FH-OÖ Hagenberg/HSD
SDP3, WS 2025
Übung 1

Name: Simon Offenberger / Simon Vogelhuber	Aufwand in h: siehe Doku.
Mat.Nr: S2410306027 / S2410306014	Punkte:
Übungsgruppe: 1	korrigiert:

Beispiel1: Fuhrpark (24 Punkte)

Ein Fuhrpark soll verschiedene Fahrzeuge verwalten: PKWs, LKWs und Motorräder. Entwerfen Sie dazu ein geeignetes Klassendiagramm (Klassenhierarchie) und ordnen Sie folgende Eigenschaften den einzelnen Klassen zu: Automarke, Kennzeichen und die Kraftstoffart (Benzin, Diesel, elektrisch oder Gas). Weiters muss jedes Fahrzeug ein Fahrtenbuch führen. Ein Eintrag im Fahrtenbuch speichert das Datum und die Anzahl der gefahrenen Kilometer an diesem Tag.

Geben Sie Set- und Get-Methoden nur dann an, wenn sie sinnvoll sind!

Die Fahrzeuge stellen zur Ausgabe eine `Print`-Methode zur Verfügung!

Ein Fuhrpark soll folgende Aufgaben erledigen können:

1. Hinzufügen von neuen Fahrzeugen.
2. Entfernen von bestehenden Fahrzeugen.
3. Suchen eines Fahrzeuges nach seinem Kennzeichen.
4. Ausgeben aller Fahrzeuge samt ihrer Eigenschaften und dem Fahrtenbuch auf dem Ausgabestrom und in einer Datei.
5. Verwenden Sie im Fuhrpark zur Verwaltung aller Fahrzeuge einen entsprechenden Container!
6. Der Fuhrpark muss kopierbar und zweisbar sein!

Die Ausgabe soll folgendermaßen aussehen:

```
Fahrzeugart: Motorrad
Marke:       Honda CBR
Kennzeichen: FR-45AU
```

```
04.04.2018:  52 km
05.06.2018:   5 km

Fahrzeugart: PKW
Marke:       Opel Astra
Kennzeichen: LL-345UI
04.07.2018:  51 km
05.07.2018:  45 km

Fahrzeugart: LKW
Marke:       Scania 1100
Kennzeichen: PE-34MU
04.08.2018:  512 km
05.08.2018:   45 km
07.08.2018:  678 km
14.08.2018:  321 km
```

Die Fahrzeugart wird nicht als Attribut gespeichert, sondern bei der Ausgabe direkt ausgegeben! Für den Fuhrpark ist der Ausgabeoperator zu überschreiben.

Für jedes Fahrzeug soll die Summe der gefahrenen Kilometer ermittelt werden können und der Fuhrpark soll die Summe der gefahrenen Kilometer aller seiner Fahrzeuge liefern. Verwenden Sie dazu entsprechende Algorithmen.

Die folgenden Punkte gelten auch für alle nachfolgenden Übungen:

1. Werfen Sie wo nötig Exceptions und geben Sie Fehlermeldungen aus!
2. Implementieren Sie einen ausführlichen Testtreiber und geben sie entsprechende Meldungen für die Testprotokollierung aus.
3. Verfassen Sie weiters eine Systemdokumentation mit folgendem Inhalt:
    - Verteilung der Aufgaben auf die Teammitglieder.
    - Anforderungsdefinition mit eventuell zusätzlich getroffenen Annahmen. Treffen Sie für alle unzureichenden Angaben sinnvolle Annahmen und begründen Sie diese.
    - Systementwurf in Form eines Klassendiagrammes mit allen Klassen und deren Beziehungen, inklusive der wichtigsten Attribute und Methoden. Geben Sie zusätzlich in den entsprechenden Header-Dateien den Verfasser an! Das Klassendiagramm muss nicht vollständig dem implementierten Sourcecode entsprechen! Geben Sie weiters Ihre Designentscheidungen an und begründen sie diese!
    - Testausgaben: die Ausgaben sollen aussagekräftig sein, damit aus der Ausgabe erkennbar ist, was getestet wurde.

    - Vollständig dokumentierter Sourcecode (Korrektur der Tutoren). Verwenden Sie Doxygen-Kommentare.

4. Die einzelnen Klassen (Komponenten) werden direkt im Quellcode dokumentiert und mit Hilfe von Doxygen eine HTML-Doku generiert.

5. Führen Sie zusammen mit Ihrer Teamkollegin bzw. mit Ihrem Teamkollegen vor der Realisierung eine Aufwandsschätzung in (Ph) durch und notieren Sie die geschätzte Zeitdauer am Deckblatt.

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!

FH OBERÖSTERREICH HSD
FH-HAGENBERG

Systemdokumentation Projekt Fuhrpark

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 20. Oktober 2025

Inhaltsverzeichnis

1 Organisatorisches

1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@students.fh-hagenberg.at
- Simon Vogelhuber, Matr.-Nr.: S2410306014, E-Mail: s2410306014@students.fh-hagenberg.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
  - Design Klassendiagramm
  - Implementierung und Test der Klassen:
    - Object,
    - RecordEntry,
    - DriveRecord,
    - Vehicle,
  - Implementierung des Testtreibers
  - Dokumentation
- Simon Vogelhuber
  - Design Klassendiagramm
  - Implementierung und Komponententest der Klassen:
    - Garage

        * Car
        * Bike
        * Truck
    - Implementierung des Testtreibers
    - Dokumentation

1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich 9 Ph
- Simon Vogelhuber: geschätzt 10 Ph / tatsächlich 7,5 Ph

2 Anforderungsdefinition (Systemspezifikation)

In diesem System werden Fahrzeuge in einem Fuhrpark verwaltet. Zusätzlich soll auch noch ein Fahrtenbuch zu jedem Fahrzeug gespeichert werden.

Funktionen des Fahrtenbuches

- Berechnen des Kilometerstands der aufgezeichneten Fahrten.
- Speichere Datum und Distanz einer Fahrt.

Funktionen des Fuhrparks

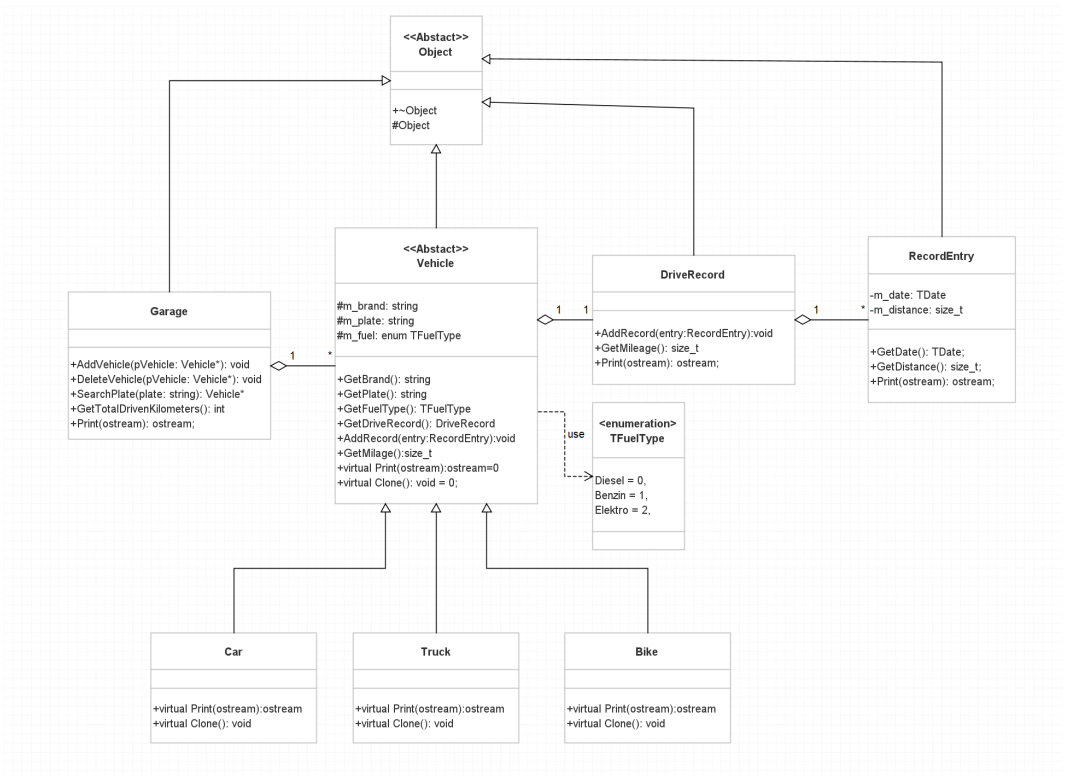
- Verwalten von verschiedenen Fahrzeugarten (Auto, LKW, Motorrad,...).
- Hinzufügen und löschen eines Fahrzeuges
- Ausgabe aller Fahrzeugdaten inklusive der Fahrtenbucheinträge.
- Suchen nach einem Fahrzeug mit dessen Kennzeichen.
- Berechnung der Gesamtkilometer aller Fahrzeuge im Fuhrpark.
- Der Fuhrpark muss kopierbar und zuweisbar sein.
- Nach hinzufügen der Fahrzeuge sind diese im Besitz des Fuhrparks, dieser ist dadurch auch für das Löschen verantwortlich.

Funktionen der Fahrzeuge

- Bereitstellen einer Print Funktion mit Info über das Fahrzeug und die Fahrtenbucheinträge.
- Hinzufügen von Fahrtenbucheinträgen.
- Ermittlung vom Kilometerstand eines Fahrzeugs.
- Speichern von Hersteller, Treibstoff und Kennzeichen des Fahrzeugs

3 Systementwurf

3.1 Klassendiagramm



<<Abstact>>
Object
+~Object
#Object
<<Abstact>>
Vehicle
#m_brand: string
#m_plate: string
#m_fuel: enum TFuelType
+GetBrand(): string
+GetPlate(): string
+GetFuelType(): TFuelType
+GetDriveRecord(): DriveRecord
+AddRecord(entry:RecordEntry):void
+GetMilage():size_t
+virtual Print(ostream):ostream=0
+virtual Clone(): void = 0;
Garage
+AddVehicle(pVehicle: Vehicle*): void
+DeleteVehicle(pVehicle: Vehicle*): void
+SearchPlate(plate: string): Vehicle*
+GetTotalDrivenKilometers(): int
+Print(ostream): ostream;
1
*
DriveRecord
+AddRecord(entry:RecordEntry):void
+GetMileage(): size_t
+Print(ostream): ostream;
1
1
RecordEntry
-m_date: TDate
-m_distance: size_t
+GetDate(): TDate;
+GetDistance(): size_t;
+Print(ostream): ostream;
1
*
use
<enumeration>
TFuelType
Diesel = 0,
Benzin = 1,
Elektro = 2,
Car
+virtual Print(ostream):ostream
+virtual Clone(): void
Truck
+virtual Print(ostream):ostream
+virtual Clone(): void
Bike
+virtual Print(ostream):ostream
+virtual Clone(): void

CTOR und DTOR nur im Object eingetragen ,da dies implementierungsspezifische Angaben sind und nicht im UML Standard definiert sind.

3.2 Designentscheidungen

Im Klassendiagramm wurde der Polymorphismus angewendet, um unterschiedliche Fahrzeugarten mit der gemeinsamen Schnittstelle 'Vehicle' anzusprechen. Die Klasse **Garage** speichert einen Container mit der abstrakte Basisklasse 'Vehicle' als Elementtyp und kann somit alle bestehenden und auch neuen Fahrzeugarten verwalten, die sich von der gemeinsamen Basisklasse 'Vehicle' ableiten. Für die Aufzeichnung eines Fahrtenbuches wurde die Klasse **DriveRecord** implementiert. Diese Klasse speichert mehrere Objekte der Klasse **RecordEntry**. Die Record Entries werden im Fahrtenbuch in einem **Multiset** gespeichert, damit sind die Einträge ins Fahrtenbuch immer nach dem Datum aufsteigend sortiert.
Aus diesem Grund wurde der **operator<** für die Record Entries definiert. Dieser vergleicht das Datum der Einträge. Dadurch, dass die Einträge ins Fahrtenbuch als eigene Klasse implementiert wurde, lassen sich die einzelnen Einträge schnell und einfach erweitern.

Als Container für die Speicherung der Fahrzeuge in der Klasse **Garage** wurde der Vektor verwendet. Dieser erlaubt es schnell Fahrzeuge hinzuzufügen, und das Suchen geschieht relativ schnell in O(n). Einzig und allein, das Löschen aus der Mitte des Vektors stellt bei größerwerdenden Fuhrparks ein Problem dar. Wenn dies in der Verwendung des Fuhrparks öfters passiert sollte der verwendete Container ausgetauscht werden.

Die Klassen **Car, Truck und Bike** wurden für die Konkretisierung der Printfunktion verwendet. Diese Klassen lassen sich schnell und einfach erweitern, und können trotzdem weiter vom Fuhrpark verwaltet werden.

4 Dokumentation der Komponenten (Klassen)

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

5 Testprotokollierung

```

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

Test RecordEntry Get Date
[Test OK] Result: (Expected: 2025-10-13 == Result: 2025-10-13)

Test RecordEntry Get Distance
[Test OK] Result: (Expected: 150 == Result: 150)

Test RecordEntry Print
[Test OK] Result: (Expected: true == Result: true)

Test RecordEntry Exception Bad Ostream
[Test OK] Result: (Expected: ERROR: Provided Ostream is bad == Result:
    ↪ ERROR: Provided Ostream is bad)

Test RecordEntry less than operator
[Test OK] Result: (Expected: true == Result: true)

Test RecordEntry Exceotion Distance = 0
[Test OK] Result: (Expected: ERROR: Distance cannot be zero! == Result:
    ↪ ERROR: Distance cannot be zero!)


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


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

Test DriveRecord Print Sorted and Add Record
[Test OK] Result: (Expected: true == Result: true)

Test DriveRecord Get Milage
[Test OK] Result: (Expected: 450 == Result: 450)

Test DriveRecord Exception Bad Ostream
[Test OK] Result: (Expected: ERROR: Provided Ostream is bad == Result:
    ↪ ERROR: Provided Ostream is bad)
```

```

Test DriveRecord Empty Print
[Test OK] Result: (Expected: No Exception == Result: No Exception)


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


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

vehicle plate search
[Test OK] Result: (Expected: 000001CABA4C2410 == Result: 000001CABA4C2410)

Test garage plate search - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test garage plate search invalid plate
[Test OK] Result: (Expected: 0000000000000000 == Result: 0000000000000000)

Test garage plate search invalid plate - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Garage Print
[Test OK] Result: (Expected:
Fahrzeugart:  PKW
Marke:        UAZ
Kennzeichen:  SR770BA
13.10.2025:    25 km
 == Result:
Fahrzeugart:  PKW
Marke:        UAZ
Kennzeichen:  SR770BA
13.10.2025:    25 km
)

Test garage print - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test garage print empty garage
[Test OK] Result: (Expected: true == Result: true)

Test garage print empty garage - error buffer
```

```
[Test OK] Result: (Expected: true == Result: true)

Test Delete Vehicle
[Test OK] Result: (Expected: 0000000000000000 == Result: 0000000000000000)

Test garage print - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Delete Vehicle
[Test OK] Result: (Expected: 0000000000000000 == Result: 0000000000000000)

Test Delete Vehicle - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test GetTotalDrivenKilometers()
[Test OK] Result: (Expected: 118 == Result: 118)

Test GetTotalDrivenKilometers() - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test ostream operator
[Test OK] Result: (Expected:
Fahrzeugart:  PKW
Marke:        Madza
Kennzeichen:  WD40AHAH
13.10.2025:    25 km
28.10.2025:    34 km
 == Result:
Fahrzeugart:  PKW
Marke:        Madza
Kennzeichen:  WD40AHAH
13.10.2025:    25 km
28.10.2025:    34 km
)

Test ostream operator - error buffer
[Test OK] Result: (Expected: true == Result: true)

TestAdding Car as nullptr;
[Test OK] Result: (Expected: ERROR: Passed in Nullptr! == Result: ERROR:
   ↪ Passed in Nullptr!)

TestDeleting Car as nullptr;
```

```
[Test OK] Result: (Expected: ERROR: Passed in Nullptr! == Result: ERROR:
    ↪ Passed in Nullptr!)


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


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

Test car fueltype
[Test OK] Result: (Expected: 1 == Result: 1)

Test car fueltype - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test car plate
[Test OK] Result: (Expected: SR770BA == Result: SR770BA)

Test car plate - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test car brand
[Test OK] Result: (Expected: Steyr == Result: Steyr)

Test car brand - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test car milage
[Test OK] Result: (Expected: 25 == Result: 25)

Test car milage - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test car driveRecord
[Test OK] Result: (Expected: 13.10.2025:    25 km
 == Result: 13.10.2025:    25 km
)

Test car driveRecord - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Car CTOR empty brand
```

```
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
    ↪ ERROR: Passed in empty string!)

Test Car CTOR empty plate
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
    ↪ ERROR: Passed in empty string!)


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


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

Test Bike fueltype
[Test OK] Result: (Expected: 1 == Result: 1)

Test Bike fueltype - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Bike plate
[Test OK] Result: (Expected: SR770BA == Result: SR770BA)

Test Bike plate - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Bike brand
[Test OK] Result: (Expected: Steyr == Result: Steyr)

Test Bike brand - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Bike milage
[Test OK] Result: (Expected: 25 == Result: 25)

Test Bike milage - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Bike driveRecord
[Test OK] Result: (Expected: 13.10.2025:    25 km
 == Result: 13.10.2025:    25 km
)

```

```
Test Bike driveRecord - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Bike CTOR empty brand
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
    ↪ ERROR: Passed in empty string!)

Test Bike CTOR empty plate
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
    ↪ ERROR: Passed in empty string!)


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


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

Test Truck fueltype
[Test OK] Result: (Expected: 1 == Result: 1)

Test Truck fueltype - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Truck plate
[Test OK] Result: (Expected: SR770BA == Result: SR770BA)

Test Truck plate - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Truck brand
[Test OK] Result: (Expected: Steyr == Result: Steyr)

Test car brand - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Truck milage
[Test OK] Result: (Expected: 50 == Result: 50)

Test Truck milage - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test truck driveRecord
```

```
[Test OK] Result: (Expected: 13.10.2025:    25 km
 == Result: 13.10.2025:    25 km
)

Test truck driveRecord - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test truck CTOR empty brand
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
   ↪ ERROR: Passed in empty string!)

Test truck CTOR empty plate
[Test OK] Result: (Expected: ERROR: Passed in empty string! == Result:
   ↪ ERROR: Passed in empty string!)


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

TEST OK!!
```

6 Quellcode

6.1 Object.hpp

```
/*******************************************************************//**
 * \file   Object.hpp
 * \brief  Root of all Objects
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef OBJECT_HPP
#define OBJECT_HPP

#include <iostream>

class Object {
public:
   /**
   * Defintions of the Exceptionmessages
   */
   inline static const std::string ERROR_BAD_OSTREAM = "ERROR:␣Provided␣Ostream␣is␣bad";
   inline static const std::string ERROR_FAIL_WRITE = "ERROR:␣Fail␣to␣write␣on␣provided␣Ostream";

   /**
    * Virtual DTOR, once virtual always virtual.
    */
   virtual ~Object() = default;

protected:
   /**
    * \brief protected CTOR -> abstract.
    */
   Object() = default;
};

#endif // !1
```

6.2 RecordEntry.hpp

```cpp
/*****************************************************************//**
 * \file   RecordEntry.hpp
 * \brief  Class that defines an entry in a dirve record.
 * \brief  This record entry is used by the drive record class.
 * \brief  The drive record class stores multiple record entries.
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef RECORD_ENTRY_HPP
#define RECORD_ENTRY_HPP


#include <chrono>
#include "Object.hpp"

// Using Statement for date type
using TDate = std::chrono::year_month_day;

class RecordEntry : public Object {
public:

   /**
   * Defintions of the Exceptionmessages
   */
   inline static const std::string ERROR_DISTANCE_ZERO = "ERROR:␣Distance␣cannot␣be␣zero!";

   /**
    * \brief CTOR of a drive record.
    *
    * \param date : date when the drive happend
    * \param distance : the distance of the drive in km
    */
   RecordEntry(const TDate& date, const size_t& distance);

   /**
    * \brief Getter of the distance member of the Record Entry Class.
    *
    * \return Distance of this Record Entry
    */
   size_t GetDistance() const;

   /**
    * \brief Getter of the data member of the Record Entry Class.
    *
    * \return Date of this Record Entry
    */
   TDate GetDate() const;

   /**
    * \brief Formatted output of this Record Entry on an ostream.
    *
    * \param ost : Refernce to an ostream where the Entry should be printed at.
    * \return Referenced ostream
     * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const;

   /**
    * \brief less than operater, is used for storing the Entries in a multiset.
    *
    * \param rh : Righthandside of the less than operator
    * \return true:  left hand side is less than the right hand side.
    * \return false: left hand side is greather or equal than the right hand side.
    */
   bool operator<(const RecordEntry& rh) const;

private:
   TDate m_date;     // private date member
   size_t m_distance;   // private distance member
};
```

```


#endif
```

6.3 RecordEntry.cpp

```cpp
/******************************************************************//**
 * \file   RecordEntry.cpp
 * \brief  Implementation of a Record Entry
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#include "RecordEntry.hpp"

using namespace std;

RecordEntry::RecordEntry(const TDate& date, const size_t& distance) : m_date{date}
{
   if (distance == 0) throw RecordEntry::ERROR_DISTANCE_ZERO;
   m_distance = distance;
}


size_t RecordEntry::GetDistance() const
{
   return m_distance;
}


TDate RecordEntry::GetDate() const
{
   return m_date;
}


std::ostream& RecordEntry::Print(std::ostream& ost) const
{
   if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

   ost << std::setfill('0')<< right << std::setw(2) << m_date.day() << "."
      << std::setw(2) << static_cast<unsigned>(m_date.month()) << "."
      << std::setw(4) << m_date.year() << ":" << std::setfill(' ')
      << std::setw(6) << m_distance << " km\n";

   if (ost.fail()) throw Object::ERROR_FAIL_WRITE;

   return ost;
}


bool RecordEntry::operator<(const RecordEntry& rh) const
{
   return m_date < rh.m_date;
}
```

6.4 DriveRecord.hpp

```
/*********************************************************************//**
 * \file   DriveRecord.hpp
 * \brief  This Class implements a drive record book which holds multiple
 * \brief  record entries in a TCont, which is defined as a multiset.
 * \brief  The multiset is used because it stores the data sorted.
 * \brief  This sorting mandatory because the entries should be date ascending.
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef DRIVE_RECORD_HPP
#define DRIVE_RECORD_HPP

#include <set>
#include "RecordEntry.hpp"
#include "Object.hpp"

// Using statement for the used container to store the record entries
using TCont = std::multiset<RecordEntry>;

class DriveRecord : public Object {
public:

   /**
    * \brief Methode for adding a record entry to a collection of drive records.
    *
    * \param entry : Record to be added to the colletion
    */
   void AddRecord(const RecordEntry & entry);

   /**
    * \brief This methode adds up all the distance of all record entries.
    *
    * \return the sum of all distances in the collection
    */
   size_t GetMilage() const;

   /**
    * \brief Formatted output of all Record Entry on an ostream.
    *
    * \param ost : Refernce to an ostream where the Entries should be printed at.
    * \return Referenced ostream
    * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const;

private:

   TCont m_driveRecords;
};


#endif // !1
```

6.5 DriveRecord.cpp

```cpp
/*******************************************************************//**
 * \file   DriveRecord.cpp
 * \brief  Implementation of a Drive Record
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#include <numeric>
#include <algorithm>
#include "DriveRecord.hpp"


void DriveRecord::AddRecord(const RecordEntry& entry)
{
   m_driveRecords.insert(entry);
}


size_t DriveRecord::GetMilage() const
{
   // use std accumulet + lambda to calc the total Milage
   return std::accumulate(m_driveRecords.cbegin(), m_driveRecords.cend(), static_cast<size_t>(0),
      [](const size_t val,const RecordEntry& entry) {return val + entry.GetDistance();});
}


std::ostream& DriveRecord::Print(std::ostream& ost) const
{
   if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

   std::for_each(m_driveRecords.cbegin(), m_driveRecords.cend(), [&](const RecordEntry& entry) {entry.Print(ost);});

   if (ost.fail()) throw Object::ERROR_FAIL_WRITE;

   return ost;
}
```

6.6 Garage.hpp

```
/*********************************************************************//**
 * \file   Garage.hpp
 * \brief  This Class implements a polymorph container containing
 * \brief  all derivatives of the 'Vehicle' Class.
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#ifndef GARAGE_HPP
#define GARAGE_HPP

#include <vector>
#include <string>
#include "Object.hpp"
#include "Vehicle.hpp"

// Using Statement for the used Container to store the Vehicles
using TGarageCont = std::vector<Vehicle const *>;

class Garage : public Object {
public:

    /**
    * Defintions of the Exceptionmessages
    */
    inline static const std::string ERROR_NULLPTR= "ERROR:␣Passed␣in␣Nullptr!";

    /**
     * \brief Default CTOR.
     *
     */
    Garage() = default;

    /**
     * \brief Adds a vehicle to a vehicle collection.
     * \brief A specific vehicle is passed in and casted to a vehicle Pointer.
     * \brief This is allowed because Car,Truck and Bike are derived from Vehicle.
     * \brief A car is a Vehicle.
     * \brief This casted Pointer is copied ito this methode and added to the collection
     * \param newVehicle : Pointer to a Vehicle.
     */
    void AddVehicle(Vehicle const * const newVehicle);

    /**
     * \brief deletes Vehicle inside garage from provided pointer.
     * \param pVehicle : Pointer to a Vehicle.
     */
    void DeleteVehicle(Vehicle const * const pVehicle);

    /**
     * \brief Functions searches for vehicle with matching plate.
     *
     * \return pointer to the vehicle inside the garage
     */
    Vehicle const * const SearchPlate(const std::string & plate) const;

    /**
     * \brief Formatted of every car and its drive record
     * \param ost : Refernce to an ostream where the Entry should be printed at.
     * \return Referenced ostream
     * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
     */
   std::ostream& Print(std::ostream& ost = std::cout) const;

    /**
     * \brief Calculates sum of every kilometer every vehicle has driven
     * \brief in total
     * \return size_t total kilometers
     */
    size_t GetTotalDrivenKilometers() const;

    /**
```

```
     * \brief Copy CTOR of Garage. Is Needed because Garage
     * \brief owns all the Vehicle Objects that are allocated on the heap.
     *
     * \param garage Garage that should be copied
     */
    Garage(const Garage& garage);

    /**
     * \brief Assign Operator for a Object of Garage.
     *
     * \param garage Garage of the right hand side of the assignment.
     */
    void operator=(Garage garage);

    /**
     * \brief DTOR of a Garage obj.
     * \brief Frees all the allocated Memory
     *
     */
    ~Garage();

private:
    TGarageCont m_vehicles;
};

/**
 * \brief Override for output operator
 * \return ostream
 */
std::ostream& operator <<(std::ostream& ost, Garage& garage);

#endif
```

6.7 Garage.cpp

```cpp
/*****************************************************************//**
 * \file   Garage.cpp
 * \brief  Implementation of Garage.h
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#include "Garage.hpp"
#include <algorithm>
#include <numeric>

void Garage::AddVehicle(Vehicle const * const newVehicle)
{
    if (newVehicle == nullptr) throw ERROR_NULLPTR;
    // Add the new vehicle to the collection.
    m_vehicles.push_back(newVehicle);
}

/**
 * \brief deletes Vehicle inside garage from provided pointer.
 * \param pVehicle : Pointer to a Vehicle.
 */
void Garage::DeleteVehicle(Vehicle const * const pVehicle)
{
    if (pVehicle == nullptr) throw ERROR_NULLPTR;

    // if pVehicle is inside m_Vehicles -> erase and free
    auto itr = std::find(m_vehicles.begin(), m_vehicles.end(), pVehicle);
    if (itr != m_vehicles.end())
    {
        m_vehicles.erase(itr);
        delete pVehicle;
    }
}

const Vehicle* const Garage::SearchPlate(const std::string & plate) const
{
    for (const auto &elem : m_vehicles)
    {
        if (elem->GetPlate() == plate)
        {
            return elem;
        }
    }

    return nullptr;
}

std::ostream& Garage::Print(std::ostream& ost) const
{
    if (!ost.good())
        throw Object::ERROR_BAD_OSTREAM;

    for (auto& elem : m_vehicles)
    {
        elem->Print(ost);
    }

    if (ost.fail())
        throw Object::ERROR_FAIL_WRITE;

    return ost;
}

size_t Garage::GetTotalDrivenKilometers() const
{
    size_t sum = std::accumulate(m_vehicles.cbegin(), m_vehicles.cend(), static_cast<size_t>(0),
        [](auto last_val, auto vehicle) {
            return last_val + vehicle->GetMilage();
        });
    return sum;
}

```

```
Garage::Garage(const Garage& garage)
{
    for_each(
        garage.m_vehicles.cbegin(), garage.m_vehicles.cend(),
        [&](auto v) {AddVehicle(v->Clone());
        });
}

void Garage::operator=(Garage garage)
{
    std::swap(m_vehicles, garage.m_vehicles);
}

Garage::~Garage()
{
    for (auto elem : m_vehicles)
    {
        delete elem;
    }

    m_vehicles.clear();
}

std::ostream& operator<<(std::ostream& ost, Garage& garage)
{
    garage.Print(ost);
    return ost;
}
```

6.8 TFuel.hpp

```
/******************************************************************//**
 * \file   TFuel.hpp
 * \brief  This Enum provides a specifier for the fuel type
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef TFUEL_HPP
#define TFUEL_HPP

// Enumeration for a fuel type
enum TFuel {
   Diesel = 0,
   Benzin = 1,
   Elektro = 2,
};

#endif // !1
```

6.9 Vehicle.hpp

```
/*******************************************************************//**
 * \file   Vehicle.hpp
 * \brief  This class imlements an abstract vehicle which is used in the
 * \brief  Garage class. It implements all the core featues of a vehicle
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef VEHICLE_HPP
#define VEHICLE_HPP

#include "Object.hpp"
#include "DriveRecord.hpp"
#include "TFuel.hpp"

class Vehicle: public Object {
public:

   /**
   * Defintions of the Exceptionmessages
   */
   inline static const std::string ERROR_EMPTY_STRING = "ERROR:␣Passed␣in␣empty␣string!";

   /**
    * \brief Getter for the brand member.
    *
    * \return string with the brand name
    */
   std::string GetBrand() const;

   /**
    * \brief Getter for the plate member.
    *
    * \return string with the plate name
    */
   std::string GetPlate() const;

   /**
    * \brief Getter for the fuel member.
    *
    * \return TFuel with the specified fuel type
    */
   TFuel GetFuelType() const;

   /**
    * \brief Getter for the drive record.
    *
    * \return const refernce to the drive record
    */
   const DriveRecord & GetDriveRecord() const;

   /**
    * \brief Methode for adding a record entry to the drive record collection.
    *
    * \param entry : Entry which should be added to the drive recod
    */
   void AddRecord(const RecordEntry& entry);

   /**
    * \brief Getter for the total milage of a vehicle.
    *
    * \return Total milage of a vehicle
    */
   size_t GetMilage() const;

   /**
    * @brief Creates a clone of the vehicle.
    *
    * \return a excat replicate of a vehicle
    */
   virtual Vehicle const* Clone() const = 0;


   /**
    * \brief Print function that is implementet by dirved Classes.
    *
    * \param ost Reference to an ostream where the Result should be printed at
    * \return referenced ostream
     * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const = 0;


protected:

   /**
    * \brief protected CTOR of a vehicle.
    * \brief protected because it is a abstract class
    *
    * \param plate : string that represents the plate of the vehicle
    * \param brand : string that represents the brand of the vehicle
    * \param fuelType : Fuel type of the vehicle
    */
   Vehicle(const std::string& brand, const TFuel& fuelType, const std::string& plate);

private:
   std::string m_brand;
   std::string m_plate;
   TFuel m_fuel;
   DriveRecord m_record;
};


#endif // !1
```

6.10 Vehicle.cpp

```
/*******************************************************************//**
 * \file   Vehicle.cpp
 * \brief  Implementation of the abstract vehicle class
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#include "Vehicle.hpp"

/**
* \brief Getter for the brand member.
*
* \return string with the brand name
*/
std::string Vehicle::GetBrand() const
{
    return m_brand;
}

/**
* \brief Getter for the plate member.
*
* \return string with the plate name
*/
std::string Vehicle::GetPlate() const
{
    return m_plate;
}

/**
* \brief Getter for the fuel member.
*
* \return TFuel with the specified fuel type
*/
TFuel Vehicle::GetFuelType() const
{
    return m_fuel;
}

/**
* \brief Getter for the drive record.
*
* \return const refernce to the drive record
*/
const DriveRecord & Vehicle::GetDriveRecord() const
{
    return m_record;
}

/**
* \brief Methode for adding a record entry to the drive record collection.
*
* \param entry : Entry which should be added to the drive recod
*/
void Vehicle::AddRecord(const RecordEntry& entry)
{
    m_record.AddRecord(entry);
}

/**
* \brief Getter for the total milage of a vehicle.
*
* \return Total milage of a vehicle
*/
size_t Vehicle::GetMilage() const
{
    return m_record.GetMilage();
}

Vehicle::Vehicle(const std::string& brand, const TFuel& fuelType, const std::string& plate) : m_fuel{fuelType}
{
    if (brand.empty() || plate.empty()) throw ERROR_EMPTY_STRING;

    m_brand = brand;
    m_plate = plate;

}
```

6.11 Car.hpp

```
/*******************************************************************//**
 * \file   Car.hpp
 * \brief  Header fo the specific Class Car
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#ifndef CAR_HPP
#define CAR_HPP

#include "Vehicle.hpp"

class Car : public Vehicle {
public:

   /**
    * \brief CTOR of a CAR -> calles the Base Class vehicle CTOR.
    *
    * \param brand string that identifies the brand.
    * \param fuelType Fueltype of the Car
    * \param plate string that identifies the plate.
    * \throw ERROR_EMPTY_STRING
    */
   Car(const std::string & brand,const TFuel & fuelType, const std::string & plate) : Vehicle(brand, fuelType,plate) {}

   /**
    * \brief Function that print all the vehicle specific info with the drive record.
    *
    * \param ost where the data should be printed at
    * \return referenced ostream
    * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const override;

   /**
    * @brief Creates a clone of the vehicle.
    *
    * \return a excat replicate of a vehicle
    */
   virtual Vehicle const* Clone() const;

private:
};


#endif // !1
```

6.12 Car.cpp

```
/*******************************************************************//**
 * \file   Car.cpp
 * \brief  Implemetation of a Car
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#include "Car.hpp"

using namespace std;


std::ostream& Car::Print(std::ostream& ost) const
{
   if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

   ost <<endl<< left << setw(14) << "Fahrzeugart:" << "PKW" << endl;
   ost << left << setw(14) << "Marke:" << GetBrand() << endl;
   ost << left << setw(14) << "Kennzeichen:" << GetPlate() << endl;
   GetDriveRecord().Print(ost);

   if (ost.fail()) throw Object::ERROR_FAIL_WRITE;

   return ost;
}


Vehicle const* Car::Clone() const
{
   return new Car(*this);
}
```

6.13 Truck.hpp

```
/******************************************************************//**
 * \file   Truck.hpp
 * \brief  Header fo the specific Class Truck
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#ifndef TRUCK_HPP
#define TRUCK_HPP

#include "Vehicle.hpp"

class Truck : public Vehicle {
public:

   /**
    * \brief CTOR of a Truck -> calles the Base Class vehicle CTOR.
    *
    * \param brand string that identifies the brand.
    * \param fuelType Fueltype of the Truck
    * \param plate string that identifies the plate.
    * \throw ERROR_EMPTY_STRING
    */
   Truck(const std::string& brand, const TFuel& fuelType, const std::string& plate) : Vehicle(brand, fuelType, plate) {}

   /**
    * \brief Function that print all the vehicle specific info with the drive record.
    *
    * \param ost where the data should be printed at
    * \return referenced ostream
    * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const override;

   /**
    * @brief Creates a clone of the vehicle.
    *
    * \return a excat replicate of a vehicle
    */
   virtual Vehicle const* Clone() const;

private:
};


#endif
```

6.14 Truck.cpp

```
/*****************************************************************//**
 * \file   Truck.cpp
 * \brief  Implementation of a Truck
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#include "Truck.hpp"

using namespace std;


std::ostream& Truck::Print(std::ostream& ost) const
{
   if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

   ost << endl << left << setw(14) << "Fahrzeugart:" << "LKW" << endl;
   ost << left << setw(14) << "Marke:" << GetBrand() << endl;
   ost << left << setw(14) << "Kennzeichen:" << GetPlate() << endl;
   GetDriveRecord().Print(ost);

   if (ost.fail()) throw Object::ERROR_FAIL_WRITE;

   return ost;
}


Vehicle const* Truck::Clone() const
{
   return new Truck(*this);
}
```

6.15 Bike.hpp

```
/*****************************************************************//**
 * \file   Bike.hpp
 * \brief  Header fo the specific Class Bike
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#ifndef BIKE_HPP
#define BIKE_HPP

#include "Vehicle.hpp"

class Bike : public Vehicle {
public:

   /**
    * \brief CTOR of a Bike -> calles the Base Class vehicle CTOR.
    *
    * \param brand string that identifies the brand.
    * \param fuelType Fueltype of the Bike
    * \param plate string that identifies the plate.
    * \throw ERROR_EMPTY_STRING
    */
   Bike(const std::string& brand, const TFuel& fuelType, const std::string& plate) : Vehicle(brand, fuelType, plate) {}

   /**
    * \brief Function that print all the vehicle specific info with the drive record.
    *
    * \param ost where the data should be printed at
    * \return referenced ostream
    * \throw ERROR_BAD_OSTREAM
    * \throw ERROR_WRITE_FAIL
    */
   virtual std::ostream& Print(std::ostream& ost = std::cout) const override;

   /**
    * @brief Creates a clone of the vehicle.
    *
    * \return a excat replicate of a vehicle
    */
   virtual Vehicle const* Clone() const;

private:
};


#endif
```

6.16 Bike.cpp

```cpp
/*****************************************************************//**
 * \file   Bike.cpp
 * \brief  Implementation of the Bike Class
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#include "Bike.hpp"

using namespace std;


std::ostream& Bike::Print(std::ostream& ost) const
{
   if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

   ost << endl << left << setw(14) << "Fahrzeugart:" << "Motorrad" << endl;
   ost << left << setw(14) << "Marke:" << GetBrand() << endl;
   ost << left << setw(14) << "Kennzeichen:" << GetPlate() << endl;
   GetDriveRecord().Print(ost);

   if (ost.fail()) throw Object::ERROR_FAIL_WRITE;

   return ost;
}


Vehicle const* Bike::Clone() const
{
   return new Bike(*this);
}
```

6.17 main.cpp

```
/*****************************************************************//**
 * \file   main.cpp
 * \brief  Testdriver
 *
 * \author Simon / Simon
 * \date   October 2025
 *********************************************************************/
#include <iostream>
#include <fstream>
#include <sstream>
#include <cassert>
#include "Test.hpp"
#include "RecordEntry.hpp"
#include "DriveRecord.hpp"
#include "Car.hpp"
#include "Bike.hpp"
#include "Truck.hpp"
#include "Garage.hpp"
#include "vld.h"

using namespace std;
using namespace chrono;

static bool Test_RecordEntry(ostream & ost = cout);
static bool Test_DriveRecord(ostream & ost = cout);
static bool Test_Garage(ostream & ost = cout);
static bool Test_Car(ostream & ost = cout);
static bool Test_Bike(ostream & ost = cout);
static bool Test_Truck(ostream & ost = cout);

#define WriteOutputFile true

int main(void){
   bool Test_OK = true;

   Test_OK = Test_OK && Test_RecordEntry(cout);
   Test_OK = Test_OK && Test_DriveRecord(cout);
   Test_OK = Test_OK && Test_Garage(cout);
   Test_OK = Test_OK && Test_Car(cout);
   Test_OK = Test_OK && Test_Bike(cout);
   Test_OK = Test_OK && Test_Truck(cout);

   if (Test_OK) TestCaseOK(cout);
   else TestCaseFail(cout);


   if (WriteOutputFile) {

      ofstream test_output;
      test_output.open("TestOutput.txt");

      Test_OK = Test_OK && Test_RecordEntry(test_output);
      Test_OK = Test_OK && Test_DriveRecord(test_output);
      Test_OK = Test_OK && Test_Garage(test_output);
      Test_OK = Test_OK && Test_Car(test_output);
      Test_OK = Test_OK && Test_Bike(test_output);
      Test_OK = Test_OK && Test_Truck(test_output);

      if (Test_OK) TestCaseOK(test_output);
      else TestCaseFail(test_output);

      test_output.close();
   }
}

bool Test_RecordEntry(ostream& ost)
{
   assert(ost.good());

   bool Test_OK = true;
   string error_msg;

   ost << TestStart;
   const TDate date{ 2025y,October,13d };
   const size_t distance = 150;

   RecordEntry entry1{ date, distance };

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣Get␣Date", date,entry1.GetDate());

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣Get␣Distance", distance,entry1.GetDistance());

   stringstream result;
   string expected = "13.10.2025:␣␣␣150␣km\n";
   entry1.Print(result);

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣Print", true, result.str() == expected);

   ofstream badstream;
   badstream.setstate(ios::badbit);

   try {
      RecordEntry entry{ TDate{2025y,October,13d}, 150 };
      entry.Print(badstream);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣Exception␣Bad␣Ostream", Object::ERROR_BAD_OSTREAM, error_msg);

   badstream.close();


   RecordEntry entrygreater{ {2025y,October,8d},10 };
   RecordEntry entryless{ {2025y,October,6d},6 };

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣less␣than␣operator", true, entryless<entrygreater);

   try {
      RecordEntry entry{ TDate{2025y,October,13d}, 0 };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣RecordEntry␣Exceotion␣Distance␣=␣0", RecordEntry::ERROR_DISTANCE_ZERO, error_msg);


   ost << TestEnd;

   return Test_OK;
}

bool Test_DriveRecord(ostream& ost)
{
   assert(ost.good());

   bool Test_OK = true;
   string error_msg;

   ost << TestStart;
   const TDate date{ 2025y,October,13d };
   const TDate date1{ 2025y,October,10d };
   const TDate date2{ 2025y,October,5d };
   const size_t distance = 150;
   const size_t distance1 = 150;
   const size_t distance2 = 150;

   RecordEntry entry{ date,distance };
   RecordEntry entry1{ date1,distance1 };
   RecordEntry entry2{ date2,distance2 };

   DriveRecord dRecord;

   dRecord.AddRecord(entry);
   dRecord.AddRecord(entry1);
   dRecord.AddRecord(entry2);

   stringstream result;
   stringstream expected;
   dRecord.Print(result);

   entry2.Print(expected);
   entry1.Print(expected);
   entry.Print(expected);

   Test_OK = Test_OK && check_dump(ost, "Test␣DriveRecord␣Print␣Sorted␣and␣Add␣Record", true, result.str() == expected.str());

   const size_t total_milage = 450;

   Test_OK = Test_OK && check_dump(ost, "Test␣DriveRecord␣Get␣Milage", total_milage, dRecord.GetMilage());

   ofstream badstream;
   badstream.setstate(ios::badbit);

   try {
      DriveRecord dEntry{};
      dEntry.AddRecord(entry);
      dEntry.Print(badstream);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣DriveRecord␣Exception␣Bad␣Ostream", Object::ERROR_BAD_OSTREAM, error_msg);

   const string NoExc = "No␣Exception";

   try {
      DriveRecord dEntry{};
      dEntry.Print(result);
      error_msg = NoExc;
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
```

```cpp
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣DriveRecord␣Empty␣Print", error_msg, NoExc);

   badstream.close();

   ost << TestEnd;

   return Test_OK;
}

static bool Test_Garage(ostream& ost)
{
   assert(ost.good());

   bool Test_OK = true;
   string error_msg;

   ost << TestStart;

   // Testing search plate func
   try
   {

      std::string testPlate = "SR770BA";
      Car* testCar = new Car{ "UAZ", Diesel,testPlate };
      testCar->AddRecord({ { 2025y,October,13d }, 25 });

      Garage testGarage;
      testGarage.AddVehicle(testCar);
      testGarage.AddVehicle(new Bike{"Kawasaki␣Z650RS", Benzin, "SB13KK"});
      testGarage.AddVehicle(new Truck{"Scania", Diesel, "SB132KK"});

      Test_OK = Test_OK &&
         check_dump(
            ost,
            "vehicle␣plate␣search",
            (const Vehicle*) testCar,
            testGarage.SearchPlate(testPlate)
         );
   }

   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣garage␣plate␣search␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Searching invalid plate
   try
   {

      std::string testPlate = "SR770BA";
      Car* testCar = new Car{ "UAZ", Diesel, testPlate };
      testCar->AddRecord({ { 2025y,October,13d }, 25 });

      Garage testGarage;
      testGarage.AddVehicle(testCar);

      Vehicle const* result = testGarage.SearchPlate("NOTREAL");
      Test_OK = Test_OK && check_dump(ost, "Test␣garage␣plate␣search␣invalid␣plate", result, (Vehicle const*)nullptr);

   }

   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣garage␣plate␣search␣invalid␣plate␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   try
   {
      std::string testPlate = "SR770BA";
      Car* testCar = new Car{ "UAZ", Diesel, testPlate };
      testCar->AddRecord({ { 2025y,October,13d }, 25 });

      Garage testGarage;
      testGarage.AddVehicle(testCar);

      // testing print
      std::stringstream expectation;
      std::stringstream result;

      testCar->Print(expectation);
      testGarage.Print(result);
      Test_OK = Test_OK && check_dump(ost, "Test␣Garage␣Print", expectation.str(), result.str());

   }

   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣garage␣print␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Empty Garage
   try
   {
      Garage testGarage;
      std::stringstream result;
      testGarage.Print(result);
      Test_OK = Test_OK && check_dump(ost, "Test␣garage␣print␣empty␣garage␣", result.str().empty(), true);
   }

   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣garage␣print␣empty␣garage␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

```

```
    try
    {
       std::string testPlate = "SR770BA";
       Car* testCar = new Car{ "UAZ", Diesel, testPlate };
       Car* testCar2 = new Car{"Mercedes", Benzin, "UU1234AB"};

       testCar->AddRecord({ { 2025y,October,13d }, 25 });
       testCar2->AddRecord({ { 2025y,October,13d }, 25 });

       Garage testGarage;
       testGarage.AddVehicle(testCar);
       testGarage.AddVehicle(testCar2);

       testGarage.DeleteVehicle(testGarage.SearchPlate(testPlate));
       Vehicle const * const testPtr = testGarage.SearchPlate(testPlate);

       Test_OK = Test_OK && check_dump(ost, "Test␣Delete␣Vehicle", testPtr, (Vehicle const* const) 0);
    }

    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣garage␣print␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    //Test Copy and Swap
    try
    {
       std::string testPlate = "SR770BA";
       Car* testCar = new Car{ "UAZ", Diesel, testPlate };
       Car* testCar2 = new Car{ "Mercedes", Benzin, "UU1234AB" };
       Vehicle* const testPtr = nullptr;

       testCar->AddRecord({ { 2025y,October,13d }, 25 });
       testCar2->AddRecord({ { 2025y,October,13d }, 25 });

       Garage testGarage;
       testGarage.AddVehicle(testCar);
       testGarage.AddVehicle(testCar2);

       Garage testGarageCopy = testGarage;

       Test_OK = Test_OK && check_dump(ost, "Test␣Delete␣Vehicle", testPtr, (Vehicle* const)0);
    }

    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣Delete␣Vehicle␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // Test GetTotalDrivenKilometers()

    try
    {
       Car* const testCar1 = new Car{ "Madza", Elektro, "WD40AHAH" };
       Car* const testCar2 = new Car{ "MG", Elektro, "DeiMama" };

       testCar1->AddRecord({ { 2025y,October,13d }, 25 });
       testCar1->AddRecord({ { 2025y,October,28d }, 34 });
       testCar2->AddRecord({ { 2025y,September,13d }, 25 });
       testCar2->AddRecord({ { 2025y,March,28d }, 34 });

       size_t expect = testCar1->GetMilage() + testCar2->GetMilage();

       Garage testGarage;
       testGarage.AddVehicle(testCar1);
       testGarage.AddVehicle(testCar2);

       size_t result = testGarage.GetTotalDrivenKilometers();

       Test_OK = Test_OK && check_dump(ost, "Test␣GetTotalDrivenKilometers()", expect, result);
    }

    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣GetTotalDrivenKilometers()␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    //Test ostream operator
    try
    {
       Car* const testCar1 = new Car{ "Madza", Elektro, "WD40AHAH" };

       testCar1->AddRecord({ { 2025y,October,13d }, 25 });
       testCar1->AddRecord({ { 2025y,October,28d }, 34 });
       Garage testGarage;
       testGarage.AddVehicle(testCar1);

       std::stringstream expect;
       std::stringstream result;

       testGarage.Print(expect);
       result << testGarage;

       Test_OK = Test_OK && check_dump(ost, "Test␣ostream␣operator", expect.str(), result.str());
    }

    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣ostream␣operator␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // Adding Car as nullptr;
    try
    {
       Car* const testCar1 = nullptr;
       Garage testGarage;
```

```
      testGarage.AddVehicle(testCar1);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "TestAdding␣Car␣as␣nullptr;", error_msg, Garage::ERROR_NULLPTR);
   error_msg.clear();

   // Adding Deleting as nullptr;
   try
   {
      Car* const testCar1 = nullptr;
      Garage testGarage;
      testGarage.DeleteVehicle(testCar1);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "TestDeleting␣Car␣as␣nullptr;", error_msg, Garage::ERROR_NULLPTR);
   error_msg.clear();

   // End of garage testing
   ost << TestEnd;
   return Test_OK;
}


static bool Test_Car(ostream& ost) {
   assert(ost.good());

   ost << TestStart;
   bool Test_OK = true;
   std::string error_msg;

   // Test Fuel Type Getter
   try
   {
      TFuel testType = Benzin;
      Car testCar{ "Audi", testType, "SR770BA" };

      Test_OK = Test_OK && check_dump(ost, "Test␣car␣fueltype", testCar.GetFuelType(), testType);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣car␣fueltype␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Plate Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      Car testCar{ "Audi", testType, testPlate };

      Test_OK = Test_OK && check_dump(ost, "Test␣car␣plate", testCar.GetPlate(), testPlate);
   }
```

```
    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣car␣plate␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // Test Brand Getter
    try
    {
       TFuel testType = Benzin;
       std::string testPlate = "SR770BA";
       std::string testBrand= "Steyr";
       Car testCar{ testBrand, testType, testPlate };

       Test_OK = Test_OK && check_dump(ost, "Test␣car␣brand", testCar.GetBrand(), testBrand);
    }
    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣car␣brand␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // Test Milage Getter
    try
    {
       TFuel testType = Benzin;
       std::string testPlate = "SR770BA";
       std::string testBrand = "Steyr";
       Car testCar{ testBrand, testType, testPlate };
       size_t miles = 25;
       testCar.AddRecord({ { 2025y,October,13d }, miles });

       Test_OK = Test_OK && check_dump(ost, "Test␣car␣milage", testCar.GetMilage(), miles);
    }
    catch (const string& err) {
       error_msg = err;
    }
    catch (bad_alloc const& error) {
       error_msg = error.what();
    }
    catch (const exception& err) {
       error_msg = err.what();
    }
    catch (...) {
       error_msg = "Unhandled␣exception";
    }

    Test_OK = Test_OK && check_dump(ost, "Test␣car␣milage␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // Test DriveRecord Getter
    try
    {
       TFuel testType = Benzin;
       std::string testPlate = "SR770BA";
```

```
      std::string testBrand = "Steyr";
      Car testCar{ testBrand, testType, testPlate };
      size_t miles = 25;
      DriveRecord driveRecord;
      RecordEntry recordEntry = { { 2025y,October,13d }, miles };
      driveRecord.AddRecord(recordEntry);
      testCar.AddRecord(recordEntry);

      stringstream expect;
      stringstream result;
      driveRecord.Print(expect);
      testCar.GetDriveRecord().Print(result);
      Test_OK = Test_OK && check_dump(ost, "Test␣car␣driveRecord", expect.str(), result.str());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣car␣driveRecord␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Exception emtpy string
   try
   {
      TFuel testType = Benzin;

      Car testCar{ "", testType, "SB278FH" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Car␣CTOR␣empty␣brand", Vehicle::ERROR_EMPTY_STRING, error_msg);
   error_msg.clear();

   try
   {
      TFuel testType = Benzin;

      Car testCar{ "Audi", testType, "" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Car␣CTOR␣empty␣plate", Vehicle::ERROR_EMPTY_STRING, error_msg);
   error_msg.clear();

   ost << TestEnd;
   return Test_OK;

}


static bool Test_Bike(ostream& ost) {
   assert(ost.good());

   ost << TestStart;
   bool Test_OK = true;
   std::string error_msg;

   // Test Fuel Type Getter
   try
   {
      TFuel testType = Benzin;
      Bike testCar{ "Audi", testType, "SR770BA" };

      Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣fueltype", testCar.GetFuelType(), testType);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣fueltype␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Plate Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      Bike testCar{ "Audi", testType, testPlate };

      Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣plate", testCar.GetPlate(), testPlate);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣plate␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Brand Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Bike testCar{ testBrand, testType, testPlate };

      Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣brand", testCar.GetBrand(), testBrand);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣brand␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Milage Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Car testCar{ testBrand, testType, testPlate };
      size_t miles = 25;
      testCar.AddRecord({ { 2025y,October,13d }, miles });

      Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣milage", testCar.GetMilage(), miles);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣milage␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test DriveRecord Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Bike testCar{ testBrand, testType, testPlate };
      size_t miles = 25;
      DriveRecord driveRecord;
      RecordEntry recordEntry = { { 2025y,October,13d }, miles };
      driveRecord.AddRecord(recordEntry);
      testCar.AddRecord(recordEntry);

      stringstream expect;
      stringstream result;
      driveRecord.Print(expect);
      testCar.GetDriveRecord().Print(result);
      Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣driveRecord", expect.str(), result.str());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣driveRecord␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Exception emtpy string
   try
   {
      TFuel testType = Benzin;

      Bike testCar{ "", testType, "SB278FH" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣CTOR␣empty␣brand", Vehicle::ERROR_EMPTY_STRING, error_msg);
   error_msg.clear();

   try
   {
      TFuel testType = Benzin;

      Bike testCar{ "Audi", testType, "" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Bike␣CTOR␣empty␣plate", Vehicle::ERROR_EMPTY_STRING, error_msg);
   error_msg.clear();

   ost << TestEnd;
   return Test_OK;
}

static bool Test_Truck(ostream& ost){
   assert(ost.good());

   ost << TestStart;
   bool Test_OK = true;
   std::string error_msg;

   // Test Fuel Type Getter
   try
   {
      TFuel testType = Benzin;
      Truck testCar{ "Audi", testType, "SR770BA" };

      Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣fueltype", testCar.GetFuelType(), testType);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣fueltype␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Plate Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      Truck testCar{ "Audi", testType, testPlate };

      Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣plate", testCar.GetPlate(), testPlate);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣plate␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Brand Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Truck testCar{ testBrand, testType, testPlate };

      Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣brand", testCar.GetBrand(), testBrand);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣car␣brand␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Milage Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Truck testCar{ testBrand, testType, testPlate };
      size_t miles = 25;
      testCar.AddRecord({ { 2025y,October,13d }, miles });
      testCar.AddRecord({ { 2025y,October,13d }, miles });

      Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣milage", testCar.GetMilage(), 2*miles);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣Truck␣milage␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test DriveRecord Getter
   try
   {
      TFuel testType = Benzin;
      std::string testPlate = "SR770BA";
      std::string testBrand = "Steyr";
      Truck testCar{ testBrand, testType, testPlate };
      size_t miles = 25;
      DriveRecord driveRecord;
      RecordEntry recordEntry = { { 2025y,October,13d }, miles };
      driveRecord.AddRecord(recordEntry);
      testCar.AddRecord(recordEntry);

      stringstream expect;
      stringstream result;
      driveRecord.Print(expect);
      testCar.GetDriveRecord().Print(result);
      Test_OK = Test_OK && check_dump(ost, "Test␣truck␣driveRecord", expect.str(), result.str());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣truck␣driveRecord␣-␣error␣buffer", error_msg.empty(), true);
   error_msg.clear();

   // Test Exception emtpy string
   try
   {
      TFuel testType = Benzin;

      Truck testCar{ "", testType, "SB278FH" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣truck␣CTOR␣empty␣brand", Vehicle::ERROR_EMPTY_STRING,error_msg);
   error_msg.clear();

   try
   {
      TFuel testType = Benzin;

      Truck testCar{ "Audi", testType, "" };
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandled␣exception";
   }

   Test_OK = Test_OK && check_dump(ost, "Test␣truck␣CTOR␣empty␣plate", Vehicle::ERROR_EMPTY_STRING,error_msg);
   error_msg.clear();


   ost << TestEnd;
   return Test_OK;
}
```