
# Systemdokumentation
# Projekt DriveSim

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 21. November 2025

# Inhaltsverzeichnis

# 1 Organisatorisches

## 1.1 Team


- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@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,
    - RPM Sensor

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

## 1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich x 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
- Anbfragen der anzuzeigenden Werte vom Vehicle
- Darstellung der Werte

**Funktionen des Odometer**

- Berechnung der gefahrenen Strecke anhand der Raddrehzahl

**Funktionen des Tachometer**

- Messung der Drehzahl des Motors

# 3 Systementwurf

## 3.1 Designentscheidungen

Im Klassendiagramm wurd als Pattern das *Observer Pattern* verwendet, da in diesem fall ein Fahzeug 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.

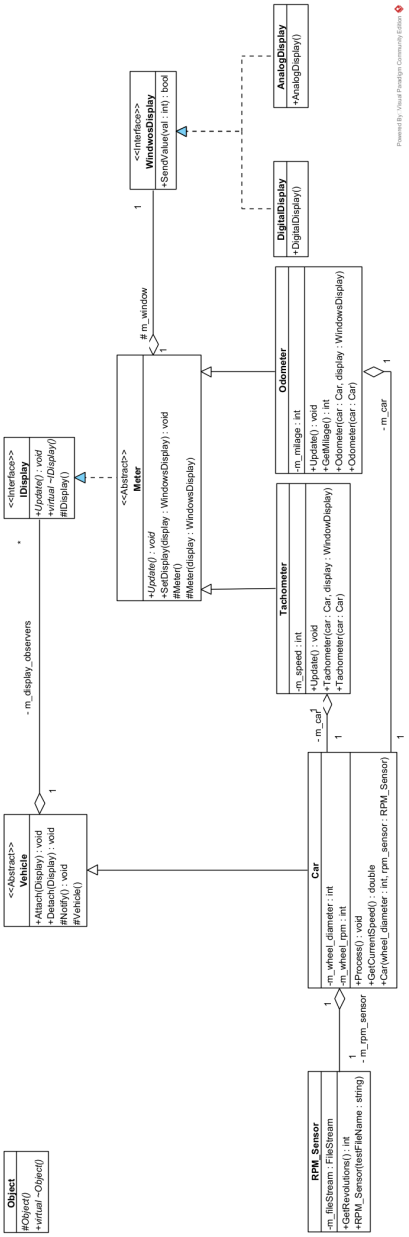
Die Displays können vom Fahrzeug die aufbereiteten Daten (z.B. Geschwindigkeit, Drehzahl) abfragen, indem sie die entsprechenden Methoden des Fahrzeugs aufrufen.

Die Abstakte *Meter* dient dazu, das WindowDisplay Interface zu implementieren und die gemeinsamen Funktionen der verschiedenen Anzeigetypen (Tachometer, Odometer, RPM Sensor) zu bündeln.

# 4 Dokumentation der Komponenten (Klassen)

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

# Klassendiagramm



# 5 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!!
```

# 6 Quellcode

## 6.1 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
```

## 6.2 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. Whenever the vehicle's internal state changes,
 * derived classes can call Notify() to update all attached displays.
 *
 * \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.
  *
  * The Vehicle class is responsible for managing observers that display
  * information about the vehicle. It provides methods to attach and detach
  * display objects, as well as a protected Notify() method that derived
  * classes can call when their internal state has changed.
  *
  * This class uses shared pointers for observer references. A null pointer
  * is treated as an invalid argument and produces an exception.
  */
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.
     *
     * The display pointer must not be null. The observer is added to the
     * internal container and will receive updates on the next Notify().
     *
     * \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.
     *
     * If the observer exists in the internal container, it is removed.
     * Passing a null pointer is considered an invalid operation.
     *
     * \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);

```

```
73 protected:
74
75     /**
76      * \brief Notifies all attached display observers.
77      *
78      * This method loops through all observers stored in the container and
79      * calls Update() on each non-null display. Derived classes should call
80      * this method whenever their observable state changes.
81      */
82     void Notify();
83
84     /**
85      * \brief Protected default constructor.
86      *
87      * Prevents direct instantiation of Vehicle, as it should be subclassed.
88      */
89     Vehicle() = default;
90
91 private:
92
93     /**
94      * \brief Container holding all attached display observers.
95      */
96     TCont m_display_observers;
97 };
98
99 #endif // !VEHICLE_HPP
```

## 6.3 Vehicle.cpp

```cpp
#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;

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

## 6.4 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.
     */
    inline static 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.
     *
     * The calculation is based on:
     * - converting RPM to revolutions per second
     * - converting wheel diameter to circumference (diameter * pi)
     * - converting millimeters to meters
     * - converting meters per second to kilometers per hour
     *
```

```
73      * \return The current speed in KPH.
74      */
75     double GetCurrentSpeed();
76
77     /**
78      * \brief Constructs a Car with a wheel diameter and an RPM sensor.
79      *
80      * \param wheel_diameter Wheel diameter in millimeters.
81      * \param wh_rpm_sen Shared pointer to an RPM sensor instance.
82      *
83      * \throws std::string ERROR_WHEEL_DIA_0 if wheel_diameter is zero.
84      * \throws std::string ERROR_NULLPTR if wh_rpm_sen is null.
85      */
86     Car(const size_t& wheel_diameter, RPM_Sensor::Sptr wh_rpm_sen);
87
88 private:
89     /**
90      * \brief Last measured wheel revolutions per minute.
91      */
92     size_t m_wheel_rmp;
93
94     /**
95      * \brief Wheel diameter in millimeters.
96      */
97     size_t m_wheel_diameter;
98
99     /**
100     * \brief Pointer to the RPM sensor used for measurement.
101     */
102    RPM_Sensor::Sptr m_wheel_rpm_sensor;
103
104    /**
105     * \brief Conversion factor from meters per second to kilometers per hour.
106     */
107    const double mps_to_kph = 3.6;
108
109    /**
110     * \brief Number of seconds in one minute.
111     */
112    const double seconds_in_min = 60;
113
114    /**
115     * \brief Millimeters in one meter.
116     */
117    const double mm_in_m = 1000;
118 };
119
120 #endif // !CAR_HPP
```

## 6.5 Car.cpp

```cpp
#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()
{
   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 = move(wh_rpm_sen);

   m_wheel_diameter = wheel_diameter;
}
```

## 6.6 IDisplay.hpp

```cpp
/*****************************************************************//**
 * \file   IDisplay.hpp
 * \brief  Interface for all display types used by vehicles.
 *
 * The IDisplay interface represents any output or visualization component
 * that can receive update notifications from a vehicle. It is designed to
 * be used with the Observer design pattern. Implementations must override
 * the Update() function to react to state changes.
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef IDISPLAY_HPP
#define IDISPLAY_HPP

#include <memory>

 /**
  * \class IDisplay
  * \brief Interface for display observer implementations.
  *
  * The IDisplay class defines the contract for any display that observes
  * a vehicle. When the vehicle's state changes, the Vehicle class calls
  * Update() on all attached observers. Classes implementing this interface
  * must provide the Update() behavior appropriate for their display type.
  */
class IDisplay {
public:

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

    /**
     * \brief Called when the observed subject updates its state.
     *
     * This method is invoked by the Vehicle class through its Notify()
     * function. Implementations must override this method to update their
     * displayed information.
     */
    virtual void Update() = 0;

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

protected:

    /**
     * \brief Protected default constructor.
     *
     * Prevents direct instantiation of the interface and enforces
     * implementation through derived classes.
     */
    IDisplay() = default;
};

#endif // !IDISPLAY_HPP
```

## 6.7 Meter.hpp

```cpp
/*****************************************************************//**
 * \file   Meter.hpp
 * \brief  Abstract base class for all meter display types.
 *
 * The Meter class acts as a display component in the Observer pattern.
 * It derives from IDisplay and represents a meter that visualizes data
 * using a WindowsDisplay instance. Concrete meter types update their
 * visual output when Update() is called by the observed subject.
 *
 * \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.
  *
  * The Meter class provides the foundation for visual display elements
  * that react to state changes from a subject (for example, a Vehicle).
  * It maintains a WindowsDisplay instance used for rendering.
  *
  * Display pointers are stored using move semantics. Null pointers are
  * not permitted and will result in an exception. Subclasses should rely
  * on the protected member m_window when performing drawing operations.
  */
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.
     *
     * The provided pointer must not be null. The display pointer is moved
     * into internal storage, transferring ownership to the Meter object.
     * Existing display pointers (if any) are overwritten.
     *
     * \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.
     *
     * The provided display pointer must not be null. Ownership of the
     * pointer is transferred into the Meter through move semantics.
     * Derived meter classes typically call this constructor to ensure
     * that drawing capabilities are available immediately.
     *
     * \param display Shared pointer to a WindowsDisplay.
     *
     * \throws std::string ERROR_NULLPTR if display is null.
     */
    Meter(WindowsDisplay::SPtr display);

    /**
     * \brief Default protected constructor.
     *
```

```
73      * Allows derived classes to be created without immediately providing
74      * a display. A valid display must later be set through SetDisplay().
75      */
76     Meter() = default;
77
78     /**
79      * \brief Pointer to the display window used by the meter.
80      *
81      * This pointer is assigned through either the constructor or
82      * SetDisplay(), using move semantics. Derived classes use this
83      * member for rendering updated values.
84      */
85     WindowsDisplay::SPtr m_window;
86
87 private:
88     // No private members currently required.
89 };
90
91 #endif // !METER_HPP
```

## 6.8 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 }
```

## 6.9 Tachometer.hpp

```cpp
/*****************************************************************//**
 * \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.
  *
  * The Tachometer observes a Car instance through a weak pointer. When
  * Update() is called, it retrieves the current speed from the car and
  * forwards the value to the attached WindowsDisplay, if present.
  *
  * If the observed Car instance is no longer valid, Update() throws an
  * exception to indicate incorrect usage or lifetime issues.
  */
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.
     *
     * The method locks the weak pointer to the observed Car. If locking
     * fails, an exception is thrown. The car's current speed is then read
     * and stored internally. If a display window is available, the value
     * is forwarded to the WindowsDisplay instance.
     *
     * \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.
     *
     * The display pointer and car pointer must not be null. Ownership of
     * the display is transferred into the Meter base class via move
     * semantics. The Car is referenced through a weak pointer.
     *
     * \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
```

```
73      * from Meter. The Car pointer must not be null and is stored as a
74      * weak pointer.
75      *
76      * \param car Shared pointer to the observed Car.
77      *
78      * \throws std::string ERROR_NULLPTR if car is null.
79      */
80     Tachometer(Car::Sptr car);
81
82 private:
83
84     /**
85      * \brief Weak pointer to the observed Car.
86      *
87      * Prevents ownership cycles between Car and Tachometer. If the Car
88      * instance is destroyed, Update() will detect this via lock().
89      */
90     Car::Wptr m_car;
91
92     /**
93      * \brief Last measured speed in kilometers per hour.
94      *
95      * This value is updated during each call to Update() and forwarded
96      * to the WindowsDisplay using SendValue().
97      */
98     double m_speed;
99 };
100
101 #endif // !TACHOMETER_HPP
```

## 6.10 Tachometer.cpp

```cpp
#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);
}
```

## 6.11 Odometer.hpp

```cpp
/*****************************************************************//**
 * \file   Odometer.hpp
 * \brief  Display that calculates and shows the current mileage of a car.
 *
 * The Odometer class observes a Car instance and accumulates mileage
 * based on the car's speed. It derives from Meter and therefore acts as
 * a display component. Updates occur when the subject notifies this
 * meter through the Observer pattern.
 *
 * \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.
     *
     * The method attempts to lock the weak pointer to the observed Car.
     * If locking fails, the method throws ERROR_NULLPTR.
     *
     * The current speed is retrieved from the Car and converted into
     * distance traveled during the update interval. The resulting mileage
     * is forwarded to the associated WindowsDisplay instance if available.
     *
     * \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.
```

```
73      */
74     size_t GetMilage() const;
75
76     /**
77      * \brief Constructs an Odometer with a Car and a display.
78      *
79      * Both pointers must be non-null. The Car pointer is stored as a weak
80      * reference. Display ownership is transferred via move semantics.
81      *
82      * \param car Shared pointer to the observed Car.
83      * \param display Shared pointer to a WindowsDisplay instance.
84      *
85      * \throws std::string ERROR_NULLPTR if either pointer is null.
86      */
87     Odometer(Car::Sptr car, WindowsDisplay::SPtr display);
88
89     /**
90      * \brief Constructs an Odometer with a Car and without a display.
91      *
92      * The display must be set later using SetDisplay() before visual output
93      * is possible.
94      *
95      * \param car Shared pointer to the Car being observed.
96      *
97      * \throws std::string ERROR_NULLPTR if car is null.
98      */
99     Odometer(Car::Sptr car);
100
101 private:
102
103     /**
104      * \brief Weak pointer to the observed Car.
105      *
106      * Ensures there is no ownership cycle between Car and Odometer.
107      */
108     Car::Wptr m_car;
109
110     /**
111      * \brief Accumulated mileage in kilometers (stored as double).
112      *
113      * Internally stored as double for precision. Exposed as size_t via
114      * GetMilage() for external usage.
115      */
116     double m_milage;
117
118     /**
119      * \brief Number of milliseconds in one hour.
120      *
121      * Used for converting speed (KPH) into incremental distance
122      * based on the update interval.
123      */
124     static const size_t mseconds_in_hours = 3600000;
125 };
126
127 #endif // !ODOMETER_HPP
```

## 6.12 Odometer.cpp

```cpp
#include "Odometer.hpp"

void Odometer::Update()
{

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

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

    const double speed = abs(car->GetCurrentSpeed());

    m_milage = m_milage + (speed * Odometer::Update_Intervall)/ mseconds_in_hours;

    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);
}
```

## 6.13 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 fiel 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 to prevent untestet behavour.
   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
```

## 6.14 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;

    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 static_cast<size_t>(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();
}
```

## 6.15 main.cpp

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

#include <iostream>
#include <list>
#include <iomanip>
#include <algorithm>
#include <sstream>
#include <memory>
#include <windows.h>

//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>

#include <cstdio>

using namespace std;

#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;
```

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

```
148         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
149         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
150         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
151         AudiA3->Attach(OdoMeter);
152     }
153     catch (const string& err) {
154         error_msg = err;
155     }
156     catch (bad_alloc const& error) {
157         error_msg = error.what();
158     }
159     catch (const exception& err) {
160         error_msg = err.what();
161     }
162     catch (...) {
163         error_msg = "Unhandelt␣Exception";
164     }
165
166     TestOK = TestOK && check_dump(ost, "Test␣normal␣operation␣in␣Odometer␣Setup", true, error_msg.empty());
167     error_msg.clear();
168
169     try {
170         Odometer::Sptr OdoMeter = make_shared<Odometer>(nullptr);
171     }
172     catch (const string& err) {
173         error_msg = err;
174     }
175     catch (bad_alloc const& error) {
176         error_msg = error.what();
177     }
178     catch (const exception& err) {
179         error_msg = err.what();
180     }
181     catch (...) {
182         error_msg = "Unhandelt␣Exception";
183     }
184
185     TestOK = TestOK && check_dump(ost, "Test␣nullptr␣Car␣in␣Odometer␣CTOR", Odometer::ERROR_NULLPTR, error_msg);
186     error_msg.clear();
187
188     try {
189         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
190         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
191         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3, nullptr);
192     }
193     catch (const string& err) {
194         error_msg = err;
195     }
196     catch (bad_alloc const& error) {
197         error_msg = error.what();
198     }
199     catch (const exception& err) {
200         error_msg = err.what();
201     }
202     catch (...) {
203         error_msg = "Unhandelt␣Exception";
204     }
205
206     TestOK = TestOK && check_dump(ost, "Test␣Display␣nullptr␣in␣CTOR␣of␣Odometer", Odometer::ERROR_NULLPTR, error_msg);
207     error_msg.clear();
208     try {
209         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
210         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
211         Odometer::Sptr OdoMeter = make_shared<Odometer>(nullptr, nullptr);
212     }
213     catch (const string& err) {
214         error_msg = err;
215     }
216     catch (bad_alloc const& error) {
217         error_msg = error.what();
218     }
219     catch (const exception& err) {
220         error_msg = err.what();
221     }
222     catch (...) {
```

```
223         error_msg = "Unhandelt Exception";
224     }
225
226     TestOK = TestOK && check_dump(ost, "Test nullptr in CTOR of Odometer", Odometer::ERROR_NULLPTR, error_msg);
227     error_msg.clear();
228
229
230     try {
231         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
232         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
233         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
234         AudiA3.reset(); // <-- Free Car
235         OdoMeter->Update(); // <-- throws exception Car not set
236     }
237     catch (const string& err) {
238         error_msg = err;
239     }
240     catch (bad_alloc const& error) {
241         error_msg = error.what();
242     }
243     catch (const exception& err) {
244         error_msg = err.what();
245     }
246     catch (...) {
247         error_msg = "Unhandelt Exception";
248     }
249
250     TestOK = TestOK && check_dump(ost, "Test Car nullptr in Update of Odometer", Odometer::ERROR_NULLPTR, error_msg);
251     error_msg.clear();
252
253     try {
254         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data.txt");
255         Car::Sptr AudiA3 = make_shared<Car>(600, sen);
256         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
257
258         OdoMeter->SetDisplay(nullptr);
259     }
260     catch (const string& err) {
261         error_msg = err;
262     }
263     catch (bad_alloc const& error) {
264         error_msg = error.what();
265     }
266     catch (const exception& err) {
267         error_msg = err.what();
268     }
269     catch (...) {
270         error_msg = "Unhandelt Exception";
271     }
272
273     TestOK = TestOK && check_dump(ost, "Test nullptr in Set Display", Odometer::ERROR_NULLPTR, error_msg);
274     error_msg.clear();
275
276     try {
277         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
278         Car::Sptr AudiA3 = make_shared<Car>(1000, sen);
279         Odometer::Sptr OdoMeter = make_shared<Odometer>(AudiA3);
280         AudiA3->Attach(OdoMeter);
281
282         TestOK = TestOK && check_dump(ost, "Test initial Milage of Odomerter", static_cast<size_t>(0), OdoMeter->GetMilage());
283
284         AudiA3->Process();
285
286         TestOK = TestOK && check_dump(ost, "Test Milage after one Process of Odomerter", static_cast<size_t>(26), OdoMeter->GetMilage()
287
288     }
289     catch (const string& err) {
290         error_msg = err;
291     }
292     catch (bad_alloc const& error) {
293         error_msg = error.what();
294     }
295     catch (const exception& err) {
296         error_msg = err.what();
297     }
```

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

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

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

```
523     }
524
525     // check if exception was thrown
526     TestOK = TestOK && check_dump(ost, "Test␣invalid␣RPM␣Data␣(-1000)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
527     error_msg.clear();
528
529     try {
530         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid3.txt");
531         size_t revs = sen->GetRevolutions();
532     }
533     catch (const string& err) {
534         error_msg = err;
535     }
536     catch (bad_alloc const& error) {
537         error_msg = error.what();
538     }
539     catch (const exception& err) {
540         error_msg = err.what();
541     }
542     catch (...) {
543         error_msg = "Unhandelt␣Exception";
544     }
545
546     // check if exception was thrown
547     TestOK = TestOK && check_dump(ost, "Test␣invalid␣RPM␣Data␣(1007ab)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
548     error_msg.clear();
549
550
551     try {
552         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_invalid4.txt");
553         size_t revs = sen->GetRevolutions();
554     }
555     catch (const string& err) {
556         error_msg = err;
557     }
558     catch (bad_alloc const& error) {
559         error_msg = error.what();
560     }
561     catch (const exception& err) {
562         error_msg = err.what();
563     }
564     catch (...) {
565         error_msg = "Unhandelt␣Exception";
566     }
567
568     // check if exception was thrown
569     TestOK = TestOK && check_dump(ost, "Test␣invalid␣RPM␣Data␣(10.00)", RPM_Sensor::ERROR_SENSOR_INVALID_DATA_INPUT, error_msg);
570     error_msg.clear();
571
572     // test file not found
573     try {
574         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("file_not_found.txt");
575         size_t revs = sen->GetRevolutions();
576     }
577     catch (const string& err) {
578         error_msg = err;
579     }
580     catch (bad_alloc const& error) {
581         error_msg = error.what();
582     }
583     catch (const exception& err) {
584         error_msg = err.what();
585     }
586     catch (...) {
587         error_msg = "Unhandelt␣Exception";
588     }
589
590     TestOK = TestOK && check_dump(ost, "Test␣file␣not␣found␣in␣RPM_Sensor", RPM_Sensor::ERROR_SENSOR_FILE_NOT_FOUND, error_msg);
591     error_msg.clear();
592
593     // check empty file
594     try {
595         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_data_empty.txt");
596         size_t revs = sen->GetRevolutions();
597     }
```

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

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

```
748         error_msg = "Unhandelt␣Exception";
749     }
750 
751     TestOK = TestOK && check_dump(ost, "Test␣Exception␣in␣TestCase", true, error_msg.empty());
752     error_msg.clear();
753 
754 
755     try {
756         RPM_Sensor::Sptr sen = make_shared<RPM_Sensor>("rpm_test_data.txt");
757         Car::Sptr c = make_shared<Car>(100,sen );
758         while(1) c->Process();
759     }
760     catch (const string& err) {
761         error_msg = err;
762     }
763     catch (bad_alloc const& error) {
764         error_msg = error.what();
765     }
766     catch (const exception& err) {
767         error_msg = err.what();
768     }
769     catch (...) {
770         error_msg = "Unhandelt␣Exception";
771     }
772 
773     TestOK = TestOK && check_dump(ost, "Test␣Exception␣End␣of␣File␣in␣Car␣Process",RPM_Sensor::ERROR_SENSOR_EOF, error_msg);
774     error_msg.clear();
775 
776     ost << TestEnd;
777 
778     return TestOK;
779 }
```

## 6.16 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
```