzeige (DigitalDisplay).

Schreiben Sie einen Testtreiber, der wiederholt im 500ms-Takt die Methode `Process()` vom PKW aufruft.

Die Displays müssen nicht mitabgegeben werden. Die mitgelieferten Klassen für die Ansteuerung der Displays sind elektronisch mitabzugeben, allerdings kann auf einen Ausdruck im pdf verzichtet werden!

Treffen Sie für alle unzureichenden Angaben sinnvolle Annahmen und begründen Sie diese. Verfassen Sie weiters eine Systemdokumentation (entsprechend den Vorgaben aus Übung1)!

**Allgemeine Hinweise:** Legen Sie bei der Erstellung Ihrer Übung großen Wert auf eine **saubere Strukturierung** und auf eine **sorgfältige Ausarbeitung!** Dokumentieren Sie alle Schnittstellen und versehen Sie Ihre Algorithmen an entscheidenden Stellen ausführlich mit Kommentaren! Testen Sie ihre Implementierungen ausführlich! Geben Sie den **Testoutput** mit ab!



# **Systemdokumentation Projekt DriveSim**

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 24. November 2025

# Inhaltsverzeichnis

<b>1 Organisatorisches</b>	<b>5</b>
1.1 Team . . . . .	5
1.2 Aufteilung der Verantwortlichkeitsbereiche . . . . .	5
1.3 Aufwand . . . . .	6
<b>2 Anforderungsdefinition (Systemspezifikation)</b>	<b>7</b>
<b>3 Systementwurf</b>	<b>8</b>
3.1 Designentscheidungen . . . . .	8
<b>4 Dateibeschreibung</b>	<b>10</b>
4.1 Datei: rmp_data.txt . . . . .	10
<b>5 Dokumentation der Komponenten (Klassen)</b>	<b>11</b>
<b>6 Testprotokollierung</b>	<b>12</b>
<b>7 Quellcode</b>	<b>16</b>
7.1 Object.h . . . . .	16
7.2 Vehicle.hpp . . . . .	17
7.3 Vehicle.cpp . . . . .	19
7.4 Car.hpp . . . . .	20
7.5 Car.cpp . . . . .	22
7.6 IDisplay.hpp . . . . .	23
7.7 Meter.hpp . . . . .	24
7.8 Meter.cpp . . . . .	25
7.9 Tachometer.hpp . . . . .	26
7.10 Tachometer.cpp . . . . .	28
7.11 Odometer.hpp . . . . .	29
7.12 Odometer.cpp . . . . .	31
7.13 RPM_Sensor.hpp . . . . .	32
7.14 RPM_Sensor.cpp . . . . .	33
7.15 main.cpp . . . . .	34
7.16 Test.hpp . . . . .	45

# 1 Organisatorisches

## 1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: S2410306027@fh-hagenberg.at
- Simon Vogelhuber, Matr.-Nr.: S2410306014, E-Mail: S2410306014@fh-hagenberg.at

## 1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
  - Design Klassendiagramm
  - Implementierung des Testtreibers
  - Dokumentation
  - Implementierung und Test der Klassen:
    - \* Object,
    - \* Vehicle,
    - \* Car,
    - \* Meter,
    - \* IDisplay,
    - \* Tachometer,
    - \* Odometer,
- Simon Vogelhuber

- Design Klassendiagramm
- Implementierung des Testtreibers
- Dokumentation
- Implementierung und Test der Klassen:
  - \* Object,
  - \* Vehicle,
  - \* Car,
  - \* Meter,
  - \* IDisplay,
  - \* RPM Sensor

### 1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich 8 Ph
- Simon Vogelhuber: geschätzt 10 Ph / tatsächlich 6 Ph

## 2 Anforderungsdefinition (Systemspezifikation)

In diesem Abschnitt werden die funktionalen Anforderungen an das System beschrieben.

### Funktionen des Vehicle

- Abstrakte Basisklasse für alle Fahrzeugtypen
- Anfragen der Sensordaten (Drehzahl der Reifen)

### Funktionen des Sensors

- Erzeugen der Sensordaten (Drehzahl der Reifen) durch lesen einer Textdatei
- Weitergabe der Sensordaten an die verbauten Anzeigen

### Funktionen des Display

- Interface für alle Anzeigetypen für die Verwendung eines Displays innerhalb eines Vehicles

### Funktionen des Meter

- Abstrakte Basisklasse für alle Anzeigetypen
- Abfragen der anzuzeigenden Werte vom Vehicle
- Darstellung der Werte

### Funktionen des Odometer

- Berechnung der gefahrenen Strecke anhand der Raddrehzahl

### Funktionen des Tachometer

- Anzeige der Fahrzeuggeschwindigkeit

## 3 Systementwurf

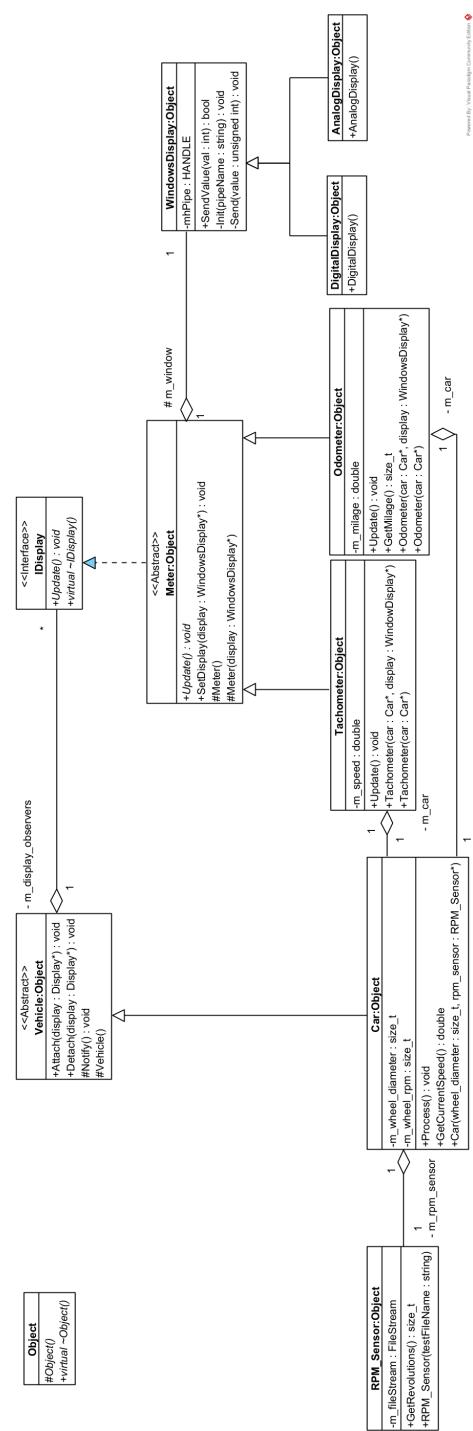
### 3.1 Designentscheidungen

Im Klassendiagramm wurde als Pattern das *Observer Pattern* verwendet, da in diesem Fall ein Fahrzeug mehrere Anzeigen (Meter) haben kann, welche auf die Daten des Fahrzeugs reagieren müssen. Das Observer Pattern ermöglicht es, dass die Anzeigen sich beim Fahrzeug registrieren können und bei einer Änderung der Sensordaten automatisch benachrichtigt werden.

Die Displays können vom Fahrzeug die aufbereiteten Daten (z.B. Geschwindigkeit, Drehzahl) abfragen, indem sie die entsprechenden Methoden des Fahrzeugs aufrufen.

Die abstrakte Basisklasse *Meter* dient dazu, die gemeinsamen Funktionen der verschiedenen Anzeigetypen (Tachometer, Odometer, RPM Sensor) zu bündeln. Weiters wird dadurch die Erweiterbarkeit des Systems verbessert, da neue Anzeigetypen leicht hinzugefügt werden können, indem sie von der Basisklasse erben.

## 3.2 Klassendiagramm



## 4 Dateibeschreibung

Im folgenden Abschnitt werden die Formate der verwendeten Dateien beschrieben.

### 4.1 Datei: rmp\_data.txt

Diese Datei wird vom RPM - Sensor verwendet, um die Testdaten für die Rad-drehzahl zu lesen. Jeder Eintrag enthält eine Ganzzahl, die die Raddrehzahl in Umdrehungen pro Minute (RPM) angibt.

```
RPM_Data = Digit {Digit}.\nDigit     = "0"..."9".
```

## 5 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis [./doxy/html/index.html](#)

## 6 Testprotokollierung

```
1 ****
2 **** TESTCASE START
3 ****
4 ****
5
6 Test normal operation in Odometer Setup
7 [Test OK] Result: (Expected: true == Result: true)
8
9 Test nullptr Car in Odometer CTOR
10 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
11
12 Test Display nullptr in CTOR of Odometer
13 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
14
15 Test nullptr in CTOR of Odometer
16 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
17
18 Test Car nullptr in Update of Odometer
19 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
20
21 Test nullptr in Set Display
22 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
23
24 Test initial Milage of Odometer
25 [Test OK] Result: (Expected: 0 == Result: 0)
26
27 Test Milage after one Process of Odometer
28 [Test OK] Result: (Expected: 26 == Result: 26)
29
30 Test for Exception in Testcase
31 [Test OK] Result: (Expected: true == Result: true)
32
33
34 ****
35
36
37 **** TESTCASE START
38 ****
39 ****
40
41 Test normal operation in Tachometer Setup
42 [Test OK] Result: (Expected: true == Result: true)
```

```
43 Test nullptr Car in Tachometer CTOR
44 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
45
46 Test Display nullptr in CTOR of Tachometer
47 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
48
49 Test nullptr in CTOR of Tachometer
50 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
51
52 Test Car nullptr in Update of Tachometer
53 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
54
55 Test nullptr in Set Display
56 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
57
58
59 ****
60 ****
61
62 ****
63 **** TESTCASE START ****
64
65 ****
66
67 Test normal operation in RPM_Sensor
68 [Test OK] Result: (Expected: true == Result: true)
69
70 Test invalid RPM Data (aaaa aaaa)
71 [Test OK] Result: (Expected: ERROR RPM sensor could not read data from file
    ↪ . == Result: ERROR RPM sensor could not read data from file.)
72
73 Test invalid RPM Data (-1000)
74 [Test OK] Result: (Expected: ERROR RPM sensor could not read data from file
    ↪ . == Result: ERROR RPM sensor could not read data from file.)
75
76 Test invalid RPM Data (1007ab)
77 [Test OK] Result: (Expected: ERROR RPM sensor could not read data from file
    ↪ . == Result: ERROR RPM sensor could not read data from file.)
78
79 Test invalid RPM Data (10.00)
80 [Test OK] Result: (Expected: ERROR RPM sensor could not read data from file
    ↪ . == Result: ERROR RPM sensor could not read data from file.)
81
82 Test file not found in RPM_Sensor
```

```
83 [Test OK] Result: (Expected: ERROR RPM sensor file was not found == Result:  
84     ↪ ERROR RPM sensor file was not found)  
85  
85 Test empty file in RPM_Sensor  
86 [Test OK] Result: (Expected: ERROR RPM sensor could not read data from file  
87     ↪ . == Result: ERROR RPM sensor could not read data from file.)  
88  
89 *****  
90  
91  
92 *****  
93         TESTCASE START  
94 *****  
95  
96 Test Car CTOR with RPM Nullptr  
97 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)  
98  
99 Test Car CTOR with 0 Wheel Diameter  
100 [Test OK] Result: (Expected: ERROR: Wheel Diameter cannot be 0! == Result:  
101     ↪ ERROR: Wheel Diameter cannot be 0!)  
102  
102 Test Car Attach with nullptr  
103 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)  
104  
105 Test Car Detach with nullptr  
106 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)  
107  
108 Test Car Detach with non attached observer  
109 [Test OK] Result: (Expected: true == Result: true)  
110  
111 Test Car Get Current Speed  
112 [Test OK] Result: (Expected: 18849 == Result: 18849)  
113  
114 Test Exception in TestCase  
115 [Test OK] Result: (Expected: true == Result: true)  
116  
117 Test Exception End of File in Car Process  
118 [Test OK] Result: (Expected: ERROR RPM sensor file has ended, theres no  
119     ↪ more data. == Result: ERROR RPM sensor file has ended, theres no more  
120     ↪ data.)  
121 *****
```

122  
123 TEST OK!!

## 7 Quellcode

### 7.1 Object.h

```
1 //*****\n2 * \file Object.h\n3 * \brief Base Object class for all other classes\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8\n9 #ifndef OBJECT_H\n10 #define OBJECT_H\n11\n12 #include <string>\n13\n14 class Object{\n15 public:\n16\n17     virtual ~Object(){}\n18\n19 protected:\n20\n21     Object(){};\n22\n23};\n24\n25 #endif // OBJECT_H
```

## 7.2 Vehicle.hpp

```
1 //*****
2 * \file Vehicle.hpp
3 * \brief Base class representing a generic vehicle that supports
4 *        display observers following the Observer design pattern.
5 *
6 * The Vehicle class manages a collection of display observers that
7 * implement the IDisplay interface. Observers can be attached or
8 * detached at runtime.
9 *
10 * \author Simon
11 * \date November 2025
12 *****/
13
14 #ifndef VEHICLE_HPP
15 #define VEHICLE_HPP
16
17 #include "Object.h"
18 #include "IDisplay.hpp"
19 #include <vector>
20
21 /**
22 * \class Vehicle
23 * \brief Base class supporting observer management for display updates.
24 */
25 class Vehicle : public Object {
26 public:
27
28     /**
29      * \brief Error message thrown when a null pointer is passed.
30      */
31     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
32
33     /**
34      * \brief Container type alias for display observers.
35      *
36      * Stores shared pointers to IDisplay implementations.
37      */
38     using TCont = std::vector<IDisplay::Sptr>;
39
40     /**
41      * \brief Attaches a display observer to the vehicle.
42      *
43      * \param display Shared pointer to a display object implementing IDisplay.
44      *
45      * \throws std::string ERROR_NULLPTR if display is null.
46      */
47     void Attach(IDisplay::Sptr display);
48
49     /**
50      * \brief Detaches a display observer from the vehicle.
51      *
52      * \param display Shared pointer to a display object that should be removed.
53      *
54      * \throws std::string ERROR_NULLPTR if display is null.
55      */
56     void Detach(IDisplay::Sptr display);
57
58 protected:
59
60     /**
61      * \brief Notifies all attached display observers.
62      *
63      * This method loops through all observers stored in the container and
64      * calls Update() on each non-null display.
65      */
66     void Notify() const;
67
68     /**
69      * \brief Protected default constructor.
70      *
71      * Prevents direct instantiation of Vehicle, as it should be subclassed.
72      */
```

```
73     Vehicle() = default;
74
75 private:
76
77     /**
78      * \brief Container holding all attached display observers.
79      */
80     TCont m_display_observers;
81 };
82
83 #endif // !VEHICLE_HPP
```

## 7.3 Vehicle.cpp

```
1 //*****\n2 * \file Vehicle.cpp\n3 * \brief Base class representing a generic vehicle that supports\n4 *       display observers following the Observer design pattern.\n5 *\n6 * The Vehicle class manages a collection of display observers that\n7 * implement the IDisplay interface. Observers can be attached or\n8 * detached at runtime.\n9 *\n10 * \author Simon\n11 * \date November 2025\n12 */\n13 #include "Vehicle.hpp"\n14 #include "algorithm"\n15\n16 using namespace std;\n17\n18 void Vehicle::Attach(IDisplay::Sptr display)\n19 {\n20     if (display == nullptr) throw Vehicle::ERROR_NULLPTR;\n21\n22     m_display_observers.emplace_back(move(display));\n23 }\n24\n25 void Vehicle::Detach(IDisplay::Sptr display)\n26 {\n27     if (display == nullptr) throw Vehicle::ERROR_NULLPTR;\n28\n29     const auto it = find(m_display_observers.cbegin(), m_display_observers.cend(), display);\n30\n31     if (it != m_display_observers.cend()) {\n32         m_display_observers.erase(it);\n33     }\n34 }\n35\n36 void Vehicle::Notify() const\n37 {\n38     for_each(m_display_observers.cbegin(), m_display_observers.cend(),\n39             [] (auto& obs) { if(obs != nullptr) obs->Update();});\n40 }
```

## 7.4 Car.hpp

```
1 //*****
2 * \file Car.hpp
3 * \brief Declares the Car class, a Vehicle implementation using an RPM
4 * sensor to calculate its current speed.
5 *
6 * \author Simon
7 * \date November 2025
8 *****/
9
10 #ifndef CAR_HPP
11 #define CAR_HPP
12
13 #include <string>
14 #include <string_view>
15 #include <memory>
16
17 #include "RPM_Sensor.hpp"
18 #include "Vehicle.hpp"
19
20 /**
21 * \class Car
22 * \brief Vehicle subclass that calculates speed using an RPM sensor.
23 */
24 class Car : public Vehicle {
25 public:
26
27     /**
28     * \brief Shared pointer alias for Car.
29     */
30     using Sptr = std::shared_ptr<Car>;
31
32     /**
33     * \brief Weak pointer alias for Car.
34     */
35     using Wptr = std::weak_ptr<Car>;
36
37     /**
38     * \brief Error message for invalid wheel diameter.
39     */
40     static inline const std::string ERROR_WHEEL_DIA_0 = "ERROR: Wheel Diameter cannot be 0!";
41
42     /**
43     * \brief Error message for null sensor pointer.
44     */
45     static inline const std::string ERROR_NULLPTR = "ERROR_Nullptr";
46
47     /**
48     * \brief Retrieves RPM data from the sensor and updates internal state.
49     *
50     * This method calls RPM_Sensor::GetRevolutions() to update the current
51     * wheel RPM. If the sensor throws an exception, the internal RPM value
52     * is reset to zero before rethrowing the exception.
53     *
54     * After processing sensor data, this method notifies all observers
55     * through Vehicle::Notify().
56     *
57     * \throws Any exception thrown by the underlying RPM sensor.
58     */
59     void Process();
60
61     /**
62     * \brief Computes the current speed of the car in kilometers per hour.
63     *
64     * The speed is calculated using the wheel RPM, wheel diameter (in mm),
65     * and conversion constants.
66     * \return The current speed in KPH.
67     */
68     double GetCurrentSpeed() const;
69
70     /**
71     * \brief Constructs a Car with a wheel diameter and an RPM sensor.
72     *
```

```
73     * \param wheel_diameter Wheel diameter in millimeters.
74     * \param wh_rpm_sen Shared pointer to an RPM sensor instance.
75     *
76     * \throws std::string ERROR_WHEEL_DIA_0 if wheel_diameter is zero.
77     * \throws std::string ERROR_NULLPTR if wh_rpm_sen is null.
78     */
79 Car(const size_t& wheel_diameter, RPM_Sensor::Sptr wh_rpm_sen);
80
81 private:
82 /**
83  * \brief Last measured wheel revolutions per minute.
84  */
85 size_t m_wheel_rmp;
86
87 /**
88  * \brief Wheel diameter in millimeters.
89  */
90 size_t m_wheel_diameter;
91
92 /**
93  * \brief Pointer to the RPM sensor used for measurement.
94  */
95 RPM_Sensor::Sptr m_wheel_rpm_sensor;
96
97 /**
98  * \brief Conversion factor from meters per second to kilometers per hour.
99  */
100 inline static const double mps_to_kph = 3.6;
101
102 /**
103  * \brief Number of seconds in one minute.
104  */
105 inline static const double seconds_in_min = 60;
106
107 /**
108  * \brief Millimeters in one meter.
109  */
110 inline static const double mm_in_m = 1000;
111 };
112
113 #endif // !CAR_HPP
```

## 7.5 Car.cpp

```
1 //*****\n2 * \file Car.cpp\n3 * \brief Declares the Car class, a Vehicle implementation using an RPM\n4 * sensor to calculate its current speed.\n5 *\n6 * \author Simon\n7 * \date November 2025\n8 *****\n9\n10 #include "Car.hpp"\n11 #include <numbers>\n12\n13 void Car::Process()\n14 {\n15     try {\n16         // Get Revolutions\n17         m_wheel_rmp = m_wheel_rpm_sensor->GetRevolutions();\n18     }\n19     catch (...) {\n20\n21         m_wheel_rmp = 0;\n22\n23         throw; // rethrow exception to inform user of the exception\n24     }\n25\n26     Notify();\n27 }\n28\n29 double Car::GetCurrentSpeed() const\n30 {\n31     return ((m_wheel_rmp / seconds_in_min) * m_wheel_diameter * std::numbers::pi * mps_to_kph) / mm_in_m;\n32 }\n33\n34 Car::Car(const size_t & wheel_diameter, RPM_Sensor::Sptr wh_rpm_sen) : m_wheel_rmp(0)\n35 {\n36     if (wheel_diameter == 0) throw Car::ERROR_WHEEL_DIA_0;\n37     if (wh_rpm_sen == nullptr) throw Car::ERROR_NULLPTR;\n38\n39     m_wheel_rpm_sensor = std::move(wh_rpm_sen);\n40\n41     m_wheel_diameter = wheel_diameter;\n42 }
```

## 7.6 IDisplay.hpp

```
1 /*****  
2 * \file IDisplay.hpp  
3 * \brief Interface for all display types used by vehicles.  
4 * \author Simon  
5 * \date November 2025  
6 *****  
7  
8 #ifndef IDISPLAY_HPP  
9 #define IDISPLAY_HPP  
10  
11 #include <memory>  
12  
13 /**  
14 * \class IDisplay  
15 * \brief Interface for display observer implementations.  
16 */  
17 class IDisplay {  
public:  
18  
    /**  
     * \brief Shared pointer alias for IDisplay.  
     */  
    using Sptr = std::shared_ptr<IDisplay>;  
19  
    /**  
     * \brief Called when the observed subject updates its state.  
     */  
    virtual void Update() = 0;  
20  
    /**  
     * \brief Virtual destructor.  
     *  
     * Ensures proper cleanup of derived display types.  
     */  
    virtual ~IDisplay() = default;  
21};  
22  
23  
#endif // !IDISPLAY_HPP
```

## 7.7 Meter.hpp

```

1 /***** *****/
2 * \file Meter.hpp
3 * \brief Abstract base class for all meter display types.
4 *
5 * \author Simon
6 * \date November 2025
7 *****/
8
9 #ifndef METER_HPP
10 #define METER_HPP
11
12 #include "Object.h"
13 #include "IDisplay.hpp"
14 #include "WindowsDisplay.h"
15
16 /**
17 * \class Meter
18 * \brief Abstract base class for all meter display components.
19 */
20 class Meter : public Object, public IDisplay {
21 public:
22
23     /**
24     * \brief Error message thrown when a null pointer is passed.
25     */
26     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
27
28     /**
29     * \brief Sets the WindowsDisplay instance used by this meter.
30     * \param display Shared pointer to a WindowsDisplay.
31     *
32     * \throws std::string ERROR_NULLPTR if display is null.
33     */
34     void SetDisplay(WindowsDisplay::SPtr display);
35
36 protected:
37
38     /**
39     * \brief Constructs a Meter with an initial WindowsDisplay instance.
40     *
41     * \param display Shared pointer to a WindowsDisplay.
42     *
43     * \throws std::string ERROR_NULLPTR if display is null.
44     */
45     Meter(WindowsDisplay::SPtr display);
46
47     /**
48     * \brief Default protected constructor.
49     *
50     * Allows derived classes to be created without immediately providing
51     * a display. A valid display must later be set through SetDisplay().
52     */
53     Meter() = default;
54
55     /**
56     * \brief Pointer to the display window used by the meter.
57     */
58     WindowsDisplay::SPtr m_window;
59
60 private:
61     // No private members currently required.
62 };
63
64 #endif // !METER_HPP

```

## 7.8 Meter.cpp

```
1 #include "Meter.hpp"
2
3 void Meter::SetDisplay(WindowsDisplay::SPtr display)
4 {
5
6     if (display == nullptr) throw Meter::ERROR_NULLPTR;
7
8     m_window = move(display);
9 }
10
11 Meter::Meter(WindowsDisplay::SPtr display)
12 {
13     if (display == nullptr) throw Meter::ERROR_NULLPTR;
14
15     m_window = move(display);
16 }
```

## 7.9 Tachometer.hpp

```
1 //*****
2 * \file Tachometer.hpp
3 * \brief Display that shows the current speed of a car.
4 *
5 * \The Tachometer observes a Car instance and displays its current speed.
6 * It derives from Meter and serves as an observer in the Vehicle system.
7 *
8 * \author Simon
9 * \date November 2025
10 *****/
11
12 #ifndef TACHOMETER_HPP
13 #define TACHOMETER_HPP
14
15 #include "Object.h"
16 #include "Meter.hpp"
17 #include "Car.hpp"
18
19 /**
20 * \class Tachometer
21 * \brief Meter implementation that displays the real-time speed of a vehicle.
22 */
23 class Tachometer : public Meter {
24 public:
25
26     /**
27     * \brief Error message thrown when a null pointer is passed.
28     */
29     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
30
31     /**
32     * \brief Shared pointer alias for Tachometer.
33     */
34     using Sptr = std::shared_ptr<Tachometer>;
35
36     /**
37     * \brief Updates the tachometer display with the current speed.
38     *
39     * \throws std::string ERROR_NULLPTR if the car pointer is expired.
40     */
41     virtual void Update() override;
42
43     /**
44     * \brief Constructs a Tachometer with both a Car and a display.
45     *
46     * \param car Shared pointer to the observed Car.
47     * \param display Shared pointer to a WindowsDisplay instance.
48     *
49     * \throws std::string ERROR_NULLPTR if car or display is null.
50     */
51     Tachometer(Car::Sptr car, WindowsDisplay::SPtr display);
52
53     /**
54     * \brief Constructs a Tachometer with a Car but without a display.
55     *
56     * A valid display can be attached later using SetDisplay(), inherited
57     * from Meter. The Car pointer must not be null and is stored as a
58     * weak pointer.
59     *
60     * \param car Shared pointer to the observed Car.
61     *
62     * \throws std::string ERROR_NULLPTR if car is null.
63     */
64     Tachometer(Car::Sptr car);
65
66 private:
67
68     /**
69     * \brief Weak pointer to the observed Car.
70     *
71     * Prevents ownership cycles between Car and Tachometer. If the Car
```

```
73     * instance is destroyed, Update() will detect this via lock().
74     */
75     Car::Wptr m_car;
76
77     /**
78     * \brief Last measured speed in kilometers per hour.
79     *
80     * This value is updated during each call to Update() and forwarded
81     * to the WindowsDisplay using SendValue().
82     */
83     double m_speed;
84 };
85
86 #endif // !TACHOMETER_HPP
```

## 7.10 Tachometer.cpp

```
1 //*****\n2 * \file Tachometer.cpp\n3 * \brief Display that shows the current speed of a car.\n4 *\n5 * The Tachometer observes a Car instance and displays its current speed.\n6 * It derives from Meter and serves as an observer in the Vehicle system.\n7 *\n8 * \author Simon\n9 * \date November 2025\n10 *****\n11\n12 #include "Tachometer.hpp"\n13\n14 #include <iostream>\n15\n16 void Tachometer::Update()\n17 {\n18     Car::Sptr car = m_car.lock();\n19\n20     // check if sptr is valid\n21     if (car == nullptr) throw Tachometer::ERROR_NULLPTR;\n22\n23     m_speed = car->GetCurrentSpeed();\n24\n25     if (m_window != nullptr) m_window->SendValue(static_cast<unsigned int>(m_speed));\n26 }\n27\n28 Tachometer::Tachometer(Car::Sptr car, WindowsDisplay::SPtr display) : m_speed{0}, Meter{move(display)}\n29 {\n30     if (car == nullptr) throw Tachometer::ERROR_NULLPTR;\n31\n32     m_car = move(car);\n33 }\n34\n35\n36 Tachometer::Tachometer(Car::Sptr car) : m_speed{ 0 }\n37 {\n38     if (car == nullptr) throw Tachometer::ERROR_NULLPTR;\n39\n40     m_car = move(car);\n41 }
```

## 7.11 Odometer.hpp

```

1 /*****//**
2 * \file Odometer.hpp
3 * \brief Display that calculates and shows the current mileage of a car.
4 *
5 * \author Simon
6 * \date November 2025
7 *****/
8
9 #ifndef ODOMETER_HPP
10 #define ODOMETER_HPP
11
12 #include "Object.h"
13 #include "Meter.hpp"
14 #include "Car.hpp"
15
16 /**
17 * \class Odometer
18 * \brief Meter implementation that calculates and displays vehicle mileage
19 *
20 * The Odometer observes a Car instance and increases its internal mileage
21 * value based on the car's current speed and a fixed update interval.
22 * Mileage is stored as a double for accumulation precision, but exposed
23 * as a size_t when queried.
24 *
25 * The car pointer is stored as a weak pointer to avoid ownership cycles.
26 * If the car expires, Update() throws an exception.
27 */
28 class Odometer : public Meter {
29 public:
30
31 /**
32 * \brief Error message thrown when a null pointer is passed.
33 */
34 inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
35
36 /**
37 * \brief Update interval in milliseconds.
38 *
39 * This value determines how frequently the odometer expects to be
40 * updated. It is used in the mileage calculation:
41 *   mileage += speed_kph * (interval_ms / milliseconds_per_hour)
42 */
43 inline static const size_t Update_Interval = 500;
44
45 /**
46 * \brief Shared pointer alias for Odometer.
47 */
48 using Sptr = std::shared_ptr<Odometer>;
49
50 /**
51 * \brief Updates the mileage and forwards the new value to the display.
52 *
53 * \throws std::string ERROR_NULLPTR if the car pointer is expired.
54 */
55 virtual void Update() override;
56
57 /**
58 * \brief Returns the accumulated mileage.
59 *
60 * \return Mileage as a size_t value.
61 */
62 size_t GetMileage() const;
63
64 /**
65 * \brief Constructs an Odometer with a Car and a display.
66 *
67 * \param car Shared pointer to the observed Car.
68 * \param display Shared pointer to a WindowsDisplay instance.
69 *
70 * \throws std::string ERROR_NULLPTR if either pointer is null.
71 */
72 Odometer(Car::Sptr car, WindowsDisplay::SPtr display);

```

```
73     /**
74      * \brief Constructs an Odometer with a Car and without a display.
75      *
76      * The display must be set later using SetDisplay() before visual output
77      * is possible.
78      *
79      * \param car Shared pointer to the Car being observed.
80      *
81      * \throws std::string ERROR_NULLPTR if car is null.
82      */
83     Odometer(Car::Sptr car);
84
85 private:
86
87     /**
88      * \brief Weak pointer to the observed Car.
89      *
90      * Ensures there is no ownership cycle between Car and Odometer.
91      */
92     Car::Wptr m_car;
93
94     /**
95      * \brief Accumulated mileage in kilometers (stored as double).
96      *
97      * Internally stored as double for precision. Exposed as size_t via
98      * GetMilage() for external usage.
99      */
100    double m_milage;
101
102    /**
103     * \brief Number of milliseconds in one hour.
104     *
105     * Used for converting speed (KPH) into incremental distance
106     * based on the update interval.
107     */
108    static const size_t msecounds_in_hours = 3600000;
109 };
110
111 #endif // !ODOMETER_HPP
```

## 7.12 Odometer.cpp

```
1 //*****\n2 * \file Odometer.cpp\n3 * \brief Display that calculates and shows the current mileage of a car.\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8 #include "Odometer.hpp"\n9\n10 void Odometer::Update()\n11 {\n12     Car::Sptr car = m_car.lock();\n13\n14     if (car == nullptr) throw Odometer::ERROR_NULLPTR;\n15\n16     // get the absolute value of the Current Speed\n17     // negative Speed does not make the milage go down!\n18     const double speed = abs(car->GetCurrentSpeed());\n19\n20     // Integrate the speed over the update interval to get the distance\n21     m_milage = m_milage + (speed * Odometer::Update_Interval) / mseconds_in_hours;\n22\n23     // If a display is set, send the new milage value\n24     if(m_window != nullptr) m_window->SendValue(static_cast<unsigned int>(m_milage));\n25 }\n26\n27\n28 Odometer::Odometer(Car::Sptr car, WindowsDisplay::SPtr display) : m_milage{ 0 }, Meter{ move(display) }\n29 {\n30     if (car == nullptr) throw Odometer::ERROR_NULLPTR;\n31\n32     m_car = move(car);\n33 }\n34\n35 Odometer::Odometer(Car::Sptr car) : m_milage{ 0 }\n36 {\n37     if (car == nullptr) throw Odometer::ERROR_NULLPTR;\n38\n39     m_car = move(car);\n40 }\n41\n42\n43 size_t Odometer::GetMilage() const\n44 {\n45     return static_cast<size_t>(m_milage);\n46 }
```

## 7.13 RPM\_Sensor.hpp

```
1 //*****\n2 * \file RPM_Sensor.hpp\n3 * \brief A "sensor" for returning individual readings when\n4 * GetRevoltioins() is called.\n5 *\n6 * \author Simon\n7 * \date November 2025\n8 *****\n9\n10 #ifndef RPM_SENSOR_HPP\n11 #define RPM_SENSOR_HPP\n12\n13 #include "Object.h"\n14 #include <string>\n15 #include <string_view>\n16 #include <memory>\n17 #include <ifstream>\n18\n19 class RPM_Sensor : public Object {\n20 public:\n21     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";\n22     inline static const std::string ERROR_SENSOR_FILE_NOT_FOUND = "ERROR_RPM_sensor_file_was_not_found";\n23     inline static const std::string ERROR_SENSOR_INVALID_DATA_INPUT = "ERROR_RPM_sensor_could_not_read_data_from_file.";\n24     inline static const std::string ERROR_SENSOR_EOF = "ERROR_RPM_sensor_file_has_ended,_theres_no_more_data.";\n25\n26     /**\n27     * \brief Shared pointer type for RPM_Sensor\n28     */\n29     using Sptr = std::shared_ptr<RPM_Sensor>;\n30\n31     /**\n32     * \brief Returns current rpm. This is achieved by parsing\n33     * from a testfile - if the end of the file is reached a\n34     * exception is thrown (ERROR_SENSOR_EOF). This has to\n35     * be handled by the user of this class.\n36     * \return unsigned int revs\n37     */\n38     size_t GetRevolutions();\n39\n40     /**\n41     * \brief RPM_Sensor constructor\n42     * throws error (ERROR_SENSOR_FILE_NOT_FOUND)\n43     * if the provided file/path is invalid.\n44     */\n45     RPM_Sensor(std::string_view testFileName);\n46\n47     /**\n48     * \brief custom destructor is needed to\n49     * close ifstream.\n50     */\n51     ~RPM_Sensor();\n52\n53     // delete CopyCtor and Assign Operator because we warp a file stream\n54     // which should not be copied or assigned!\n55     RPM_Sensor(RPM_Sensor& s) = delete;\n56     void operator=(RPM_Sensor s) = delete;\n57\n58 private:\n59     // open a filestream when sensor is constructed\n60     // close when destructor is called.\n61     std::ifstream m_fileStream;\n62 },\n63\n64 #endif // !RPM_SENSOR_HPP
```

## 7.14 RPM\_Sensor.cpp

```
1 //*****\n2 * \file    RPM_Sensor.cpp\n3 * \brief   A "sensor" for returning individual readings when\n4 * GetRevolutioins() is called.\n5 *\n6 * \author Simon\n7 * \date   November 2025\n8 *****\n9\n10 #include "RPM_Sensor.hpp"\n11\n12 #include <algorithm>\n13 #include <sstream>\n14\n15 size_t RPM_Sensor::GetRevolutions()\n16 {\n17     std::string sensor_reading;\n18     stringstream converter;\n19     size_t sensor_value = 0;\n20\n21     if (m_fileStream.eof())\n22         throw ERROR_SENSOR_EOF;\n23\n24     if(m_fileStream.fail())\n25         throw ERROR_SENSOR_INVALID_DATA_INPUT;\n26\n27     m_fileStream >> sensor_reading;\n28\n29     if (sensor_reading.empty())\n30         throw ERROR_SENSOR_INVALID_DATA_INPUT;\n31\n32     // check if all of the readings are digits\n33     if (!std::all_of(sensor_reading.cbegin(), sensor_reading.cend(), ::isdigit))\n34         throw ERROR_SENSOR_INVALID_DATA_INPUT;\n35\n36     // use Stringstream for type Conversion\n37     converter << sensor_reading;\n38     converter >> sensor_value;\n39\n40     return sensor_value;\n41 }\n42\n43 RPM_Sensor::RPM_Sensor(std::string_view testFileName)\n44 {\n45     m_fileStream = std::ifstream(testFileName.data());\n46\n47     if (!m_fileStream.is_open())\n48         throw ERROR_SENSOR_FILE_NOT_FOUND;\n49 }\n50\n51 RPM_Sensor::~RPM_Sensor()\n52 {\n53     m_fileStream.close();\n54 }
```

## 7.15 main.cpp

```
1 //*****\n2 * \file main.cpp\n3 * \brief Test Driver for the Drive Sim\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8\n9 //zur Verwendung der analogen und digitalen Anzeige\n10 #include "WindowsDisplay.h"\n11\n12 #include "RPM_Sensor.hpp"\n13 #include "Car.hpp"\n14 #include "Tachometer.hpp"\n15 #include "Odometer.hpp"\n16\n17 #include "Test.hpp"\n18 #include "vld.h"\n19 #include <fstream>\n20 #include <iostream>\n21 #include <cassert>\n22\n23 #define WriteOutputFile ON\n24\n25 using namespace std;\n26\n27 static bool TestOdometer(std::ostream & ost = std::cout);\n28 static bool TestTachometer(std::ostream & ost = std::cout);\n29 static bool TestRPMSensor(std::ostream& ost = std::cout);\n30 static bool TestCar(std::ostream& ost = std::cout);\n31\n32 int main()\n33 {\n34     bool TestOK = true;\n35\n36     ofstream output( "output.txt" );\n37\n38     try {\n39\n40         TestOK = TestOK && TestOdometer();\n41         TestOK = TestOK && TestTachometer();\n42         TestOK = TestOK && TestRPMSensor();\n43         TestOK = TestOK && TestCar();\n44\n45\n46     if (WriteOutputFile) {\n47\n48         TestOK = TestOK && TestOdometer(output);\n49         TestOK = TestOK && TestTachometer(output);\n50         TestOK = TestOK && TestRPMSensor(output);\n51         TestOK = TestOK && TestCar(output);\n52\n53         if (TestOK) {\n54             output << TestCaseOK;\n55         }\n56         else {\n57             output << TestCaseFail;\n58         }\n59\n60         output.close();\n61     }\n62\n63     if (TestOK) {\n64         cout << TestCaseOK;\n65     }\n66     else {\n67         cout << TestCaseFail;\n68     }\n69 }\n70 catch (const string& err) {\n71     cerr << err << TestCaseFail;\n72 }
```

```
73     catch (bad_alloc const& error) {
74         cerr << error.what() << TestCaseFail;
75     }
76     catch (const exception& err) {
77         cerr << err.what() << TestCaseFail;
78     }
79     catch (...) {
80         cerr << "Unhandelt_Exception" << TestCaseFail;
81     }
82
83     if (output.is_open()) output.close();
84
85     try{
86
87         //Erzeugen der Objekte
88         WindowsDisplay::SPtr digDisp = make_shared<DigitalDisplay>();
89         WindowsDisplay::SPtr anaDisp = make_shared<AnalogDisplay>();
90
91         RPM_Sensor::SPtr rpm_sens = make_shared<RPM_Sensor>("rpm_data.txt");
92         Car::SPtr TestCar = make_shared<Car>( 600, rpm_sens );
93         Tachometer::SPtr tacho = make_shared<Tachometer>(TestCar,anaDisp);
94         Odometer::SPtr odo = make_shared<Odometer>(TestCar, digDisp);
95         TestCar->Attach(tacho);
96         TestCar->Attach(odo);
97
98         //send values to displays
99         while(1){
100
101             TestCar->Process();
102             Sleep(Odometer::Update_Interval);
103         }
104     }
105     catch (const string& err) {
106         cerr << err;
107     }
108     catch (bad_alloc const& error) {
109         cerr << error.what();
110     }
111     catch (const exception& err) {
112         cerr << err.what();
113     }
114     catch (...) {
115         cerr << "Unhandelt_Exception";
116     }
117
118
119     return 0;
120 }
121
122 bool TestOdometer(std::ostream& ost)
123 {
124     assert(ost.good());
125
126     ost << TestStart;
127
128     bool TestOK = true;
129     string error_msg;
130
131     try {
132         RPM_Sensor::SPtr sen = make_shared<RPM_Sensor>("rpm_data.txt");
133         Car::SPtr AudiA3 = make_shared<Car>(600, sen);
134         Odometer::SPtr OdoMeter = make_shared<Odometer>(AudiA3);
135         AudiA3->Attach(OdoMeter);
136     }
137     catch (const string& err) {
138         error_msg = err;
139     }
140     catch (bad_alloc const& error) {
141         error_msg = error.what();
142     }
143     catch (const exception& err) {
144         error_msg = err.what();
145     }
146     catch (...) {
147         error_msg = "Unhandelt_Exception";
```

```
148     }
149 
150     TestOK = TestOK && check_dump(ost, "Test_normal_operation_in_Odometer_Setup", true, error_msg.empty());
151     error_msg.clear();
152 
153     try {
154         Odometer::Sptr OdoMeter = make_shared<Odometer>(nullptr);
155     }
156     catch (const string& err) {
157         error_msg = err;
158     }
159     catch (bad_alloc const& error) {
160         error_msg = error.what();
161     }
162     catch (const exception& err) {
163         error_msg = err.what();
164     }
165     catch (...) {
166         error_msg = "Unhandelt_Exception";
167     }
168 
169     TestOK = TestOK && check_dump(ost, "Test_nullptr_Car_in_OdometerCTOR", Odometer::ERROR_NULLPTR, error_msg);
170     error_msg.clear();
171 
172     try {
173         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
174         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
175         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3, nullptr);
176     }
177     catch (const string& err) {
178         error_msg = err;
179     }
180     catch (bad_alloc const& error) {
181         error_msg = error.what();
182     }
183     catch (const exception& err) {
184         error_msg = err.what();
185     }
186     catch (...) {
187         error_msg = "Unhandelt_Exception";
188     }
189 
190     TestOK = TestOK && check_dump(ost, "Test_Display_nullptr_inCTOR_of_Odometer", Odometer::ERROR_NULLPTR, error_msg);
191     error_msg.clear();
192 
193     try {
194         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
195         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
196         Odometer::Sptr OdoMeter = make_shared<Odometer>(nullptr, nullptr);
197     }
198     catch (const string& err) {
199         error_msg = err;
200     }
201     catch (bad_alloc const& error) {
202         error_msg = error.what();
203     }
204     catch (const exception& err) {
205         error_msg = err.what();
206     }
207     catch (...) {
208         error_msg = "Unhandelt_Exception";
209     }
210 
211     TestOK = TestOK && check_dump(ost, "Test_nullptr_inCTOR_of_Odometer", Odometer::ERROR_NULLPTR, error_msg);
212     error_msg.clear();
213 
214     try {
215         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
216         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
217         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
218         AudiA3.reset(); // <-- Free Car
219         OdoMeter->Update(); // <-- throws exception Car not set
220     }
221     catch (const string& err) {
222         error_msg = err;
```

```
223     }
224     catch (bad_alloc const& error) {
225         error_msg = error.what();
226     }
227     catch (const exception& err) {
228         error_msg = err.what();
229     }
230     catch (...) {
231         error_msg = "UnhandeltedException";
232     }
233
234     TestOK = TestOK && check_dump(ost, "Test_Car_nullptr_in_Update_of_Odometer", Odometer::ERROR_NULLPTR, error_msg);
235     error_msg.clear();
236
237     try {
238         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
239         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
240         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
241
242         OdoMeter->SetDisplay(nullptr);
243     }
244     catch (const string& err) {
245         error_msg = err;
246     }
247     catch (bad_alloc const& error) {
248         error_msg = error.what();
249     }
250     catch (const exception& err) {
251         error_msg = err.what();
252     }
253     catch (...) {
254         error_msg = "UnhandeltedException";
255     }
256
257     TestOK = TestOK && check_dump(ost, "Test_nullptr_in_Set_Display", Odometer::ERROR_NULLPTR, error_msg);
258     error_msg.clear();
259
260     try {
261         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
262         Car::Sptr AudiA3 = make_shared<Car>(1000, sen);
263         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
264         AudiA3->Attach(OdoMeter);
265
266         TestOK = TestOK && check_dump(ost, "Test_initial_Milage_of_Odometer", static_cast<size_t>(0), OdoMeter->GetMilage());
267
268         AudiA3->Process();
269
270         TestOK = TestOK && check_dump(ost, "Test_Milage_after_one_Process_of_Odometer", static_cast<size_t>(26), OdoMeter->GetMilage());
271
272     }
273     catch (const string& err) {
274         error_msg = err;
275     }
276     catch (bad_alloc const& error) {
277         error_msg = error.what();
278     }
279     catch (const exception& err) {
280         error_msg = err.what();
281     }
282     catch (...) {
283         error_msg = "UnhandeltedException";
284     }
285
286     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
287     error_msg.clear();
288
289     ost << TestEnd;
290
291     return TestOK;
292 }
293
294 bool TestTachometer(std::ostream& ost)
295 {
296     assert(ost.good());
```

```
298     ost << TestStart;
299
300
301
302     bool TestOK = true;
303     string error_msg;
304
305     try {
306         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
307         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
308         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3);
309         AudiA3->Attach(TachMeter);
310     }
311     catch (const string& err) {
312         error_msg = err;
313     }
314     catch (bad_alloc const& error) {
315         error_msg = error.what();
316     }
317     catch (const exception& err) {
318         error_msg = err.what();
319     }
320     catch (...) {
321         error_msg = "Unhandelt_Exception";
322     }
323
324     TestOK = TestOK && check_dump(ost, "Test_normal_operation_in_Tachometer_Setup", true, error_msg.empty());
325     error_msg.clear();
326
327     try {
328         Tachometer::Sptr TachMeter = make_shared<Tachometer>(nullptr);
329     }
330     catch (const string& err) {
331         error_msg = err;
332     }
333     catch (bad_alloc const& error) {
334         error_msg = error.what();
335     }
336     catch (const exception& err) {
337         error_msg = err.what();
338     }
339     catch (...) {
340         error_msg = "Unhandelt_Exception";
341     }
342
343     TestOK = TestOK && check_dump(ost, "Test_Nullptr_Car_in_TachometerCTOR", Tachometer::ERROR_NULLPTR, error_msg);
344     error_msg.clear();
345
346     try {
347         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
348         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
349         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3, nullptr);
350     }
351     catch (const string& err) {
352         error_msg = err;
353     }
354     catch (bad_alloc const& error) {
355         error_msg = error.what();
356     }
357     catch (const exception& err) {
358         error_msg = err.what();
359     }
360     catch (...) {
361         error_msg = "Unhandelt_Exception";
362     }
363
364     TestOK = TestOK && check_dump(ost, "Test_Display_Nullptr_inCTOR_of_Tachometer", Tachometer::ERROR_NULLPTR, error_msg);
365     error_msg.clear();
366     try {
367         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
368         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
369         Tachometer::Sptr TachMeter = make_shared<Tachometer>(nullptr, nullptr);
370     }
371     catch (const string& err) {
372         error_msg = err;
```

```
373     }
374     catch (bad_alloc const& error) {
375         error_msg = error.what();
376     }
377     catch (const exception& err) {
378         error_msg = err.what();
379     }
380     catch (...) {
381         error_msg = "Unhandelt_Exception";
382     }
383
384     TestOK = TestOK && check_dump(ost, "Test_nullptr_inCTOR_of_Tachometer", Tachometer::ERROR_NULLPTR, error_msg);
385     error_msg.clear();
386
387     try {
388         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
389         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
390         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3);
391         AudiA3.reset(); // <-- Free Car
392         TachMeter->Update(); // <-- throws exception Car not set
393     }
394     catch (const string& err) {
395         error_msg = err;
396     }
397     catch (bad_alloc const& error) {
398         error_msg = error.what();
399     }
400     catch (const exception& err) {
401         error_msg = err.what();
402     }
403     catch (...) {
404         error_msg = "Unhandelt_Exception";
405     }
406
407     TestOK = TestOK && check_dump(ost, "Test_Car_nullptr_in_Update_of_Tachometer", Tachometer::ERROR_NULLPTR, error_msg);
408     error_msg.clear();
409
410     try {
411         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
412         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
413         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3);
414
415         TachMeter->SetDisplay(nullptr);
416     }
417     catch (const string& err) {
418         error_msg = err;
419     }
420     catch (bad_alloc const& error) {
421         error_msg = error.what();
422     }
423     catch (const exception& err) {
424         error_msg = err.what();
425     }
426     catch (...) {
427         error_msg = "Unhandelt_Exception";
428     }
429
430     TestOK = TestOK && check_dump(ost, "Test_nullptr_in_Set_Display", Tachometer::ERROR_NULLPTR, error_msg);
431     error_msg.clear();
432
433     ost << TestEnd;
434
435     return TestOK;
436 }
437
438 bool TestRPMsensor(std::ostream& ost)
439 {
440     assert(ost.good());
441     ost << TestStart;
442
443     bool TestOK = true;
444     string error_msg;
445
446     // test normal operation
```

```
448     try {
449         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
450         size_t revs = sen->GetRevolutions();
451     }
452     catch (const string& err) {
453         error_msg = err;
454     }
455     catch (bad_alloc const& error) {
456         error_msg = error.what();
457     }
458     catch (const exception& err) {
459         error_msg = err.what();
460     }
461     catch (...) {
462         error_msg = "UnhandeltedException";
463     }
464
465     // check if exception was thrown
466     TestOK = TestOK && check_dump(ost, "Test_normal_operation_in_RPM_Sensor", true, error_msg.empty());
467     error_msg.clear();
468
469     // test invalid Data
470     try {
471         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid1.txt");
472         size_t revs = sen->GetRevolutions();
473     }
474     catch (const string& err) {
475         error_msg = err;
476     }
477     catch (bad_alloc const& error) {
478         error_msg = error.what();
479     }
480     catch (const exception& err) {
481         error_msg = err.what();
482     }
483     catch (...) {
484         error_msg = "UnhandeltedException";
485     }
486
487     // check if exception was thrown
488     TestOK = TestOK && check_dump(ost, "Test_invalid_RPM_Data_(aaaa_aaaa)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
489     error_msg.clear();
490
491     try {
492         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid2.txt");
493         size_t revs = sen->GetRevolutions();
494     }
495     catch (const string& err) {
496         error_msg = err;
497     }
498     catch (bad_alloc const& error) {
499         error_msg = error.what();
500     }
501     catch (const exception& err) {
502         error_msg = err.what();
503     }
504     catch (...) {
505         error_msg = "UnhandeltedException";
506     }
507
508     // check if exception was thrown
509     TestOK = TestOK && check_dump(ost, "Test_invalid_RPM_Data_(-1000)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
510     error_msg.clear();
511
512     try {
513         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid3.txt");
514         size_t revs = sen->GetRevolutions();
515     }
516     catch (const string& err) {
517         error_msg = err;
518     }
519     catch (bad_alloc const& error) {
520         error_msg = error.what();
521     }
522 }
```

```
523     catch (const exception& err) {
524         error_msg = err.what();
525     }
526     catch (...) {
527         error_msg = "Unhandelt_Exception";
528     }
529
530     // check if exception was thrown
531     TestOK = TestOK && check_dump(ost, "Test_invalid_RPM_Data_(1007ab)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
532     error_msg.clear();
533
534
535     try {
536         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid4.txt");
537         size_t revs = sen->GetRevolutions();
538     }
539     catch (const string& err) {
540         error_msg = err;
541     }
542     catch (bad_alloc const& error) {
543         error_msg = error.what();
544     }
545     catch (const exception& err) {
546         error_msg = err.what();
547     }
548     catch (...) {
549         error_msg = "Unhandelt_Exception";
550     }
551
552     // check if exception was thrown
553     TestOK = TestOK && check_dump(ost, "Test_invalid_RPM_Data_(10.00)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
554     error_msg.clear();
555
556     // test file not found
557     try {
558         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("file_not_found.txt");
559         size_t revs = sen->GetRevolutions();
560     }
561     catch (const string& err) {
562         error_msg = err;
563     }
564     catch (bad_alloc const& error) {
565         error_msg = error.what();
566     }
567     catch (const exception& err) {
568         error_msg = err.what();
569     }
570     catch (...) {
571         error_msg = "Unhandelt_Exception";
572     }
573
574     TestOK = TestOK && check_dump(ost, "Test_file_not_found_in_RPM_Sensor", RPM_Sensor::ERROR_SENSOR_FILE_NOT_FOUND, error_msg);
575     error_msg.clear();
576
577     // check empty file
578     try {
579         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_empty.txt");
580         size_t revs = sen->GetRevolutions();
581     }
582     catch (const string& err) {
583         error_msg = err;
584     }
585     catch (bad_alloc const& error) {
586         error_msg = error.what();
587     }
588     catch (const exception& err) {
589         error_msg = err.what();
590     }
591     catch (...) {
592         error_msg = "Unhandelt_Exception";
593     }
594     TestOK = TestOK && check_dump(ost, "Test_empty_file_in_RPM_Sensor", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
595     error_msg.clear();
596     ost << TestEnd;
597 }
```

```
598     return TestOK;
599 }
600
601 bool TestCar(std::ostream& ost)
602 {
603     assert(ost.good());
604     ost << TestStart;
605
606     bool TestOK = true;
607     string error_msg;
608
609     try {
610         Car c{ 100,nullptr };
611     }
612     catch (const string& err) {
613         error_msg = err;
614     }
615     catch (bad_alloc const& error) {
616         error_msg = error.what();
617     }
618     catch (const exception& err) {
619         error_msg = err.what();
620     }
621     catch (...) {
622         error_msg = "UnhandeltedException";
623     }
624
625     // check if exception was thrown
626     TestOK = TestOK && check_dump(ost, "Test_CarCTOR_with_RPM_Nullptr", Car::ERROR_NULLPTR,error_msg);
627     error_msg.clear();
628
629     try {
630         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
631
632         Car c{ 0 ,sen };
633     }
634     catch (const string& err) {
635         error_msg = err;
636     }
637     catch (bad_alloc const& error) {
638         error_msg = error.what();
639     }
640     catch (const exception& err) {
641         error_msg = err.what();
642     }
643     catch (...) {
644         error_msg = "UnhandeltedException";
645     }
646
647     // check if exception was thrown
648     TestOK = TestOK && check_dump(ost, "Test_CarCTOR_with_0_Wheel_Diameter", Car::ERROR_WHEEL_DIA_0,error_msg);
649     error_msg.clear();
650
651     try {
652         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
653         Car c{ 100, sen };
654         c.Attach(nullptr);
655     }
656     catch (const string& err) {
657         error_msg = err;
658     }
659     catch (bad_alloc const& error) {
660         error_msg = error.what();
661     }
662     catch (const exception& err) {
663         error_msg = err.what();
664     }
665     catch (...) {
666         error_msg = "UnhandeltedException";
667     }
668
669     TestOK = TestOK && check_dump(ost, "Test_CarAttach_with_nullptr", Car::ERROR_NULLPTR, error_msg);
670     error_msg.clear();
671
672     try {
```

```
673     RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
674     Car c{ 100, sen };
675     c.Detach(nullptr);
676 }
677 catch (const string& err) {
678     error_msg = err;
679 }
680 catch (bad_alloc const& error) {
681     error_msg = error.what();
682 }
683 catch (const exception& err) {
684     error_msg = err.what();
685 }
686 catch (...) {
687     error_msg = "UnhandeltedException";
688 }
689
690 TestOK = TestOK && check_dump(ost, "Test_Car_Detach_with_nullptr", Car::ERROR_NULLPTR, error_msg);
691 error_msg.clear();
692
693
694 try {
695     RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
696     Car::Sptr c = make_shared<Car>(100, sen );
697     Odometer::Sptr odmeter = make_shared<Odometer>(c);
698     c->Detach(odmeter);
699 }
700 catch (const string& err) {
701     error_msg = err;
702 }
703 catch (bad_alloc const& error) {
704     error_msg = error.what();
705 }
706 catch (const exception& err) {
707     error_msg = err.what();
708 }
709 catch (...) {
710     error_msg = "UnhandeltedException";
711 }
712
713 TestOK = TestOK && check_dump(ost, "Test_Car_Detach_with_non_attached_observer", true, error_msg.empty());
714 error_msg.clear();
715
716
717 try {
718     RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
719     Car::Sptr c = make_shared<Car>(100, sen );
720     c->Process();
721     TestOK = TestOK && check_dump(ost, "Test_Car_Get_Current_Speed", static_cast<size_t>(18849), static_cast<size_t>(c->GetCurrentSpeed));
722 }
723 catch (const string& err) {
724     error_msg = err;
725 }
726 catch (bad_alloc const& error) {
727     error_msg = error.what();
728 }
729 catch (const exception& err) {
730     error_msg = err.what();
731 }
732 catch (...) {
733     error_msg = "UnhandeltedException";
734 }
735
736 TestOK = TestOK && check_dump(ost, "Test_Exception_in_TestCase", true, error_msg.empty());
737 error_msg.clear();
738
739
740 try {
741     RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
742     Car::Sptr c = make_shared<Car>(100, sen );
743     while(1) c->Process();
744 }
745 catch (const string& err) {
746     error_msg = err;
747 }
748 catch (bad_alloc const& error) {
```

```
748     error_msg = error.what();
749 }
750 catch (const exception& err) {
751     error_msg = err.what();
752 }
753 catch (...) {
754     error_msg = "UnhandeltedException";
755 }
756
757 TestOK = TestOK && check_dump(ost, "Test_Exception_End_of_File_in_Car_Process", RPM_Sensor::ERROR_SENSOR_EOF, error_msg);
758 error_msg.clear();
759
760 ost << TestEnd;
761
762 return TestOK;
763 }
```

## 7.16 Test.hpp

```

1 /***** // *****
2 * \file Test.hpp
3 * \brief File that provides a Test Function with a formated output
4 *
5 * \author Simon
6 * \date April 2025
7 *****/
8 #ifndef TEST_HPP
9 #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream.
23 const std::string colorRed = "\x1B[31m";
24 const std::string colorGreen = "\x1B[32m";
25 const std::string colorWhite = "\x1B[37m";
26
27 inline std::ostream& RED(std::ostream& ost) {
28     if (ost.good()) {
29         ost << colorRed;
30     }
31     return ost;
32 }
33 inline std::ostream& GREEN(std::ostream& ost) {
34     if (ost.good()) {
35         ost << colorGreen;
36     }
37     return ost;
38 }
39 inline std::ostream& WHITE(std::ostream& ost) {
40     if (ost.good()) {
41         ost << colorWhite;
42     }
43     return ost;
44 }
45
46 inline std::ostream& TestStart(std::ostream& ost) {
47     if (ost.good()) {
48         ost << std::endl;
49         ost << "*****" << std::endl;
50         ost << "_____TESTCASE_START_____" << std::endl;
51         ost << "*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67
68 #if COLOR_OUTPUT
69     if (ost.good()) {
70         ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;
71     }
72 #else

```

```
73     if (ost.good()) {
74         ost << "TEST_OK!!" << std::endl;
75     }
76 #endif // COLOR_OUTPUT
77
78     return ost;
79 }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed << "TEST_FAILED!!" << colorWhite << std::endl;
86     }
87 #else
88     if (ost.good()) {
89         ost << "TEST_FAILED!!" << std::endl;
90     }
91 #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97 * \brief function that reports if the testcase was successful.
98 *
99 * \param testcase String that indicates the testcase
100 * \param successful true -> reports to cout test OK
101 * \param successful false -> reports test failed
102 */
103 template <typename T>
104 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
105     if (ostr.good()) {
106 #if COLOR_OUTPUT
107         if (expected == result) {
108             ostr << testcase << std::endl << colorGreen << "[Test_OK]" << colorWhite << "Result:" << std::boolalpha << expected
109             << std::noboolalpha << std::endl << std::endl;
110         }
111         else {
112             ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite << "Result:" << std::boolalpha << expected
113             << std::noboolalpha << std::endl << std::endl;
114         }
115 #else
116         if (expected == result) {
117             ostr << testcase << std::endl << "[Test_OK]" << "Result:" << std::boolalpha << expected << "==" << "Result:" <<
118         }
119         else {
120             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:" << std::boolalpha << expected << "!=" << "Result:" <<
121         }
122 #endif
123         if (ostr.fail()) {
124             std::cerr << "Error: Write_Ostream" << std::endl;
125         }
126     }
127     else {
128         std::cerr << "Error: Bad_Ostream" << std::endl;
129     }
130     return expected == result;
131 }
132
133 template <typename T1, typename T2>
134 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
135     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
136     ost << "(" << p.first << "," << p.second << ")";
137     return ost;
138 }
139
140 template <typename T>
141 std::ostream& operator<< (std::ostream& ost, const std::vector<T> & cont) {
142     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
143     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
144     return ost;
145 }
```

```
146
147 template <typename T>
148 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
149     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
150     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
151     return ost;
152 }
153
154 template <typename T>
155 std::ostream& operator<< (std::ostream& ost, const std::deque<T> & cont) {
156     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
157     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
158     return ost;
159 }
160
161 template <typename T>
162 std::ostream& operator<< (std::ostream& ost, const std::forward_list<T> & cont) {
163     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
164     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
165     return ost;
166 }
167
168
169 #endif // !TEST_HPP
```