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) x 0,6m x PI x 3.6 = 56,549 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 = 7,854m (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, 23. November 2025

Inhaltsverzeichnis

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

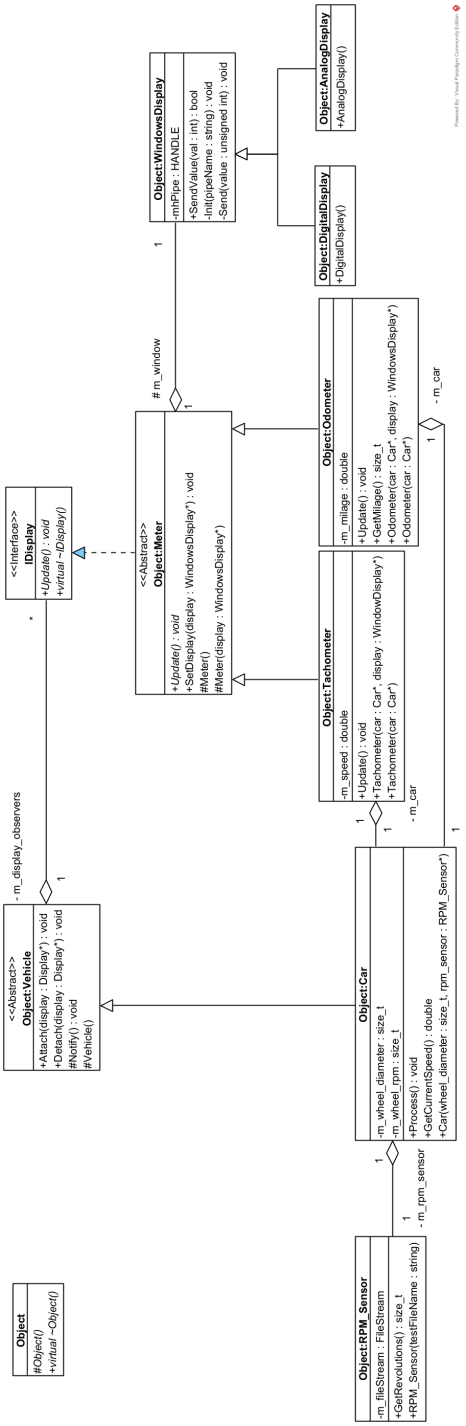
3.1 Designentscheidungen

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 Raddrehzahl zu lesen. Jeder Eintrag enthält eine Ganzzahl, die die Raddrehzahl in Umdrehungen pro Minute (RPM) angibt.

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

5 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis ./../doxy/html/index.html

6 Testprotokollierung

```
1
2  ****************************************
3              TESTCASE START
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 Odomerter
25 [Test OK] Result: (Expected: 0 == Result: 0)
26
27 Test Milage after one Process of Odomerter
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 ****************************************
38             TESTCASE START
39 ****************************************
40
41 Test normal operation in Tachometer Setup
42 [Test OK] Result: (Expected: true == Result: true)
```

```
43
44 Test nullptr Car in Tachometer CTOR
45 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
46
47 Test Display nullptr in CTOR of Tachometer
48 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
49
50 Test nullptr in CTOR of Tachometer
51 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
52
53 Test Car nullptr in Update of Tachometer
54 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
55
56 Test nullptr in Set Display
57 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
58
59
60 ****************************************
61
62
63 ****************************************
64              TESTCASE START
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
```

```
[Test OK] Result: (Expected: ERROR RPM sensor file was not found == Result:
    ↪  ERROR RPM sensor file was not found)

Test empty file in RPM_Sensor
[Test OK] Result: (Expected: ERROR RPM sensor could not read data from file
    ↪ . == Result: ERROR RPM sensor could not read data from file.)


*****************************************


*****************************************
              TESTCASE START
*****************************************

Test Car CTOR with RPM Nullptr
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)

Test Car CTOR with 0 Wheel Diameter
[Test OK] Result: (Expected: ERROR: Wheel Diameter cannot be 0! == Result:
    ↪ ERROR: Wheel Diameter cannot be 0!)

Test Car Attach with nullptr
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)

Test Car Detach with nullptr
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)

Test Car Detach with non attached observer
[Test OK] Result: (Expected: true == Result: true)

Test Car Get Current Speed
[Test OK] Result: (Expected: 18849 == Result: 18849)

Test Exception in TestCase
[Test OK] Result: (Expected: true == Result: true)

Test Exception End of File in Car Process
[Test OK] Result: (Expected: ERROR RPM sensor file has ended, theres no
    ↪ more data. == Result: ERROR RPM sensor file has ended, theres no more
    ↪  data.)


*****************************************
```

```
122
123 TEST OK!!
```

7 Quellcode

7.1 Object.h

```
/*****************************************************************//**
 * \file   Object.h
 * \brief  Base Object class for all other classes
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef OBJECT_H
#define OBJECT_H

#include <string>

class Object{
public:

   virtual ~Object(){}

protected:

   Object(){};

};

#endif // OBJECT_H
```

7.2 Vehicle.hpp

```cpp
/*****************************************************************//**
 * \file   Vehicle.hpp
 * \brief  Base class representing a generic vehicle that supports
 *         display observers following the Observer design pattern.
 *
 * The Vehicle class manages a collection of display observers that
 * implement the IDisplay interface. Observers can be attached or
 * detached at runtime.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef VEHICLE_HPP
#define VEHICLE_HPP

#include "Object.h"
#include "IDisplay.hpp"
#include <vector>

 /**
  * \class Vehicle
  * \brief Base class supporting observer management for display updates.
  */
class Vehicle : public Object {
public:

    /**
     * \brief Error message thrown when a null pointer is passed.
     */
    inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

    /**
     * \brief Container type alias for display observers.
     *
     * Stores shared pointers to IDisplay implementations.
     */
    using TCont = std::vector<IDisplay::Sptr>;

    /**
     * \brief Attaches a display observer to the vehicle.
     *
     * \param display Shared pointer to a display object implementing IDisplay.
     *
     * \throws std::string ERROR_NULLPTR if display is null.
     */
    void Attach(IDisplay::Sptr display);

    /**
     * \brief Detaches a display observer from the vehicle.
     *
     * \param display Shared pointer to a display object that should be removed.
     *
     * \throws std::string ERROR_NULLPTR if display is null.
     */
    void Detach(IDisplay::Sptr display);

protected:

    /**
     * \brief Notifies all attached display observers.
     *
     * This method loops through all observers stored in the container and
     * calls Update() on each non-null display.
     */
    void Notify() const;

    /**
     * \brief Protected default constructor.
     *
     * Prevents direct instantiation of Vehicle, as it should be subclassed.
     */
```

```
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

```cpp
/*****************************************************************//**
 * \file   Vehicle.cpp
 * \brief  Base class representing a generic vehicle that supports
 *         display observers following the Observer design pattern.
 *
 * The Vehicle class manages a collection of display observers that
 * implement the IDisplay interface. Observers can be attached or
 * detached at runtime.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "Vehicle.hpp"
#include "algorithm"

using namespace std;

void Vehicle::Attach(IDisplay::Sptr display)
{
   if (display == nullptr) throw Vehicle::ERROR_NULLPTR;

   m_display_observers.emplace_back(move(display));
}

void Vehicle::Detach(IDisplay::Sptr display)
{
   if (display == nullptr) throw Vehicle::ERROR_NULLPTR;

   const auto it = find(m_display_observers.cbegin(), m_display_observers.cend(), display);

   if (it != m_display_observers.cend()) {
      m_display_observers.erase(it);
   }
}

void Vehicle::Notify() const
{
   for_each(m_display_observers.cbegin(), m_display_observers.cend(),
      [](auto& obs) { if(obs != nullptr) obs->Update();});
}
```

7.4 Car.hpp

```cpp
/*****************************************************************//**
 * \file   Car.hpp
 * \brief  Declares the Car class, a Vehicle implementation using an RPM
 *         sensor to calculate its current speed.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef CAR_HPP
#define CAR_HPP

#include <string>
#include <string_view>
#include <memory>

#include "RPM_Sensor.hpp"
#include "Vehicle.hpp"

 /**
  * \class Car
  * \brief Vehicle subclass that calculates speed using an RPM sensor.
  */
class Car : public Vehicle {
public:

    /**
     * \brief Shared pointer alias for Car.
     */
    using Sptr = std::shared_ptr<Car>;

    /**
     * \brief Weak pointer alias for Car.
     */
    using Wptr = std::weak_ptr<Car>;

    /**
     * \brief Error message for invalid wheel diameter.
     */
    static inline const std::string ERROR_WHEEL_DIA_0 = "ERROR: Wheel Diameter cannot be 0!";

    /**
     * \brief Error message for null sensor pointer.
     */
    static inline const std::string ERROR_NULLPTR = "ERROR Nullptr";

    /**
     * \brief Retrieves RPM data from the sensor and updates internal state.
     *
     * This method calls RPM_Sensor::GetRevolutions() to update the current
     * wheel RPM. If the sensor throws an exception, the internal RPM value
     * is reset to zero before rethrowing the exception.
     *
     * After processing sensor data, this method notifies all observers
     * through Vehicle::Notify().
     *
     * \throws Any exception thrown by the underlying RPM sensor.
     */
    void Process();

    /**
     * \brief Computes the current speed of the car in kilometers per hour.
     *
     * The speed is calculated using the wheel RPM, wheel diameter (in mm),
     * and conversion constants.
     * \return The current speed in KPH.
     */
    double GetCurrentSpeed() const;

    /**
     * \brief Constructs a Car with a wheel diameter and an RPM sensor.
     *
```

```
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.5
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

```cpp
/*****************************************************************//**
 * \file   Car.cpp
 * \brief  Declares the Car class, a Vehicle implementation using an RPM
 *         sensor to calculate its current speed.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#include "Car.hpp"
#include <numbers>

void Car::Process()
{
   try {
      // Get Revolutions
      m_wheel_rmp = m_wheel_rpm_sensor->GetRevolutions();
   }
   catch (...) {

      m_wheel_rmp = 0;

      throw; // rethrow exception to inform user of the exception
   }

   Notify();
}

double Car::GetCurrentSpeed() const
{
   return ((m_wheel_rmp / seconds_in_min) * m_wheel_diameter * std::numbers::pi * mps_to_kph) / mm_in_m;
}

Car::Car(const size_t & wheel_diameter, RPM_Sensor::Sptr wh_rpm_sen) : m_wheel_rmp{0}
{
   if (wheel_diameter == 0) throw Car::ERROR_WHEEL_DIA_0;
   if (wh_rpm_sen == nullptr) throw Car::ERROR_NULLPTR;

   m_wheel_rpm_sensor = std::move(wh_rpm_sen);

   m_wheel_diameter = wheel_diameter;
}
```

7.6 IDisplay.hpp

```cpp
/*****************************************************************//**
 * \file   IDisplay.hpp
 * \brief  Interface for all display types used by vehicles.
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef IDISPLAY_HPP
#define IDISPLAY_HPP

#include <memory>

 /**
  * \class IDisplay
  * \brief Interface for display observer implementations.
  */
class IDisplay {
public:

    /**
     * \brief Shared pointer alias for IDisplay.
     */
    using Sptr = std::shared_ptr<IDisplay>;

    /**
     * \brief Called when the observed subject updates its state.
     */
    virtual void Update() = 0;

    /**
     * \brief Virtual destructor.
     *
     * Ensures proper cleanup of derived display types.
     */
    virtual ~IDisplay() = default;
};

#endif // !IDISPLAY_HPP
```

7.7 Meter.hpp

```cpp
/*****************************************************************//**
 * \file   Meter.hpp
 * \brief  Abstract base class for all meter display types.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef METER_HPP
#define METER_HPP

#include "Object.h"
#include "IDisplay.hpp"
#include "WindowsDisplay.h"

 /**
  * \class Meter
  * \brief Abstract base class for all meter display components.
  */
class Meter : public Object, public IDisplay {
public:

    /**
     * \brief Error message thrown when a null pointer is passed.
     */
    inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

    /**
     * \brief Sets the WindowsDisplay instance used by this meter.
     * \param display Shared pointer to a WindowsDisplay.
     *
     * \throws std::string ERROR_NULLPTR if display is null.
     */
    void SetDisplay(WindowsDisplay::SPtr display);

protected:

    /**
     * \brief Constructs a Meter with an initial WindowsDisplay instance.
     *
     * \param display Shared pointer to a WindowsDisplay.
     *
     * \throws std::string ERROR_NULLPTR if display is null.
     */
    Meter(WindowsDisplay::SPtr display);

    /**
     * \brief Default protected constructor.
     *
     * Allows derived classes to be created without immediately providing
     * a display. A valid display must later be set through SetDisplay().
     */
    Meter() = default;

    /**
     * \brief Pointer to the display window used by the meter.
     */
    WindowsDisplay::SPtr m_window;

private:
    // No private members currently required.
};

#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

```
/*****************************************************************//**
 * \file   Tachometer.hpp
 * \brief  Display that shows the current speed of a car.
 *
 * The Tachometer observes a Car instance and displays its current speed.
 * It derives from Meter and serves as an observer in the Vehicle system.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef TACHOMETER_HPP
#define TACHOMETER_HPP

#include "Object.h"
#include "Meter.hpp"
#include "Car.hpp"

 /**
  * \class Tachometer
  * \brief Meter implementation that displays the real-time speed of a vehicle.
  */
class Tachometer : public Meter {
public:

    /**
     * \brief Error message thrown when a null pointer is passed.
     */
    inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

    /**
     * \brief Shared pointer alias for Tachometer.
     */
    using Sptr = std::shared_ptr<Tachometer>;

    /**
     * \brief Updates the tachometer display with the current speed.
     *
     *
     * \throws std::string ERROR_NULLPTR if the car pointer is expired.
     */
    virtual void Update() override;

    /**
     * \brief Constructs a Tachometer with both a Car and a display.
     *
     * \param car Shared pointer to the observed Car.
     * \param display Shared pointer to a WindowsDisplay instance.
     *
     * \throws std::string ERROR_NULLPTR if car or display is null.
     */
    Tachometer(Car::Sptr car, WindowsDisplay::SPtr display);

    /**
     * \brief Constructs a Tachometer with a Car but without a display.
     *
     * A valid display can be attached later using SetDisplay(), inherited
     * from Meter. The Car pointer must not be null and is stored as a
     * weak pointer.
     *
     * \param car Shared pointer to the observed Car.
     *
     * \throws std::string ERROR_NULLPTR if car is null.
     */
    Tachometer(Car::Sptr car);

private:

    /**
     * \brief Weak pointer to the observed Car.
     *
     * 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

```cpp
/*****************************************************************//**
 * \file   Tachometer.cpp
 * \brief  Display that shows the current speed of a car.
 *
 * The Tachometer observes a Car instance and displays its current speed.
 * It derives from Meter and serves as an observer in the Vehicle system.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#include "Tachometer.hpp"

#include <iostream>

void Tachometer::Update()
{
   Car::Sptr car = m_car.lock();

   // check if sptr is valid
   if (car == nullptr) throw Tachometer::ERROR_NULLPTR;

   m_speed = car->GetCurrentSpeed();

   if (m_window != nullptr) m_window->SendValue(static_cast<unsigned int>(m_speed));
}

Tachometer::Tachometer(Car::Sptr car, WindowsDisplay::SPtr display) :  m_speed{0}, Meter{move(display)}
{
   if (car == nullptr) throw Tachometer::ERROR_NULLPTR;

   m_car = move(car);

}

Tachometer::Tachometer(Car::Sptr car) : m_speed{ 0 }
{
   if (car == nullptr) throw Tachometer::ERROR_NULLPTR;

   m_car = move(car);
}
```

7.11 Odometer.hpp

```cpp
/*****************************************************************//**
 * \file   Odometer.hpp
 * \brief  Display that calculates and shows the current mileage of a car.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef ODOMETER_HPP
#define ODOMETER_HPP

#include "Object.h"
#include "Meter.hpp"
#include "Car.hpp"

 /**
  * \class Odometer
  * \brief Meter implementation that calculates and displays vehicle mileage.
  *
  * The Odometer observes a Car instance and increases its internal mileage
  * value based on the car's current speed and a fixed update interval.
  * Mileage is stored as a double for accumulation precision, but exposed
  * as a size_t when queried.
  *
  * The car pointer is stored as a weak pointer to avoid ownership cycles.
  * If the car expires, Update() throws an exception.
  */
class Odometer : public Meter {
public:

    /**
     * \brief Error message thrown when a null pointer is passed.
     */
    inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

    /**
     * \brief Update interval in milliseconds.
     *
     * This value determines how frequently the odometer expects to be
     * updated. It is used in the mileage calculation:
     *   mileage += speed_kph * (interval_ms / milliseconds_per_hour)
     */
    inline static const size_t Update_Intervall = 500;

    /**
     * \brief Shared pointer alias for Odometer.
     */
    using Sptr = std::shared_ptr<Odometer>;

    /**
     * \brief Updates the mileage and forwards the new value to the display.
     *
     * \throws std::string ERROR_NULLPTR if the car pointer is expired.
     */
    virtual void Update() override;

    /**
     * \brief Returns the accumulated mileage.
     *
     * \return Mileage as a size_t value.
     */
    size_t GetMilage() const;

    /**
     * \brief Constructs an Odometer with a Car and a display.
     *
     * \param car Shared pointer to the observed Car.
     * \param display Shared pointer to a WindowsDisplay instance.
     *
     * \throws std::string ERROR_NULLPTR if either pointer is null.
     */
    Odometer(Car::Sptr car, WindowsDisplay::SPtr display);
```

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

7.12 Odometer.cpp

```cpp
/*****************************************************************//**
 * \file   Odometer.cpp
 * \brief  Display that calculates and shows the current mileage of a car.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "Odometer.hpp"

void Odometer::Update()
{

   Car::Sptr car = m_car.lock();

   if (car == nullptr) throw Odometer::ERROR_NULLPTR;

   // get the absolute value of the Current Speed
   // negative Speed does not make the milage go down!
   const double speed = abs(car->GetCurrentSpeed());

   // Integrate the speed over the update interval to get the distance
   m_milage = m_milage + (speed * Odometer::Update_Intervall)/ mseconds_in_hours;

   // If a display is set, send the new milage value
   if(m_window != nullptr) m_window->SendValue(static_cast<unsigned int>(m_milage));
}

Odometer::Odometer(Car::Sptr car, WindowsDisplay::SPtr display) : m_milage { 0 }, Meter{ move(display) }
{
   if (car == nullptr) throw Odometer::ERROR_NULLPTR;

   m_car = move(car);
}

Odometer::Odometer(Car::Sptr car) : m_milage{ 0 }
{
   if (car == nullptr) throw Odometer::ERROR_NULLPTR;

   m_car = move(car);
}


size_t Odometer::GetMilage() const
{
   return static_cast<size_t>(m_milage);
}
```

7.13 RPM_Sensor.hpp

```cpp
/*****************************************************************//**
 * \file   RPM_Sensor.hpp
 * \brief  A "sensor" for returning individual readings when
 * GetRevolutioins() is called.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef RPM_SENSOR_HPP
#define RPM_SENSOR_HPP

#include "Object.h"
#include <string>
#include <string_view>
#include <memory>
#include <fstream>

class RPM_Sensor : public Object {
public:
    inline static const std::string ERROR_NULLPTR = "ERROR: Nullptr";
    inline static const std::string ERROR_SENSOR_FILE_NOT_FOUND = "ERROR: RPM sensor file was not found";
    inline static const std::string ERROR_SENSOR_INVALID_DATA_INPUT = "ERROR: RPM sensor could not read data from file.";
    inline static const std::string ERROR_SENSOR_EOF = "ERROR: RPM sensor file has ended, theres no more data.";

    /**
     * \brief Shared pointer tyope for RPM_Sensor
     */
    using Sptr = std::shared_ptr<RPM_Sensor>;

    /**
     * \brief Returns current rpm. This is achieved by parsing
     * from a testfile - if the end of the file is reached a
     * exception is thrown (ERROR_SENSOR_EOF). This has to
     * be handled by the user of this class.
     * \return unsigned int revs
     */
    size_t GetRevolutions();

    /**
     * \brief RPM_Sensor constructor
     * throws error (ERROR_SENSOR_FILE_NOT_FOUND)
     * if the provided file/path is invalid.
     */
    RPM_Sensor(std::string_view testFileName);

    /**
     * \brief custom destructor is needed to
     * close ifstream.
     */
    ~RPM_Sensor();

    // delete CopyCtor and Assign Operator because we warp a file stream
    // which should not be copied or assigned!
    RPM_Sensor(RPM_Sensor& s) = delete;
    void operator= (RPM_Sensor s) = delete;

private:
    // open a filestream when sensor is constructed
    // close when destructor is called.
    std::ifstream m_fileStream;
};

#endif // !RPM_SENSOR_HPP
```

7.14 RPM_Sensor.cpp

```cpp
/*****************************************************************//**
 * \file   RPM_Sensor.cpp
 * \brief  A "sensor" for returning individual readings when
 * GetRevolutioins() is called.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#include "RPM_Sensor.hpp"

#include <algorithm>
#include <sstream>

size_t RPM_Sensor::GetRevolutions()
{
    std::string sensor_reading;
    std::stringstream converter;
    size_t sensor_value = 0;

    if (m_fileStream.eof())
        throw ERROR_SENSOR_EOF;

    if(m_fileStream.fail())
        throw ERROR_SENSOR_INVALID_DATA_INPUT;

    m_fileStream >> sensor_reading;

    if (sensor_reading.empty())
        throw ERROR_SENSOR_INVALID_DATA_INPUT;

    // check if all of the readings are digits
    if (!std::all_of(sensor_reading.cbegin(), sensor_reading.cend(), ::isdigit))
        throw ERROR_SENSOR_INVALID_DATA_INPUT;

    // use Stringstream for type Conversion
    converter << sensor_reading;
    converter >> sensor_value;

    return sensor_value;
}

RPM_Sensor::RPM_Sensor(std::string_view testFileName)
{
    m_fileStream = std::ifstream(testFileName.data());

    if (!m_fileStream.is_open())
        throw ERROR_SENSOR_FILE_NOT_FOUND;
}

RPM_Sensor::~RPM_Sensor()
{
    m_fileStream.close();
}
```

7.15 main.cpp

```cpp
/*****************************************************************//**
 * \file   main.cpp
 * \brief  Test Driver for the Drive Sim
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

//zur Verwendung der analogen und digitalen Anzeige
#include "WindowsDisplay.h"

#include "RPM_Sensor.hpp"
#include "Car.hpp"
#include "Tachometer.hpp"
#include "Odometer.hpp"

#include "Test.hpp"
#include "vld.h"
#include <fstream>
#include <iostream>
#include <cassert>

#define WriteOutputFile ON

using namespace std;

static bool TestOdometer(std::ostream & ost = std::cout);
static bool TestTachometer(std::ostream & ost = std::cout);
static bool TestRPMSensor(std::ostream& ost = std::cout);
static bool TestCar(std::ostream& ost = std::cout);

int main()
{
    bool TestOK = true;

    ofstream output{ "output.txt" };

    try {

        TestOK = TestOK && TestOdometer();
        TestOK = TestOK && TestTachometer();
        TestOK = TestOK && TestRPMSensor();
        TestOK = TestOK && TestCar();


        if (WriteOutputFile) {

            TestOK = TestOK && TestOdometer(output);
            TestOK = TestOK && TestTachometer(output);
            TestOK = TestOK && TestRPMSensor(output);
            TestOK = TestOK && TestCar(output);

            if (TestOK) {
                output << TestCaseOK;
            }
            else {
                output << TestCaseFail;
            }

            output.close();
        }

        if (TestOK) {
            cout << TestCaseOK;
        }
        else {
            cout << TestCaseFail;
        }
    }
    catch (const string& err) {
        cerr << err << TestCaseFail;
    }
```

```
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_Intervall);
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␣Odometer␣CTOR", 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␣in␣CTOR␣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␣in␣CTOR␣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 = "Unhandelt Exception";
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 = "Unhandelt Exception";
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 Odomerter", static_cast<size_t>(0), OdoMeter->GetMilage());
267 
268         AudiA3->Process();
269 
270         TestOK = TestOK && check_dump(ost, "Test Milage after one Process of Odomerter", 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 = "Unhandelt Exception";
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 
295 bool TestTachometer(std::ostream& ost)
296 {
297     assert(ost.good());
```

```
298
299     ost << TestStart;
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␣Tachometer␣CTOR", 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␣in␣CTOR␣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␣in␣CTOR␣of␣Tachometer", Tachometer::ERROR_NULLPTR, error_msg);
385     error_msg.clear();
386 
387 
388     try {
389         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
390         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
391         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3);
392         AudiA3.reset(); // <-- Free Car
393         TachMeter->Update(); // <-- throws exception Car not set
394     }
395     catch (const string& err) {
396         error_msg = err;
397     }
398     catch (bad_alloc const& error) {
399         error_msg = error.what();
400     }
401     catch (const exception& err) {
402         error_msg = err.what();
403     }
404     catch (...) {
405         error_msg = "Unhandelt␣Exception";
406     }
407 
408     TestOK = TestOK && check_dump(ost, "Test␣Car␣nullptr␣in␣Update␣of␣Tachometer", Tachometer::ERROR_NULLPTR, error_msg);
409     error_msg.clear();
410 
411     try {
412         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
413         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
414         Tachometer::Sptr TachMeter = make_shared<Tachometer>(AudiA3);
415 
416         TachMeter->SetDisplay(nullptr);
417     }
418     catch (const string& err) {
419         error_msg = err;
420     }
421     catch (bad_alloc const& error) {
422         error_msg = error.what();
423     }
424     catch (const exception& err) {
425         error_msg = err.what();
426     }
427     catch (...) {
428         error_msg = "Unhandelt␣Exception";
429     }
430 
431     TestOK = TestOK && check_dump(ost, "Test␣nullptr␣in␣Set␣Display", Tachometer::ERROR_NULLPTR, error_msg);
432     error_msg.clear();
433 
434     ost << TestEnd;
435 
436     return TestOK;
437 }
438 
439 bool TestRPMSensor(std::ostream& ost)
440 {
441     assert(ost.good());
442     ost << TestStart;
443 
444     bool TestOK = true;
445     string error_msg;
446 
447     // 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 = "Unhandelt Exception";
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 = "Unhandelt Exception";
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 
492     try {
493         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid2.txt");
494         size_t revs = sen->GetRevolutions();
495     }
496     catch (const string& err) {
497         error_msg = err;
498     }
499     catch (bad_alloc const& error) {
500         error_msg = error.what();
501     }
502     catch (const exception& err) {
503         error_msg = err.what();
504     }
505     catch (...) {
506         error_msg = "Unhandelt Exception";
507     }
508 
509     // check if exception was thrown
510     TestOK = TestOK && check_dump(ost, "Test invalid RPM Data (-1000)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
511     error_msg.clear();
512 
513     try {
514         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid3.txt");
515         size_t revs = sen->GetRevolutions();
516     }
517     catch (const string& err) {
518         error_msg = err;
519     }
520     catch (bad_alloc const& error) {
521         error_msg = error.what();
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 = "Unhandelt Exception";
623     }
624
625     // check if exception was thrown
626     TestOK = TestOK && check_dump(ost, "Test Car CTOR 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 = "Unhandelt Exception";
645     }
646
647     // check if exception was thrown
648     TestOK = TestOK && check_dump(ost, "Test Car CTOR 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 = "Unhandelt Exception";
667     }
668
669     TestOK = TestOK && check_dump(ost, "Test Car Attach 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 = "Unhandelt Exception";
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 = "Unhandelt Exception";
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     try {
717         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
718         Car::Sptr c = make_shared<Car>(100,sen );
719         c->Process();
720         TestOK = TestOK && check_dump(ost, "Test Car Get Current Speed", static_cast<size_t>(18849), static_cast<size_t>(c->GetCurrentS
721     }
722     catch (const string& err) {
723         error_msg = err;
724     }
725     catch (bad_alloc const& error) {
726         error_msg = error.what();
727     }
728     catch (const exception& err) {
729         error_msg = err.what();
730     }
731     catch (...) {
732         error_msg = "Unhandelt Exception";
733     }
734
735     TestOK = TestOK && check_dump(ost, "Test Exception in TestCase", true, error_msg.empty());
736     error_msg.clear();
737
738
739     try {
740         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
741         Car::Sptr c = make_shared<Car>(100,sen );
742         while(1) c->Process();
743     }
744     catch (const string& err) {
745         error_msg = err;
746     }
747     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 = "Unhandelt Exception";
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

```
/*****************************************************************//**
 * \file   Test.hpp
 * \brief  File that provides a Test Function with a formated output
 *
 * \author Simon
 * \date   April 2025
 *********************************************************************/
#ifndef TEST_HPP
#define TEST_HPP

#include <string>
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <forward_list>

#define ON 1
#define OFF 0
#define COLOR_OUTPUT OFF

// Definitions of colors in order to change the color of the output stream.
const std::string colorRed = "\x1B[31m";
const std::string colorGreen = "\x1B[32m";
const std::string colorWhite = "\x1B[37m";

inline std::ostream& RED(std::ostream& ost) {
   if (ost.good()) {
      ost << colorRed;
   }
   return ost;
}
inline std::ostream& GREEN(std::ostream& ost) {
   if (ost.good()) {
      ost << colorGreen;
   }
   return ost;
}
inline std::ostream& WHITE(std::ostream& ost) {
   if (ost.good()) {
      ost << colorWhite;
   }
   return ost;
}

inline std::ostream& TestStart(std::ostream& ost) {
   if (ost.good()) {
      ost << std::endl;
      ost << "****************************************" << std::endl;
      ost << "          TESTCASE START          " << std::endl;
      ost << "****************************************" << std::endl;
      ost << std::endl;
   }
   return ost;
}

inline std::ostream& TestEnd(std::ostream& ost) {
   if (ost.good()) {
      ost << std::endl;
      ost << "****************************************" << std::endl;
      ost << std::endl;
   }
   return ost;
}

inline std::ostream& TestCaseOK(std::ostream& ost) {

#if COLOR_OUTPUT
   if (ost.good()) {
      ost << colorGreen << "TEST OK!!" << colorWhite << std::endl;
   }
#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     }
88  #else
89     if (ost.good()) {
90        ost << "TEST␣FAILED␣!!" << std::endl;
91
92     }
93  #endif // COLOR_OUTPUT
94
95     return ost;
96  }
97
98  /**
99     * \brief function that reports if the testcase was successful.
100    *
101    * \param testcase String that indicates the testcase
102    * \param succsessful true -> reports to cout test OK
103    * \param succsessful false -> reports test failed
104    */
105 template <typename T>
106 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
107    if (ostr.good()) {
108 #if COLOR_OUTPUT
109       if (expected == result) {
110          ostr << testcase << std::endl <<  colorGreen << "[Test␣OK]␣" << colorWhite <<"Result:␣(Expected:␣" << std::boolalpha << expect
   << std::noboolalpha << std::endl << std::endl;
111       }
112       else {
113          ostr << testcase << std::endl << colorRed << "[Test␣FAILED]␣" << colorWhite << "Result:␣(Expected:␣" << std::boolalpha << expe
   << std::noboolalpha << std::endl << std::endl;
114       }
115 #else
116       if (expected == result) {
117          ostr << testcase << std::endl << "[Test␣OK]␣"  << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣==" << "␣Result:␣"
118       }
119       else {
120          ostr << testcase << std::endl  << "[Test␣FAILED]␣"  << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Resul
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
```