FH-OÖ Hagenberg/HSD
SDP3, WS 2025
Übung 2

Name: Simon Offenberger / Simon Vogelhuber — Aufwand in h: siehe Doku

Mat.Nr: S2410306027 / S2410306014 — Punkte:

Übungsgruppe: 1 — korrigiert:

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

Eine Firma benötigt eine Software für die Verwaltung ihrer Mitarbeiter. Es wird unterschieden zwischen verschiedenen Arten von Mitarbeitern, für die jeweils das Gehalt unterschiedlich berechnet wird.

Jeder Mitarbeiter hat: einen Vor- und einen Nachnamen, ein Namenskürzel (3 Buchstaben), eine Sozialversicherungsnummer (z.B. 1234020378 -> Geburtsdatum: 2. März 1978) und ein Einstiegsjahr (wann der Mitarbeiter zur Firma gekommen ist).

Bei der Bezahlung wird unterschieden zwischen:

- *CommissionWorker*: Grundgehalt + Fixbetrag pro verkauftem Stück
- *HourlyWorker*: Stundenlohn x gearbeitete Monatsstunden
- *PieceWorker*: Summe erzeugter Stücke x Stückwert
- *Boss*: monatliches Fixgehalt

Überlegen Sie sich, welche Members und Methoden die einzelnen Klassen benötigen, um mindestens folgende Abfragen zu ermöglichen:

- Wie viele Mitarbeiter hat die Firma?
- Wie viele *CommissionWorker* arbeiten in der Firma?
- Wie viele Stück wurden im Monat erzeugt?
- Wie viele Stück wurden im Monat verkauft?
- Wie viele Mitarbeiter sind vor 1970 geboren?

- Wie hoch ist das Monatsgehalt eines Mitarbeiters?
- Gibt es einen Mitarbeiter zu einem gegebenen Namenskürzel?
- Welche(r) Mitarbeiter ist/sind am längsten in der Firma?
- Ausgabe aller Datenblätter der Mitarbeiter

Zur Vereinfachung braucht nur ein Monat berücksichtigt werden (d.h. pro Mitarbeiter nur ein Wert für Stückzahl oder verkaufte Stück). Realisieren Sie die Ausgabe des Datenblattes als *Template Method*. Der Ausdruck hat dabei folgendes Aussehen:

```
****************************************
Fa. Hofer, Linz
****************************************
Datenblatt
---------------
Name: Max Huber
Kürzel: mhu
Sozialversicherungsnummer: 1234010273
Einstiegsjahr: 2005
Mitarbeiterklasse: CommissionWorker
Grundgehalt: 2500 EUR
Provision: 350 EUR
Gesamtgehalt: 2850 EUR
----------------------------------------
v1.0 Oktober 2025
----------------------------------------
```

Achten Sie bei Ihrem Entwurf auf die Einhaltung der Design-Prinzipien!

Schreiben Sie einen Testtreiber, der mehrere Mitarbeiter aus den unterschiedlichen Gruppen anlegt. Die erforderlichen Abfragen werden von einer Klasse `Client` durchgeführt und die Ergebnisse ausgegeben. Achten Sie darauf, dass diese Klasse nicht von Implementierungen abhängig ist.

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

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

Systemdokumentation
Projekt Gehaltsberechnung

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 23. Oktober 2025

Inhaltsverzeichnis

1 Organisatorisches **5**
1.1 Team 5
1.2 Aufteilung der Verantwortlichkeitsbereiche 5
1.3 Aufwand 6

2 Anforderungsdefinition (Systemspezifikation) **7**

3 Systementwurf **9**
3.1 Klassendiagramm 9
3.2 Designentscheidungen 10

4 Dokumentation der Komponenten (Klassen) **10**

5 Testprotokollierung **11**

6 Quellcode **19**
6.1 Object.hpp 19
6.2 Client.hpp 20
6.3 Client.cpp 22
6.4 IComp.hpp 26
6.5 Company.hpp 28
6.6 Company.cpp 30
6.7 TWorker.hpp 32
6.8 Employee.hpp 33
6.9 Employee.cpp 35
6.10 Boss.hpp 36
6.11 Boss.cpp 37
6.12 HourlyWorker.hpp 38
6.13 HourlyWorker.cpp 39
6.14 PieceWorker.hpp 40
6.15 PieceWorker.cpp 41
6.16 ComissionWorker.hpp 42
6.17 ComissionWorker.cpp 43
6.18 main.cpp 44
6.19 Test.hpp 55

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@fhooe.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
  - Design Klassendiagramm
  - Implementierung und Test der Klassen:
    - Company
    - Company Interface
    - Client
  - Implementierung des Testtreibers
  - Dokumentation
- Simon Vogelhuber
  - Design Klassendiagramm
  - Implementierung und Komponententest der Klassen:
    - Employee
    - Boss
    - ComissionWorker

    - PieceWorker
    - HourlyWorker
  - Implementierung des Testtreibers
  - Dokumentation

1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich 7 Ph
- Simon Vogelhuber: geschätzt 10 Ph / tatsächlich 8 Ph

2 Anforderungsdefinition (Systemspezifikation)

In diesem Projekt geht es darum die Mitarbeiter eines Unternehmens zu verwalten und deren Gehälter zu berechnen. Es gibt verschiedene Arten von Mitarbeitern, welche unterschiedliche Gehaltsberechnungen haben. Der Zugriff auf die Mitarbeiter soll über eine gemeinsame Schnittstelle erfolgen.

Funktionen der Firmenschnittstelle

- Zugriff auf die wichtigsten Mitarbeiter und Firmendaten

Funktionen der Firma

- Abfage nach der Anzahl der Mitarbeiter.
- Abfage nach der Anzahl eines Mitarbeitertyps in der Firma
- Wie viele Stück wurden im Monat erzeugt?
- Wie viele Stück wurden im Monat verkauft?
- Wie viele Mitarbeiter sind vor einem bestimmten Datum geboren?
- Wie hoch ist das Monatsgehalt eines Mitarbeiters?
- Gibt es einen Mitarbeiter zu einem gegebenen Namenskürzel?
- Welche(r) Mitarbeiter ist/sind am längsten in der Firma?
- Ausgabe aller Datenblätter der Mitarbeiter

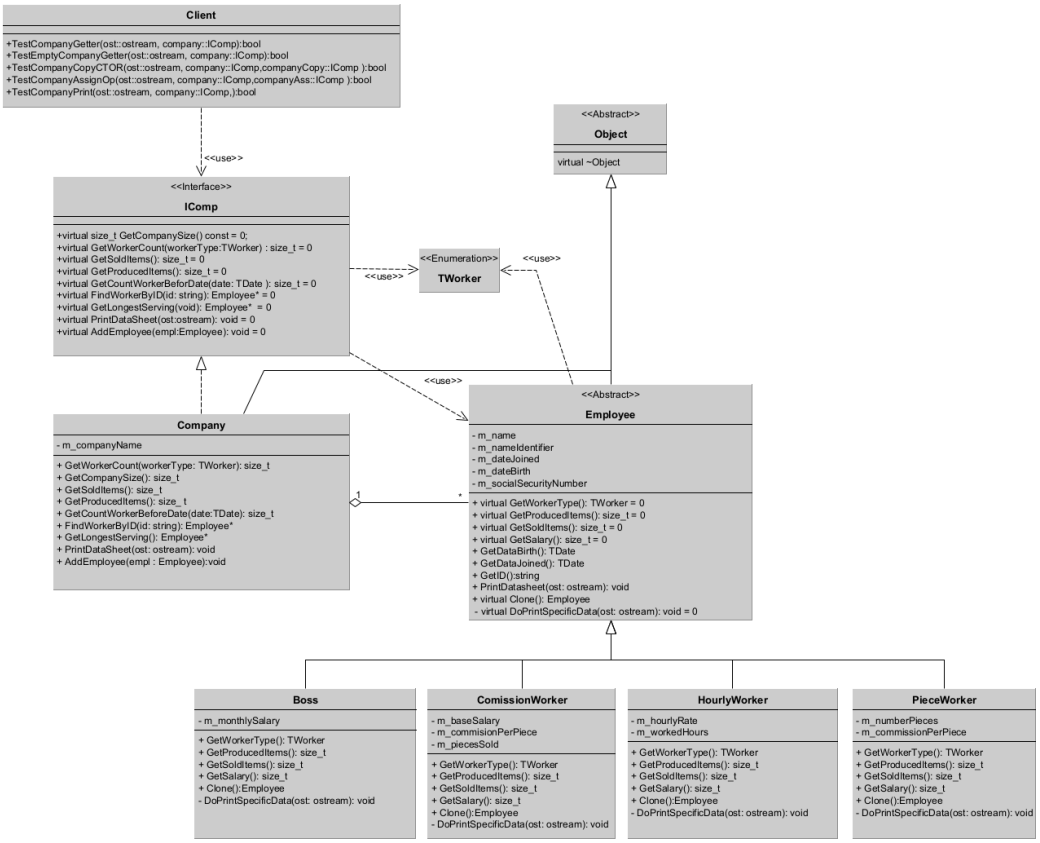
Funktionen der Mitarbeiter

- Speichern von Mitarbeiterdaten.
  - Name
  - Namenskürzel

  - Sozialversicherungsnummer
  - Einstiegsjahr
  - Geburtsjahr
- Berechnung des Gehalts je nach Mitarbeiterklasse.
- Ausgabe von Mitarbeiterinformationen in form eines Datenblatts.

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Das Interface **ICompany** wurde erstellt, um dem zugreifenden **Client** eine Schnittstelle zur Verfügung zu stellen. Dadurch kann sich der Client auf die Schnittstelle konzentrieren und muss sich nicht um die Implementierungsdetails der Firma kümmern.

Die Firma speichert einen polymorphen Container, der Objekte der abstrakten Klasse **Employee** verwaltet. Bei dem Container wurde eine Map verwendet, da die Mitarbeiter über eine eindeutige ID angesprochen werden können. Somit ist auch das Suchen nach einem Mitarbeiter sehr performant gelöst.

Die Klasse **Employee** ist abstrakt, da es keine generellen Mitarbeiter geben soll, sondern nur spezielle Arten von Mitarbeitern. Die einzelnen Mitarbeiter speichern Daten, die für die Gehaltsberechnung notwendig sind. Die Gehaltsberechnung wird über eine virtuelle Funktion realisiert, die in den abgeleiteten Klassen überschrieben wird. Weiters soll die Ausgabe eines Datenblatts zu jedem Mitarbeiter möglich sein dies wurde mittels **Template Methode Pattern** gelöst!

Das Enum mit dem Mitarbeitertypen **TWorker** wurde eingebaut, da die Company den Typen des Mitarbeiters kennen muss, um den Mitarbeiter korrekt zu verwalten. Hierbei wurde aktiv auf RTTI verzichtet, um die Kopplung zwischen Company und den konkreten Klassen die von Employee ableiten zu reduzieren.

4 Dokumentation der Komponenten (Klassen)

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

5 Testprotokollierung

```
1
2  ****************************************
3              TESTCASE START
4  ****************************************
5
6  Test Company Get Comission Worker Cnt & Add Empl
7  [Test OK] Result: (Expected: 2 == Result: 2)
8
9  Test Company Get Houerly Worker Cnt & Add Empl
10 [Test OK] Result: (Expected: 1 == Result: 1)
11
12 Test Company Get Boss Cnt & Add Empl
13 [Test OK] Result: (Expected: 1 == Result: 1)
14
15 Test Company Get Piece Worker Cnt & Add Empl
16 [Test OK] Result: (Expected: 2 == Result: 2)
17
18 Test Company FindWorker by ID
19 [Test OK] Result: (Expected: Si1 == Result: Si1)
20
21 Test Company FindWorker by empty ID
22 [Test OK] Result: (Expected: 0000000000000000 == Result:
      ↪ 0000000000000000)
23
24 Test Company Get Size
25 [Test OK] Result: (Expected: 6 == Result: 6)
26
27 Test Company Get Count worker bevor 1930 date
28 [Test OK] Result: (Expected: 0 == Result: 0)
29
30 Test Company Get Count worker bevor 1951 date
31 [Test OK] Result: (Expected: 2 == Result: 2)
32
33 Test Company Get longest serving employee
34 [Test OK] Result: (Expected: 0 == Result: 0)
35
36 Test Company Get total pieces produced
37 [Test OK] Result: (Expected: 50 == Result: 50)
38
39 Test Company Get total pieces sold
40 [Test OK] Result: (Expected: 2700 == Result: 2700)
41
```

```
42
43 ****************************************
44
45
46 ****************************************
47              TESTCASE START
48 ****************************************
49
50 Test Company Copy Ctor
51 [Test OK] Result: (Expected: true == Result: true)
52
53
54 ****************************************
55
56
57 ****************************************
58              TESTCASE START
59 ****************************************
60
61 Test Company Assign Operator
62 [Test OK] Result: (Expected: true == Result: true)
63
64
65 ****************************************
66
67
68 ****************************************
69              TESTCASE START
70 ****************************************
71
72 Test Company Print Exception
73 [Test OK] Result: (Expected: ERROR: Provided Ostream is bad ==
    ↪  Result: ERROR: Provided Ostream is bad)
74
75
76 ****************************************
77
78
79 ****************************************
80              TESTCASE START
81 ****************************************
82
83 Test Empty Company Get Comission Worker Cnt & Add Empl
84 [Test OK] Result: (Expected: 0 == Result: 0)
```

```
85
86  Test Empty Company Get Houerly Worker Cnt & Add Empl
87  [Test OK] Result: (Expected: 0 == Result: 0)
88
89  Test Empty Company Get Boss Cnt & Add Empl
90  [Test OK] Result: (Expected: 0 == Result: 0)
91
92  Test Empty Company Get Piece Worker Cnt & Add Empl
93  [Test OK] Result: (Expected: 0 == Result: 0)
94
95  Test Empty Company FindWorker by ID
96  [Test OK] Result: (Expected: 0000000000000000 == Result:
        ↪ 0000000000000000)
97
98  Test Empty Company FindWorker by ID empty ID
99  [Test OK] Result: (Expected: 0000000000000000 == Result:
        ↪ 0000000000000000)
100
101 Test Empty Company Get Size
102 [Test OK] Result: (Expected: 0 == Result: 0)
103
104 Test Empty Company Get Count worker bevor 1930 date
105 [Test OK] Result: (Expected: 0 == Result: 0)
106
107 Test Empty Company Get Count worker bevor 1951 date
108 [Test OK] Result: (Expected: 0 == Result: 0)
109
110 Test Empty Company Get longest serving employee
111 [Test OK] Result: (Expected: 0000000000000000 == Result:
        ↪ 0000000000000000)
112
113 Test Empty Company Get total pieces produced
114 [Test OK] Result: (Expected: 0 == Result: 0)
115
116 Test Empty Company Get total pieces sold
117 [Test OK] Result: (Expected: 0 == Result: 0)
118
119 Test Company Add nullptr
120 [Test OK] Result: (Expected: ERROR: Passed in Nullptr! ==
        ↪ Result: ERROR: Passed in Nullptr!)
121
122
123 ****************************************
124
```

```
125
126 ****************************************
127              TESTCASE START
128 ****************************************
129
130 Test - Boss.GetSalary()
131 [Test OK] Result: (Expected: 7800 == Result: 7800)
132
133 Test - Boss.GetSoldItems()
134 [Test OK] Result: (Expected: 0 == Result: 0)
135
136 Test - Boss.GetProducedItems()
137 [Test OK] Result: (Expected: 0 == Result: 0)
138
139 Test - Boss.GetWorkerType()
140 [Test OK] Result: (Expected: 0 == Result: 0)
141
142 Test - Boss.GetDateBirth()
143 [Test OK] Result: (Expected: 2000-11-22 == Result: 2000-11-22)
144
145 Test - Boss.GetDateJoined()
146 [Test OK] Result: (Expected: 2022-11-23 == Result: 2022-11-23)
147
148 Test - error buffer
149 [Test OK] Result: (Expected: true == Result: true)
150
151 Test Boss.Clone()
152 [Test OK] Result: (Expected: true == Result: true)
153
154 Test - error buffer
155 [Test OK] Result: (Expected: true == Result: true)
156
157 Boss Constructor bad ID
158 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪  to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
159
160 Boss Constructor bad SV - invalid character
161 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
162
163 Boss Constructor bad SV - too many nums
164 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
```

```
165
166
167 ****************************************
168
169
170 ****************************************
171              TESTCASE START
172 ****************************************
173
174 Test - HourlyWorker.GetSalary()
175 [Test OK] Result: (Expected: 3360 == Result: 3360)
176
177 Test - HourlyWorker.GetSoldItems()
178 [Test OK] Result: (Expected: 0 == Result: 0)
179
180 Test - HourlyWorker.GetProducedItems()
181 [Test OK] Result: (Expected: 0 == Result: 0)
182
183 Test - HourlyWorker.GetWorkerType()
184 [Test OK] Result: (Expected: 2 == Result: 2)
185
186 Test - HourlyWorker.GetDateBirth()
187 [Test OK] Result: (Expected: 2000-11-22 == Result: 2000-11-22)
188
189 Test - HourlyWorker.GetDateJoined()
190 [Test OK] Result: (Expected: 2022-11-23 == Result: 2022-11-23)
191
192 Test - error buffer
193 [Test OK] Result: (Expected: true == Result: true)
194
195 Test testPieceWorker.Clone()
196 [Test OK] Result: (Expected: true == Result: true)
197
198 Test - error buffer
199 [Test OK] Result: (Expected: true == Result: true)
200
201 HourlyWorker Constructor bad ID
202 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪  to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
203
204 HourlyWorker Constructor bad SV - invalid character
205 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
```

```
206
207 HourlyWorker Constructor bad SV - too many nums
208 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
209
210
211 ****************************************
212
213
214 ****************************************
215             TESTCASE START
216 ****************************************
217
218 Test - PieceWorker.GetSalary()
219 [Test OK] Result: (Expected: 1900 == Result: 1900)
220
221 Test - PieceWorker.GetSoldItems()
222 [Test OK] Result: (Expected: 0 == Result: 0)
223
224 Test - PieceWorker.GetProducedItems()
225 [Test OK] Result: (Expected: 950 == Result: 950)
226
227 Test - PieceWorker.GetWorkerType()
228 [Test OK] Result: (Expected: 3 == Result: 3)
229
230 Test - PieceWorker.GetDateBirth()
231 [Test OK] Result: (Expected: 2000-11-22 == Result: 2000-11-22)
232
233 Test - PieceWorker.GetDateJoined()
234 [Test OK] Result: (Expected: 2022-11-23 == Result: 2022-11-23)
235
236 Test - error buffer
237 [Test OK] Result: (Expected: true == Result: true)
238
239 Test testPieceWorker.Clone()
240 [Test OK] Result: (Expected: true == Result: true)
241
242 Test - error buffer
243 [Test OK] Result: (Expected: true == Result: true)
244
245 PieceWorker Constructor bad ID
246 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪  to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
```

```
247
248 PieceWorker Constructor bad SV - invalid character
249 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
250
251 PieceWorker Constructor bad SV - too many nums
252 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
253
254
255 ****************************************
256
257
258 ****************************************
259              TESTCASE START
260 ****************************************
261
262 Test - ComissionWorker.GetSalary()
263 [Test OK] Result: (Expected: 2900 == Result: 2900)
264
265 Test - ComissionWorker.GetSoldItems()
266 [Test OK] Result: (Expected: 300 == Result: 300)
267
268 Test - ComissionWorker.GetProducedItems()
269 [Test OK] Result: (Expected: 0 == Result: 0)
270
271 Test - ComissionWorker.GetWorkerType()
272 [Test OK] Result: (Expected: 1 == Result: 1)
273
274 Test - ComissionWorker.GetDateBirth()
275 [Test OK] Result: (Expected: 2000-11-22 == Result: 2000-11-22)
276
277 Test - ComissionWorker.GetDateJoined()
278 [Test OK] Result: (Expected: 2022-11-23 == Result: 2022-11-23)
279
280 Test - error buffer
281 [Test OK] Result: (Expected: true == Result: true)
282
283 Test testPieceWorker.Clone()
284 [Test OK] Result: (Expected: true == Result: true)
285
286 Test - error buffer
287 [Test OK] Result: (Expected: true == Result: true)
288
```

```
289 ComissionWorker Constructor bad ID
290 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪  to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
291
292 ComissionWorker Constructor bad SV - invalid character
293 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
294
295 ComissionWorker Constructor bad SV - too many nums
296 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
297
298
299 ******************************************
300
301
302 ******************************************
303             TESTCASE START
304 ******************************************
305
306 Test Exception in Company Add Duplicate
307 [Test OK] Result: (Expected: ERROR: Duplicate Employee! ==
    ↪ Result: ERROR: Duplicate Employee!)
308
309
310 ******************************************
311
312 TEST OK!!
```

6 Quellcode

6.1 Object.hpp

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

class Object {
public:

        /**
         * \brief Constant for Exception Bad Ostream.
         */
        inline static const std::string ERROR_BAD_OSTREAM = "ERROR: Provided Ostream is bad";

        /**
         * \brief Constant for Exception Fail Write.
         */
        inline static const std::string ERROR_FAIL_WRITE = "ERROR: Fail to write on provided Ostream";

        /**
         * \brief Constant for Exception Nullprt.
         */
        inline static const std::string ERROR_NULLPTR = "ERROR: Passed in Nullptr!";

protected:

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

        /**
         * \brief virtual DTOR -> once Virtual always virtual.
         *
         */
        virtual ~Object() = default;

};

#endif // !OBJECT_HPP
```

6.2 Client.hpp

```
/*****************************************************************//**
 * \file   Client.hpp
 * \brief  Client Class that uses the Class Company via the Interface IComp
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef CLIENT_HPP
#define CLIENT_HPP

#include <iostream>
#include "IComp.hpp"

class Client {
public:
        /**
         * Constant for Exception Bad Ostream.
         */
        inline static const std::string ERROR_BAD_OSTREAM = "ERROR: Provided Ostream is bad";

        /**
         * Constant for Exception Write Fail.
         */
        inline static const std::string ERROR_FAIL_WRITE = "ERROR: Fail to write on provided Ostream";

        /**
         * \brief Test Methode for the Getter Methodes of the Company via the Interface.
         *
         * \param ost Refernce to an ostream where the Test results should be printed at
         * \param company Reference to a company interface
         * \return true -> Test OK
         * \return false -> Test NOK
         */
        bool TestCompanyGetter(std::ostream & ost,const IComp& company) const;

        /**
         * \brief Test Methode for the Getter Methodes of an Empty Company via the Interface.
         *
         * \param ost Refernce to an ostream where the Test results should be printed at
         * \param company Reference to a company interface
         * \return true -> Test OK
         * \return false -> Test NOK
         */
        bool TestEmptyCompanyGetter(std::ostream & ost, IComp& company) const;

        /**
         * \brief Test Methode for testing the Copy Ctor of the Company
         *
         * \param ost Refernce to an ostream where the Test results should be printed at
         * \param company Reference to a company interface
         * \param companyCopy Reference to the copy of company
         * \return true -> Test OK
         * \return false -> Test NOK
         */
        bool TestCompanyCopyCTOR(std::ostream & ost,const IComp& company,const IComp& companyCopy) const;

        /**
         * \brief Test Methode for the Assign Operator of Company
         *
         * \param ost Refernce to an ostream where the Test results should be printed at
         * \param company Reference to a company interface
         * \param companyAss Reference to the assigned Company should be Equal to company
         * \return true -> Test OK
         * \return false -> Test NOK
         */
        bool TestCompanyAssignOp(std::ostream & ost,const IComp& company,const IComp& companyAss) const;

        /**
         * \brief Test Methode for the Print Methode of Company
         *
         * \param ost Refernce to an ostream where the Test results should be printed at
         * \param company Reference to a company interface
         * \return true -> Test OK
```

```
74            * \return false -> Test NOK
75            */
76           bool TestCompanyPrint(std::ostream & ost,const ICOmp& company) const;
77
78   };
79
80   #endif // !CLIENT_HPP
```

6.3 Client.cpp

```cpp
/*****************************************************************//**
 * \file   Client.hpp
 * \brief  Client Class that uses the Class Company via the Interface IComp
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/

#include "Client.hpp"
#include "Test.hpp"
#include "ComissionWorker.hpp"
#include "HourlyWorker.hpp"
#include "Boss.hpp"
#include "PieceWorker.hpp"
#include <sstream>
#include <fstream>

using namespace std;
using namespace std::chrono;

bool Client::TestCompanyGetter(std::ostream& ost,const IComp & company) const
{
	if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;

	TestStart(ost);

	bool TestOK = true;
	string error_msg = "";


	try {

		TestOK = TestOK && check_dump(ost, "Test Company Get Comission Worker Cnt & Add Empl",  static_cast<size_t>(2), company.GetWorkerCount(TWorker::E_CommisionWorker));
		TestOK = TestOK && check_dump(ost, "Test Company Get Houerly Worker Cnt & Add Empl",    static_cast<size_t>(1), company.GetWorkerCount(TWorker::E_HourlyWorker));
		TestOK = TestOK && check_dump(ost, "Test Company Get Boss Cnt & Add Empl",                                  static_cast<size_t>(1), company.GetWorkerCount(TWorker::E_Boss));
		TestOK = TestOK && check_dump(ost, "Test Company Get Piece Worker Cnt & Add Empl",             static_cast<size_t>(2), company.GetWorkerCount(TWorker::E_PieceWorker));


		TestOK = TestOK && check_dump(ost, "Test Company FindWorker by ID",                    static_cast<std::string>("Si1"), company.FindWorkerByID("Si1")->GetID());
		TestOK = TestOK && check_dump(ost, "Test Company FindWorker by empty ID",   static_cast<const Employee *>(nullptr), company.FindWorkerByID(""));

		TestOK = TestOK && check_dump(ost, "Test Company Get Size",                                    static_cast<size_t>(6), company.GetCompanySize());

		TestOK = TestOK && check_dump(ost, "Test Company Get Count worker bevor 1930 date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1930y,November,23d }));
		TestOK = TestOK && check_dump(ost, "Test Company Get Count worker bevor 1951 date", static_cast<size_t>(2), company.GetCountWorkerBeforDate({ 1951y,November,23d }));

		TestOK = TestOK && check_dump(ost, "Test Company Get longest serving employee", TWorker::E_Boss, company.GetLongestServing()->GetWorkerType());

		TestOK = TestOK && check_dump(ost, "Test Company Get total pieces produced", static_cast<size_t>(50), company.GetProducedItems());

		TestOK = TestOK && check_dump(ost, "Test Company Get total pieces sold", static_cast<size_t>(2700), company.GetSoldItems());

	}
	catch (const string& err) {
		error_msg = err;
		TestOK = false;
	}
	catch (bad_alloc const& error) {
		error_msg = error.what();
		TestOK = false;
	}
	catch (const exception& err) {
		error_msg = err.what();
		TestOK = false;
	}
	catch (...) {
		error_msg = "Unhandelt Exception";
		TestOK = false;
	}

	TestEnd(ost);

	if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
```

```
74
75          return TestOK;
76  }
77
78  bool Client::TestEmptyCompanyGetter(std::ostream& ost,IComp& company) const
79  {
80          if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
81
82          TestStart(ost);
83
84          bool TestOK = true;
85          string error_msg = "";
86
87
88          try {
89
90                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Comission␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_CommisionWorker));
91                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Houerly␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_HourlyWorker));
92                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Boss␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_Boss));
93                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Piece␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_PieceWorker));
94
95
96                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣FindWorker␣by␣ID", static_cast<const Employee *>(nullptr), company.FindWorkerByID("Si1"));
97                  TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣FindWorker␣by␣ID␣empty␣ID", static_cast<const Employee *>(nullptr), company.FindWorkerByID(""));
98
99
100                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Size", static_cast<size_t>(0), company.GetCompanySize());
101
102                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Count␣worker␣bevor␣1930␣date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1930y,November,23d }));
103                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Count␣worker␣bevor␣1951␣date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1951y,November,23d }));
104
105                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣longest␣serving␣employee", static_cast<const Employee*>(nullptr), company.GetLongestServing());
106
107
108                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣total␣pieces␣produced", static_cast<size_t>(0), company.GetProducedItems());
109
110                 TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣total␣pieces␣sold", static_cast<size_t>(0), company.GetSoldItems());
111
112         }
113         catch (const string& err) {
114                 error_msg = err;
115                 TestOK = false;
116         }
117         catch (bad_alloc const& error) {
118                 error_msg = error.what();
119                 TestOK = false;
120         }
121         catch (const exception& err) {
122                 error_msg = err.what();
123                 TestOK = false;
124         }
125         catch (...) {
126                 error_msg = "Unhandelt␣Exception";
127                 TestOK = false;
128         }
129
130         try {
131
132                 company.AddEmployee(nullptr);
133         }
134         catch (const string& err) {
135                 error_msg = err;
136         }
137         catch (bad_alloc const& error) {
138                 error_msg = error.what();
139         }
140         catch (const exception& err) {
141                 error_msg = err.what();
142         }
143         catch (...) {
144                 error_msg = "Unhandelt␣Exception";
145         }
146
147
148         TestOK = TestOK && check_dump(ost, "Test␣Company␣Add␣nullptr", Object::ERROR_NULLPTR, error_msg);
149
```

```
150         TestEnd(ost);
151
152         if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
153
154         return TestOK;
155 }
156
157 bool Client::TestCompanyCopyCTOR(std::ostream& ost,const IComp& company,const IComp& companyCopy) const
158 {
159
160         if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
161
162         TestStart(ost);
163
164         bool TestOK = true;
165         string error_msg = "";
166
167         try {
168
169                 stringstream result;
170                 stringstream expected;
171
172                 company.PrintDataSheet(expected);
173                 companyCopy.PrintDataSheet(result);
174
175                 TestOK = TestOK && check_dump(ost, "Test␣Company␣Copy␣Ctor", true ,expected.str() == result.str());
176
177         }
178         catch (const string& err) {
179                 error_msg = err;
180                 TestOK = false;
181         }
182         catch (bad_alloc const& error) {
183                 error_msg = error.what();
184                 TestOK = false;
185         }
186         catch (const exception& err) {
187                 error_msg = err.what();
188                 TestOK = false;
189         }
190         catch (...) {
191                 error_msg = "Unhandelt␣Exception";
192                 TestOK = false;
193         }
194
195         TestEnd(ost);
196
197         if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
198
199         return TestOK;
200
201         return false;
202 }
203
204 bool Client::TestCompanyAssignOp(std::ostream& ost,const IComp& company,const IComp& companyAss) const
205 {
206         if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
207
208         TestStart(ost);
209
210         bool TestOK = true;
211         string error_msg = "";
212
213         try {
214
215                 stringstream result;
216                 stringstream expected;
217
218                 company.PrintDataSheet(expected);
219                 companyAss.PrintDataSheet(result);
220
221                 TestOK = TestOK && check_dump(ost, "Test␣Company␣Assign␣Operator", true, expected.str() == result.str());
222
223         }
224         catch (const string& err) {
225                 error_msg = err;
```

```
226                 TestOK = false;
227         }
228         catch (bad_alloc const& error) {
229                 error_msg = error.what();
230                 TestOK = false;
231         }
232         catch (const exception& err) {
233                 error_msg = err.what();
234                 TestOK = false;
235         }
236         catch (...) {
237                 error_msg = "Unhandelt Exception";
238                 TestOK = false;
239         }
240
241         TestEnd(ost);
242
243         if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
244
245         return TestOK;
246
247         return false;
248 }
249
250 bool Client::TestCompanyPrint(std::ostream& ost, const IComp& company) const
251 {
252         if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
253
254         TestStart(ost);
255
256         bool TestOK = true;
257         string error_msg = "";
258
259         fstream badstream;
260         badstream.setstate(ios::badbit);
261
262         try {
263
264                 company.PrintDataSheet(badstream);
265
266         }
267         catch (const string& err) {
268                 error_msg = err;
269         }
270         catch (bad_alloc const& error) {
271                 error_msg = error.what();
272         }
273         catch (const exception& err) {
274                 error_msg = err.what();
275         }
276         catch (...) {
277                 error_msg = "Unhandelt Exception";
278         }
279
280         TestOK = TestOK && check_dump(ost, "Test Company Print Exception", Client::ERROR_BAD_OSTREAM, error_msg);
281
282         badstream.close();
283
284         TestEnd(ost);
285
286         if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
287
288         return TestOK;
289
290         return false;
291 }
```

6.4 IComp.hpp

```
/*****************************************************************//**
 * \file   IComp.hpp
 * \brief  Interface which is implemented by the company and used by the client
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef ICOMP_HPP
#define ICOMP_HPP

#include <string>
#include "TWorker.hpp"
#include "Employee.hpp"

class IComp{
public:

        /**
         * \brief Gets the current size of the company.
         *
         * \return Size of the company
         */
        virtual size_t GetCompanySize() const = 0;

        /**
         * \brief Get the Count of a specific Worker Type.
         *
         * \param workerType Worker Type from which the count should be determined
         * \return Count of the Worker Type in the Company
         */
        virtual size_t GetWorkerCount(const TWorker & workerType) const = 0;

        /**
         * \brief Get the amount of Sold Items in the whole company.
         *
         * \return Amout of Sold Items
         */
        virtual size_t GetSoldItems() const = 0;

        /**
         * \brief Get the amount of produced items.
         *
         * \return Amout of produced Items
         */
        virtual size_t GetProducedItems() const = 0;

        /**
         * \brief Get the of worker with birth date bevor date.
         *
         * \param date to get the employees which are older
         * \return Amout of employees which are older than the passed in birthdate
         */
        virtual size_t GetCountWorkerBeforDate(const TDate & date) const = 0;

        /**
         * \brief Find a worker with a specific ID.
         *
         * \param id ID for which should be searched for
         * \return nullptr if no Empl is found
         * \return Pointer to Employee
         */
        virtual Employee const * FindWorkerByID(const std::string & id) const = 0;

        /**
         * \brief Get the Employee which has been the longest serving.
         *
         * \return nullptr if company is empty
         * \return Pointer to Employee
         */
        virtual Employee const * GetLongestServing(void) const = 0;

        /**
         * \brief Prints a Datasheet for each employee.
```

```
74            *
75            * \param ost ostream where the Datasheet should be printed at
76            * \return referenced ostream
77            */
78           virtual std::ostream& PrintDataSheet(std::ostream& ost) const = 0;
79
80           /**
81            * \brief Adds am Employee to the Company
82            * \brief The company now owns the Employee and is responsible for destructing of Employee.
83            *
84            * \param empl Employee that should be added to the Company
85            * \throw ERROR_DUPLICATE_EMPL if ID of Employee is already in the collection
86            * \throw ERROR_NULLPTR if an Nullptr is passed in
87            */
88           virtual void AddEmployee(Employee const* empl) = 0;
89
90           /**
91            * \brief Virtual Dtor of Icomp.
92            *
93            */
94           virtual ~IComp() = default;
95   };
96
97   #endif // !ICOMP_HPP
```

6.5 Company.hpp

```
/*****************************************************************//**
 * \file   Company.hpp
 * \brief  Company that holds Employees and provides information about the
 * \brief  Employees of the company.
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#ifndef COMPANY_HPP
#define COMPANY_HPP

#include <map>
#include <string>
#include "Object.hpp"
#include "IComp.hpp"

/**
 * Declaration of an alias for the used Container.
 */
using  TContEmployee = std::map<const std::string,Employee const *>;

class Company : public Object, public IComp{
public:
        /**
         * Constant for the Excetion of an Duplicate Employee.
         */
        inline static const std::string ERROR_DUPLICATE_EMPL = "ERROR: Duplicate Employee!";

        /**
         * \brief CTOR for a Company.
         *
         * \param name Name of the Company
         */
        Company(const std::string & name) : m_companyName{ name } {}

        /**
         * \brief Copy Ctor of the Company.
         *
         * \param comp Refernce to the company that should be copied
         */
        Company(const Company & comp);

        /**
         * \brief Assignoperator for a company uses Copy and Swap.
         *
         * \param comp Copy of the company
         */
        void operator=(Company comp);

        /**
         * \brief Adds am Employee to the Company
         * \brief The company now owns the Employee and is responsible for destructing of Employee.
         *
         * \param empl Employee that should be added to the Company
         * \throw ERROR_DUPLICATE_EMPL if ID of Employee is already in the collection
         * \throw ERROR_NULLPTR if an Nullptr is passed in
         */
        virtual void AddEmployee(Employee const* empl) override;

        /**
         * \brief Gets the current size of the company.
         *
         * \return Size of the company
         */
        virtual size_t GetCompanySize() const override;

        /**
         * \brief Get the Count of a specific Worker Type.
         *
         * \param workerType Worker Type from which the count should be determined
         * \return Count of the Worker Type in the Company
         */
        virtual size_t GetWorkerCount(const TWorker& workerType) const override;
```

```
74
75          /**
76           * \brief Get the amount of Sold Items in the whole company.
77           *
78           * \return Amout of Sold Items
79           */
80          virtual size_t GetSoldItems() const override;
81
82          /**
83           * \brief Get the amount of produced items.
84           *
85           * \return Amout of produced Items
86           */
87          virtual size_t GetProducedItems() const override;
88
89          /**
90           * \brief Get the of worker with birth date bevor date.
91           *
92           * \param date to get the employees which are older
93           * \return Amout of employees which are older than the passed in birthdate
94           */
95          virtual size_t GetCountWorkerBeforDate(const TDate& date) const override;
96
97          /**
98           * \brief Find a worker with a specific ID.
99           *
100          * \param id ID for which should be searched for
101          * \return nullptr if no Empl is found
102          * \return Pointer to Employee
103          */
104         virtual Employee const * FindWorkerByID(const std::string& id) const override;
105
106         /**
107          * \brief Get the Employee which has been the longest serving.
108          *
109          * \return nullptr if company is empty
110          * \return Pointer to Employee
111          */
112         virtual Employee const * GetLongestServing(void) const override;
113
114         /**
115          * \brief Prints a Datasheet for each employee.
116          *
117          * \param ost ostream where the Datasheet should be printed at
118          * \return referenced ostream
119          */
120         virtual std::ostream& PrintDataSheet(std::ostream& ost) const override;
121
122         /**
123          * \brief DTOR of the Company.
124          *
125          */
126         ~Company();
127
128 private:
129
130         std::string m_companyName;
131         TContEmployee m_Employees;
132 };
133
134 #endif // !COMPANY_HPP
```

6.6 Company.cpp

```cpp
/*****************************************************************//**
 * \file   Company.hpp
 * \brief  Company that holds Employees and provides information about the
 * \brief  Employees of the company.
 *
 * \author Simon Offenberger
 * \date   October 2025
 *********************************************************************/
#include <algorithm>
#include <numeric>
#include <iostream>
#include "Company.hpp"
#include "Employee.hpp"
using namespace std;

/**
 * \brief Ostream manipulater for creating a horizontal line.
 *
 * \return string
 */
static ostream & hline(ostream & ost) {

	ost << string(60, '-') << endl;
	return ost;
}

/**
 * \brief Ostream manipulater for creating a horizontal line.
 *
 * \return string
 */
static ostream & hstar(ostream & ost) {

	ost << string(60, '*') << endl;
	return ost;
}

void Company::AddEmployee(Employee const* empl)
{
	if (empl == nullptr) throw Object::ERROR_NULLPTR;
	// insert returns a pair. First = Iterator, Second bool -> bool indicates if the insertion was succsessful.
	if (!m_Employees.insert({ empl->GetID(),empl }).second) throw Company::ERROR_DUPLICATE_EMPL;
}

Company::Company(const Company& comp)
{
	// copy Company name
	m_companyName = comp.m_companyName;

	// clone all employees from one company to the other
	for_each(
		comp.m_Employees.cbegin(), comp.m_Employees.cend(),
		[&](auto& e) {AddEmployee(e.second->Clone());
		});
}

void Company::operator=(Company comp)
{
	// copy and swap
	std::swap(m_Employees, comp.m_Employees);
	std::swap(m_companyName, comp.m_companyName);
}

size_t Company::GetCompanySize() const
{
	return m_Employees.size();
}

size_t Company::GetWorkerCount(const TWorker& workerType) const
{
	// Count all Employees where workerType is equal
	return count_if(m_Employees.cbegin(), m_Employees.cend(),
		[&](auto& e) {return e.second->GetWorkerType() == workerType;});
```

```
}

size_t Company::GetSoldItems() const
{
	return accumulate(m_Employees.cbegin(), m_Employees.cend(),static_cast<size_t>(0),
		[](size_t val, const auto& e) { return val + e.second->GetSoldItems();});
}

size_t Company::GetProducedItems() const
{
	return accumulate(m_Employees.cbegin(), m_Employees.cend(), static_cast<size_t>(0),
		[](size_t val, const auto& e) { return val + e.second->GetProducedItems();});
}

size_t Company::GetCountWorkerBeforDate(const TDate& date) const
{
	return count_if(m_Employees.cbegin(), m_Employees.cend(),
		[&](const auto& e) {return e.second->GetDateBirth() < date;});
}

Employee const * Company::FindWorkerByID(const std::string& id) const
{
	auto empl = m_Employees.find(id);

	if (empl == m_Employees.cend()) return nullptr;
	else return empl->second;
}

Employee const * Company::GetLongestServing(void) const
{
	auto minElem = min_element(m_Employees.cbegin(), m_Employees.cend(),
		[](const auto& lhs, const auto& rhs) { return lhs.second->GetDateJoined() < rhs.second->GetDateJoined();});

	if (minElem == m_Employees.end()) return nullptr;
	else return minElem->second;

}

std::ostream& Company::PrintDataSheet(std::ostream& ost) const
{

	// convert system clock.now to days -> this can be used in CTOR for year month day
	std::chrono::year_month_day date{ floor<std::chrono::days>(std::chrono::system_clock::now()) };

	if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;

	ost << hstar;
	ost << m_companyName << endl;
	ost << hstar;

	for_each(m_Employees.cbegin(), m_Employees.cend(), [&](const auto& e) { e.second->PrintDatasheet(ost);});

	ost << hline;
	ost << date.month() << "␣" << date.year() << endl;
	ost << hline;

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

	return ost;
}

Company::~Company()
{
	for (auto & elem : m_Employees)
	{
		delete elem.second;
	}

	m_Employees.clear();
}
```

6.7 TWorker.hpp

```
/*****************************************************************//**
 * \file   TWorker.hpp
 * \brief  Enum for indicating the worker Type
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#ifndef TWORKER_HPP
#define TWORKER_HPP

// changed naming convention because of
// name clashes with the actual classes
// that had the same name.
enum TWorker
{
    E_Boss,
    E_CommisionWorker,
    E_HourlyWorker,
    E_PieceWorker
};

#endif // !TWORKER_HPP
```

6.8 Employee.hpp

```
/*****************************************************************//**
 * \file   Employee.cpp
 * \brief  Abstract Class for constructing Employees of all types
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

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

using TDate = std::chrono::year_month_day;

class Employee : public Object
{
public:

    inline static const std::string ERROR_BAD_ID = "ERROR: An employees ID is limited to 3 characters.";
    inline static const std::string ERROR_BAD_SOZIAL_SEC_NUM = "ERROR: Invalid Sozial Security Number";

    /**
     * \brief Returns the ID of an Employee.
     *
     * \return String indication the ID
     */
    std::string GetID() const;

    /**
     * \brief Constructor needs every
     * member set to be called.
     * \return TWorker enum
     */
    Employee(
        const std::string &    name,
        const std::string &    nameID,
        const TDate       &    dateJoined,
        const TDate       &    TDateBirthdaydateBirth,
        const std::string &    socialSecurityNumber
    );

    /**
     * \brief Gives Information about what kind
     * of Worker it is.
     * \return TWorker enum
     */
    virtual TWorker GetWorkerType() const = 0;

    /** Pure Virtual Function
     * \brief return produced items.
     * \return size_t
     */
    virtual size_t GetProducedItems() const = 0;

    /** Pure Virtual Function
     * \brief returns sold items
     * \return size_t
     */
    virtual size_t GetSoldItems() const = 0;

    /** Pure Virtual Function
     * \brief returns total pay a worker
     * recieves.
     * \return size_t
     */
    virtual size_t GetSalary() const = 0;

    /**
     * \brief returns date of birth of a given worker.
     * \return TDate
     */
```

```
74      TDate GetDateBirth() const;
75
76      /**
77       * \brief returns the date a worker.
78       * has started working at the company.
79       * \return TDate
80       */
81      TDate GetDateJoined() const;
82
83      /**
84       * \brief Prints information about a worker.
85       * \return std::ostream&
86       */
87      std::ostream& PrintDatasheet(std::ostream& ost) const;
88
89
90
91      /** Pure virtual function
92      * \brief creates a copy of the worker and puts it on the heap.
93      * \return Employee*
94      */
95      virtual Employee* Clone() const = 0;
96
97  private:
98
99      /** Pure virtual function
100     * \brief Prints specific information for a type of worker.
101     * \return std::ostream&
102     */
103     virtual std::ostream& DoPrintSpecificData(std::ostream& ost) const = 0;
104
105
106     std::string m_name;
107     std::string m_nameIdentifier;
108     TDate m_dateJoined;
109     TDate m_dateBirth;
110     std::string m_socialSecurityNumber;
111
112     const size_t SozialSecNumLen = 4;
113 };
114
115 #endif // EMPLOYEE_H
```

6.9 Employee.cpp

```cpp
/*****************************************************************//**
 * \file   Employee.cpp
 * \brief  Abstract Class for constructing Employees of all types
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#include "Employee.hpp"
#include <cctype>
#include <algorithm>

Employee::Employee(
    const std::string &    name,
    const std::string &    nameID,
    const TDate &      dateJoined,
    const TDate &  dateBirth,
    const std::string  &   socialSecurityNumber
) : m_name{ name },
m_nameIdentifier{ nameID },
m_dateJoined{ dateJoined },
m_dateBirth{ dateBirth }
{
    if (nameID.length() != 3) throw ERROR_BAD_ID;

    if (! std::all_of(socialSecurityNumber.begin(), socialSecurityNumber.end(), ::isdigit))  throw ERROR_BAD_SOZIAL_SEC_NUM;

    if (! (socialSecurityNumber.size() == SozialSecNumLen) )  throw ERROR_BAD_SOZIAL_SEC_NUM;

    m_socialSecurityNumber = socialSecurityNumber;

}

std::string Employee::GetID() const
{
    return m_nameIdentifier;
}

TDate Employee::GetDateBirth() const
{
    return m_dateBirth;
}

TDate Employee::GetDateJoined() const
{
    return m_dateJoined;
}

std::ostream& Employee::PrintDatasheet(std::ostream& ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
    }

    ost << "Datenblatt\n---------------\n";
    ost << "Name:␣" << m_name << std::endl;
    ost << "Kuerzel:␣" << m_nameIdentifier << std::endl;
    ost << "Sozialversicherungsnummer:␣" << m_socialSecurityNumber;
    ost << m_dateBirth.day() << static_cast<unsigned>(m_dateBirth.month()) << static_cast<int>(m_dateBirth.year())%100 << std::endl;
    ost << "Geburtstag:␣" << m_dateBirth << std::endl;
    ost << "Einstiegsjahr:␣" << m_dateJoined.year() << std::endl;

    DoPrintSpecificData(ost);

    ost << std::endl;

    return ost;
}
```

6.10 Boss.hpp

```cpp
/*****************************************************************//**
 * \file   Boss.hpp
 * \brief  Boss Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#ifndef BOSS_H
#define BOSS_H

#include "Employee.hpp"

class Boss : public Employee
{
public:

    Boss(
        const std::string & name,
        const std::string & nameID,
        const TDate & dateJoined,
        const TDate & dateBirth,
        const std::string & socialSecurityNumber,
        const size_t & salary
    );


    /**
     * \brief Just here because of whacky class structure.
     * Worker does not strictly produce items!
     */
    size_t GetProducedItems() const override { return 0; };

    /**
     * \brief Just here because of whacky class structure.
     * Worker Does not sell items!
     */
    size_t GetSoldItems() const override { return 0; };

    /**
    * \brief Returns the total earnings for an
    * worker in this month.
    * \return size_t
    */
    size_t GetSalary() const override;

    /**
     * \brief Returns the type of worker.
     * \return TWorker
     */
    TWorker GetWorkerType() const override;

    /**
     * \brief Creates a clone on the Heap
     * and returns a pointer.
     * \return Employee*
     */
    Employee* Clone() const override;

private:
    /**
     * \brief Prints worker specific information
     * \param std::ostream& ost
     * \return std::ostream&
     */
    std::ostream& DoPrintSpecificData(std::ostream& ost) const override;

    size_t m_salary;
};

#endif // BOSS_H
```

6.11 Boss.cpp

```cpp
/*****************************************************************//**
 * \file   Boss.cpp
 * \brief  Boss Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#include "Boss.hpp"

Boss::Boss(
    const std::string & name,
    const std::string & nameID,
    const TDate & dateJoined,
    const TDate & dateBirth,
    const std::string & socialSecurityNumber,
    const size_t & salary
) :
    Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
    m_salary{ salary } {}

std::ostream& Boss::DoPrintSpecificData(std::ostream& ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
        return ost;
    }
    ost << "Role: Boss" << std::endl;
    ost << "Salary: " << m_salary << " EUR" << std::endl;

    return ost;
}

size_t Boss::GetSalary() const
{
    return m_salary;
}

TWorker Boss::GetWorkerType() const
{
    return E_Boss;
}

Employee* Boss::Clone() const
{
    return new Boss{ *this };
}
```

6.12 HourlyWorker.hpp

```cpp
/*****************************************************************//**
 * \file   HourlyWorker.hpp
 * \brief  HourlyWorker Class - Inherits from Employee
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#ifndef HOURLY_WORKER_HPP
#define HOURLY_WORKER_HPP

#include "Employee.hpp"

class HourlyWorker : public Employee
{
public:


    HourlyWorker(
        const std::string & name,
        const std::string & nameID,
        const TDate & dateJoined,
        const TDate & dateBirth,
        const std::string & socialSecurityNumber,
        const size_t & hourlyRate,
        const size_t & workedHours
    );


    /**
     * \brief Just here because of whacky class structure.
     * Worker does not strictly produce items!
     */
    size_t GetProducedItems() const override { return 0; };

    /**
     * \brief Just here because of whacky class structure.
     * Worker Does not sell items!
     */
    size_t GetSoldItems() const override { return 0; };

    /**
    * \brief Returns the total earnings for an
    * worker in this month.
    * \return size_t
    */
    size_t GetSalary() const override;

    /**
     * \brief Returns the type of worker.
     * \return TWorker
     */
    TWorker GetWorkerType() const override;

    /**
     * \brief Creates a clone on the Heap
     * and returns a pointer.
     * \return Employee*
     */
    Employee* Clone() const override;

private:
    /**
     * \brief Prints worker specific information
     * \param std::ostream& ost
     * \return std::ostream&
     */
    std::ostream& DoPrintSpecificData(std::ostream& ost) const override;

    size_t m_hourlyRate;
    size_t m_workedHours;
};

#endif // !HOURLY_WORKER_HPP
```

6.13 HourlyWorker.cpp

```cpp
/*****************************************************************//**
 * \file   HourlyWorker.hpp
 * \brief  HourlyWorker Class - Inherits from Employee
 * \author Simon
 * \date   October 2025
 *********************************************************************/

#include "HourlyWorker.hpp"

HourlyWorker::HourlyWorker(
    const std::string & name,
    const std::string & nameID,
    const TDate & dateJoined,
    const TDate & dateBirth,
    const std::string & socialSecurityNumber,
    const size_t & hourlyRate,
    const size_t & workedHours
) :
    Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
    m_hourlyRate{ hourlyRate },
    m_workedHours{ workedHours }
{}

std::ostream& HourlyWorker::DoPrintSpecificData(std::ostream& ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
        return ost;
    }
    ost << "Role:␣HourlyWWorker" << std::endl;
    ost << "Hourly␣rate:␣" << m_hourlyRate << "␣EUR" << std::endl;
    ost << "Hours␣worked:␣" << m_workedHours << "␣EUR" << std::endl;

    return ost;
}

size_t HourlyWorker::GetSalary() const
{
    return m_hourlyRate * m_workedHours;
}

TWorker HourlyWorker::GetWorkerType() const
{
    return E_HourlyWorker;
}

Employee* HourlyWorker::Clone() const
{
    return new HourlyWorker{*this};
}
```

6.14 PieceWorker.hpp

```cpp
/*****************************************************************//**
 * \file   PieceWorker.hpp
 * \brief  PieceWorker Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#ifndef PIECE_WORKER_H
#define PIECE_WORKER_H

#include "Employee.hpp"

class PieceWorker : public Employee
{
public:

    PieceWorker(
        const std::string & name,
        const std::string & nameID,
        const TDate & dateJoined,
        const TDate & dateBirth,
        const std::string & socialSecurityNumber,
        const size_t & m_numberPieces,
        const size_t & m_commisionPerPiece
    );

    /**
     * \brief Returns the number of pieces the
     * worker has produced
     */
    size_t GetProducedItems() const override;

    /**
     * \brief Just here because of whacky class structure.
     * Worker does not strictly sell items!
     */
    size_t GetSoldItems() const override { return 0; };

    /**
    * \brief Returns the total earnings for an
    * worker in this month.
    * \return size_t
    */
    size_t GetSalary() const override;

    /**
     * \brief Returns the type of worker.
     * \return TWorker
     */
    TWorker GetWorkerType() const override;

    /**
     * \brief Creates a clone on the Heap
     * and returns a pointer.
     * \return Employee*
     */
    Employee* Clone() const override;

private:
    /**
     * \brief Prints worker specific information
     * \param std::ostream& ost
     * \return std::ostream&
     */
    std::ostream& DoPrintSpecificData(std::ostream& ost) const override;

    size_t m_numberPieces;
    size_t m_commisionPerPiece;
};

#endif // !PIECE_WORKER_H
```

6.15 PieceWorker.cpp

```cpp
/*****************************************************************//**
 * \file   PieceWorker.cpp
 * \brief  PieceWorker Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#include "PieceWorker.hpp"

PieceWorker::PieceWorker(
    const std::string & name,
    const std::string & nameID,
    const TDate & dateJoined,
    const TDate & dateBirth,
    const std::string & socialSecurityNumber,
    const size_t & m_numberPieces,
    const size_t & m_commisionPerPiece
) :
    Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
    m_numberPieces{ m_numberPieces },
    m_commisionPerPiece{ m_commisionPerPiece }{}

std::ostream& PieceWorker::DoPrintSpecificData(std::ostream& ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
        return ost;
    }
    ost << "Role:␣PieceWorker" << std::endl;
    ost << "Pieces␣produced:␣" << m_numberPieces << std::endl;
    ost << "Pay␣per␣piece:␣" << m_commisionPerPiece << "␣EUR" << std::endl;

    return ost;
}

size_t PieceWorker::GetProducedItems() const
{
    return m_numberPieces;
}

size_t PieceWorker::GetSalary() const
{
    return m_numberPieces * m_commisionPerPiece;
}

TWorker PieceWorker::GetWorkerType() const
{
    return E_PieceWorker;
}

Employee* PieceWorker::Clone() const
{
    return new PieceWorker{ *this };
}
```

6.16 ComissionWorker.hpp

```cpp
/*****************************************************************//**
 * \file   ComissionWorker.hpp
 * \brief  ComissionWorker Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#ifndef COMISSION_WORKER_H
#define COMISSION_WORKER_H

#include "Employee.hpp"

class ComissionWorker : public Employee
{
public:

    ComissionWorker(
        const std::string & name,
        const std::string & nameID,
        const TDate & dateJoined,
        const TDate & dateBirth,
        const std::string & socialSecurityNumber,
        const size_t & baseSalary,
        const size_t & commisionPerPiece,
        const size_t & piecesSold
    );

    /**
     * \brief Just here because of whacky class structure.
     * Worker does not strictly produce items!
     */
    size_t GetProducedItems() const override { return 0; };

    /**
     * \brief returns how many items the commision worker has sold
     * \return size_t sold items
     */
    size_t GetSoldItems() const override;

    /**
    * \brief Returns the total earnings for an
    * worker in this month.
    * \return size_t
    */
    size_t GetSalary() const override;

    /**
     * \brief Returns the type of worker.
     * \return TWorker
     */
    TWorker GetWorkerType() const override;

    /**
     * \brief Creates a clone on the Heap
     * and returns a pointer.
     * \return Employee*
     */
    Employee* Clone() const override;

private:
    /**
     * \brief Prints worker specific information
     * \param std::ostream& ost
     * \return std::ostream&
     */
    std::ostream& DoPrintSpecificData(std::ostream& ost) const override;

    size_t m_baseSalary;
    size_t m_commisionPerPiece;
    size_t m_piecesSold;
};

#endif // !COMISSION_WORKER_H
```

6.17 ComissionWorker.cpp

```cpp
/*****************************************************************//**
 * \file   ComissionWorker.cpp
 * \brief  ComissionWorker Class - inherits from Employee
 * \author Simon Vogelhuber
 * \date   October 2025
 *********************************************************************/
#include "ComissionWorker.hpp"

ComissionWorker::ComissionWorker(
    const std::string & name,
    const std::string & nameID,
    const TDate & dateJoined,
    const TDate & dateBirth,
    const std::string & socialSecurityNumber,
    const size_t & baseSalary,
    const size_t & commisionPerPiece,
    const size_t & piecesSold
) :
    Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
    m_baseSalary{ baseSalary },
    m_commisionPerPiece{ commisionPerPiece },
    m_piecesSold { piecesSold }
{}

std::ostream& ComissionWorker::DoPrintSpecificData(std::ostream & ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
        return ost;
    }
    ost << "Role: ComissionWorker" << std::endl;
    ost << "Base salary: " << m_baseSalary << " EUR" << std::endl;
    ost << "Comission per piece: " << m_commisionPerPiece << " EUR" << std::endl;
    ost << "Pieces sold: " << m_piecesSold << std::endl;

    return ost;
}

size_t ComissionWorker::GetSoldItems() const
{
    return m_piecesSold;
}

size_t ComissionWorker::GetSalary() const
{
    return m_baseSalary + m_piecesSold * m_commisionPerPiece;
}

TWorker ComissionWorker::GetWorkerType() const
{
    return E_CommisionWorker;
}

Employee* ComissionWorker::Clone() const
{
    return new ComissionWorker{ *this };
}
```

6.18 main.cpp

```
/*****************************************************************//**
 * \file   main.cpp
 * \brief  Testdriver for the Company
 *
 * \author Simon
 * \date   October 2025
 *********************************************************************/
#include "Company.hpp"
#include "Employee.hpp"
#include "HourlyWorker.hpp"
#include "vld.h"
#include "Client.hpp"
#include "Test.hpp"
#include "ComissionWorker.hpp"
#include "HourlyWorker.hpp"
#include "Boss.hpp"
#include "PieceWorker.hpp"
#include <iostream>
#include <fstream>
#include <cassert>

using namespace std;
using namespace std::chrono;

static bool TestEmployeeBoss(std::ostream& ost);
static bool TestEmployeeHourlyWorker(std::ostream& ost);
static bool TestEmployeePieceWorker(std::ostream& ost);
static bool TestEmployeeComissionWorker(std::ostream& ost);
static bool TestCompanyAdd(std::ostream& ost);

#define WRITE_OUTPUT true

int main(void){
        bool TestOK = true;
        ofstream testoutput;
        try {

                if (WRITE_OUTPUT == true) {
                        testoutput.open("TestOutput.txt");
                }

                Company comp{ "Offenberger Devices" };
                Client TestClient;
                ComissionWorker* cWork = new ComissionWorker{ "Simon 1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 2500 };
                ComissionWorker* cWork2 = new ComissionWorker{ "Simon 6", "Si6", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 200 };
                HourlyWorker* hWork = new HourlyWorker{ "Simon 2", "Si2", { 2022y,November,23d }, { 1934y,November,23d },"4712",20,25 };
                Boss* boss = new Boss{ "Simon 3", "Si3", { 2000y,November,23d }, { 1950y,November,23d },"4712",35000 };
                PieceWorker* pWork = new PieceWorker{ "Simon 4", "Si4", { 2022y,November,23d }, { 2010y,November,23d },"4712",25,25 };
                PieceWorker* pWork2 = new PieceWorker{ "Simon 5", "Si5", { 2022y,November,23d }, { 2011y,November,23d },"4712",25,25 };

                comp.AddEmployee(cWork);
                comp.AddEmployee(cWork2);
                comp.AddEmployee(hWork);
                comp.AddEmployee(boss);
                comp.AddEmployee(pWork);
                comp.AddEmployee(pWork2);

                TestOK = TestOK && TestClient.TestCompanyGetter(cout, comp);
                if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyGetter(testoutput, comp);

                // Copy Ctor Call !
                Company compCopy = comp;

                TestOK = TestOK && TestClient.TestCompanyCopyCTOR(cout, comp, compCopy);
                if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyCopyCTOR(testoutput, comp, compCopy);

                // Test Assign Operator
                Company compAss{ "Assign Company" };
                compAss = comp;

                TestOK = TestOK && TestClient.TestCompanyAssignOp(cout, comp, compAss);
                if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyAssignOp(testoutput, comp, compAss);

```

```
74
75                  TestOK = TestOK && TestClient.TestCompanyPrint(cout, comp);
76                  if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyPrint(testoutput, comp);
77
78                  Company emptyComp{ "empty" };
79
80                  TestOK = TestOK && TestClient.TestEmptyCompanyGetter(cout, emptyComp);
81                  if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmptyCompanyGetter(testoutput, emptyComp);
82
83                  // Test Boss
84                  TestOK = TestOK && TestEmployeeBoss(cout);
85                  if (WRITE_OUTPUT) TestOK = TestOK && TestEmployeeBoss(testoutput);
86
87                  // Test Hourly Worker
88                  TestOK = TestOK && TestEmployeeHourlyWorker(cout);
89                  if (WRITE_OUTPUT) TestOK = TestOK && TestEmployeeHourlyWorker(testoutput);
90
91                  // Test Piece Worker
92                  TestOK = TestOK && TestEmployeePieceWorker(cout);
93                  if (WRITE_OUTPUT) TestOK = TestOK && TestEmployeePieceWorker(testoutput);
94
95                  // Test Comission Worker
96                  TestOK = TestOK && TestEmployeeComissionWorker(cout);
97                  if (WRITE_OUTPUT) TestOK = TestOK && TestEmployeeComissionWorker(testoutput);
98
99                  // Test Company Add
100                 TestOK = TestOK && TestCompanyAdd(cout);
101                 if (WRITE_OUTPUT) TestOK = TestOK && TestCompanyAdd(testoutput);
102
103                 if (WRITE_OUTPUT) {
104                         if (TestOK) TestCaseOK(testoutput);
105                         else TestCaseFail(testoutput);
106
107                         testoutput.close();
108                 }
109
110                 if (TestOK) TestCaseOK(cout);
111                 else TestCaseFail(cout);
112         }
113         catch (const string& err) {
114                 cout << err;
115         }
116         catch (bad_alloc const& error) {
117                 cout << error.what();
118         }
119         catch (const exception& err) {
120                 cout  << err.what();
121         }
122         catch (...) {
123                 cout << "Unhandelt Exception";
124         }
125
126
127
128 }
129
130
131
132 static bool TestEmployeeBoss(std::ostream& ost)
133 {
134
135         assert(ost.good());
136
137         TestStart(ost);
138
139         bool TestOK = true;
140         string error_msg = "";
141
142         try {
143                 size_t testSalary = 7800;
144                 string svr = "4711";
145                 TDate dateBorn = { 2000y,November,22d };
146                 TDate dateJoined = { 2022y,November,23d };
147                 string name = "Max Musterman";
148                 string id = "MAX";
149
```

```
150             Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary };
151
152             TestOK = TestOK && check_dump(ost, "Test - Boss.GetSalary()", testSalary, testBoss.GetSalary());
153             TestOK = TestOK && check_dump(ost, "Test - Boss.GetSoldItems()", static_cast<size_t>(0), testBoss.GetSoldItems());
154             TestOK = TestOK && check_dump(ost, "Test - Boss.GetProducedItems()", static_cast<size_t>(0), testBoss.GetProducedItems());
155             TestOK = TestOK && check_dump(ost, "Test - Boss.GetWorkerType()", E_Boss, testBoss.GetWorkerType());
156             TestOK = TestOK && check_dump(ost, "Test - Boss.GetDateBirth()", dateBorn, testBoss.GetDateBirth());
157             TestOK = TestOK && check_dump(ost, "Test - Boss.GetDateJoined()", dateJoined, testBoss.GetDateJoined());
158         }
159         catch (const string& err) {
160             error_msg = err;
161         }
162         catch (bad_alloc const& error) {
163             error_msg = error.what();
164         }
165         catch (const exception& err) {
166             error_msg = err.what();
167         }
168         catch (...) {
169             error_msg = "Unhandelt Exception";
170         }
171
172         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
173         error_msg.clear();
174
175         //clone test
176         try {
177             size_t testSalary = 7800;
178             string svr = "4711";
179             TDate dateBorn = { 2000y,November,22d };
180             TDate dateJoined = { 2022y,November,23d };
181             string name = "Max Musterman";
182             string id = "MAX";
183
184             Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary };
185             Employee* pEmp = testBoss.Clone();
186             TestOK = TestOK && check_dump(ost, "Test Boss.Clone()", pEmp != nullptr && pEmp != &testBoss, true);
187             delete pEmp;
188         }
189         catch (const string& err) {
190             error_msg = err;
191         }
192         catch (bad_alloc const& error) {
193             error_msg = error.what();
194         }
195         catch (const exception& err) {
196             error_msg = err.what();
197         }
198         catch (...) {
199             error_msg = "Unhandelt Exception";
200         }
201
202         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
203         error_msg.clear();
204
205         // Unavialable ID
206         try {
207             size_t testSalary = 7800;
208             string svr = "4711";
209             TDate dateBorn = { 2000y,November,22d };
210             TDate dateJoined = { 2022y,November,23d };
211             string name = "Max Musterman";
212             string id = "MAXL";
213
214             Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary };
215         }
216         catch (const string& err) {
217             error_msg = err;
218         }
219         catch (bad_alloc const& error) {
220             error_msg = error.what();
221         }
222         catch (const exception& err) {
223             error_msg = err.what();
224         }
225         catch (...) {
```

```
226                 error_msg = "Unhandelt Exception";
227         }
228
229         TestOK = TestOK && check_dump(ost, "Boss Constructor bad ID", error_msg, Employee::ERROR_BAD_ID);
230         error_msg.clear();
231
232         // Constructor bad SV
233         try {
234                 size_t testSalary = 7800;
235                 string svr = "ARGH";
236                 TDate dateBorn = { 2000y,November,22d };
237                 TDate dateJoined = { 2022y,November,23d };
238                 string name = "Max Musterman";
239                 string id = "MAX";
240
241                 Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary };
242         }
243         catch (const string& err) {
244                 error_msg = err;
245         }
246         catch (bad_alloc const& error) {
247                 error_msg = error.what();
248         }
249         catch (const exception& err) {
250                 error_msg = err.what();
251         }
252         catch (...) {
253                 error_msg = "Unhandelt Exception";
254         }
255
256         TestOK = TestOK && check_dump(ost, "Boss Constructor bad SV - invalid character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
257
258         error_msg.clear();
259
260
261         // Constructor bad SV - too many nums
262         try {
263                 size_t testSalary = 7800;
264                 string svr = "ARGH";
265                 TDate dateBorn = { 2000y,November,22d };
266                 TDate dateJoined = { 2022y,November,23d };
267                 string name = "Max Musterman";
268                 string id = "MAX";
269
270                 Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary };
271         }
272         catch (const string& err) {
273                 error_msg = err;
274         }
275         catch (bad_alloc const& error) {
276                 error_msg = error.what();
277         }
278         catch (const exception& err) {
279                 error_msg = err.what();
280         }
281         catch (...) {
282                 error_msg = "Unhandelt Exception";
283         }
284
285         TestOK = TestOK && check_dump(ost, "Boss Constructor bad SV - too many nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
286         error_msg.clear();
287
288         TestEnd(ost);
289         return TestOK;
290 }
291
292 static bool TestEmployeeHourlyWorker(std::ostream& ost)
293 {
294         assert(ost.good());
295
296         TestStart(ost);
297
298         bool TestOK = true;
299         string error_msg = "";
300
301         try {
```

```
302             size_t hourlyRate = 21;
303             size_t workedHours = 160;
304             string svr = "4711";
305             TDate dateBorn = { 2000y,November,22d };
306             TDate dateJoined = { 2022y,November,23d };
307             string name = "Max Musterman";
308             string id = "MAX";
309
310             HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
311
312             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetSalary()", hourlyRate * workedHours, testHourlyWorker.GetSalary());
313             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
314             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProducedItems());
315             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetWorkerType()", E_HourlyWorker, testHourlyWorker.GetWorkerType());
316             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
317             TestOK = TestOK && check_dump(ost, "Test - HourlyWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
318     }
319     catch (const string& err) {
320             error_msg = err;
321     }
322     catch (bad_alloc const& error) {
323             error_msg = error.what();
324     }
325     catch (const exception& err) {
326             error_msg = err.what();
327     }
328     catch (...) {
329             error_msg = "Unhandelt Exception";
330     }
331
332     TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
333     error_msg.clear();
334
335     //clone test
336     try {
337             size_t hourlyRate = 21;
338             size_t workedHours = 160;
339             string svr = "4711";
340             TDate dateBorn = { 2000y,November,22d };
341             TDate dateJoined = { 2022y,November,23d };
342             string name = "Max Musterman";
343             string id = "MAX";
344
345             HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
346
347             Employee* pEmp = testHourlyWorker.Clone();
348             TestOK = TestOK && check_dump(ost, "Test testPieceWorker.Clone()", pEmp != nullptr && pEmp != &testHourlyWorker, true);
349             delete pEmp;
350     }
351     catch (const string& err) {
352             error_msg = err;
353     }
354     catch (bad_alloc const& error) {
355             error_msg = error.what();
356     }
357     catch (const exception& err) {
358             error_msg = err.what();
359     }
360     catch (...) {
361             error_msg = "Unhandelt Exception";
362     }
363
364     TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
365     error_msg.clear();
366
367     // Unavialable ID
368     try {
369             size_t hourlyRate = 21;
370             size_t workedHours = 160;
371             string svr = "4711";
372             TDate dateBorn = { 2000y,November,22d };
373             TDate dateJoined = { 2022y,November,23d };
374             string name = "Max Musterman";
375             string id = "MAXL";
376
377             HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
```

```
	}
	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";
	}

	TestOK = TestOK && check_dump(ost, "HourlyWorker Constructor bad ID", error_msg, Employee::ERROR_BAD_ID);
	error_msg.clear();

	// Constructor bad SV
	try {
		size_t hourlyRate = 21;
		size_t workedHours = 160;
		string svr = "ARGH";
		TDate dateBorn = { 2000y,November,22d };
		TDate dateJoined = { 2022y,November,23d };
		string name = "Max Musterman";
		string id = "MAX";

		HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
	}
	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";
	}

	TestOK = TestOK && check_dump(ost, "HourlyWorker Constructor bad SV - invalid character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);

	error_msg.clear();

	// Constructor bad SV - too many nums
	try {
		size_t hourlyRate = 21;
		size_t workedHours = 160;
		string svr = "ARGH";
		TDate dateBorn = { 2000y,November,22d };
		TDate dateJoined = { 2022y,November,23d };
		string name = "Max Musterman";
		string id = "MAX";

		HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
	}
	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";
	}

	TestOK = TestOK && check_dump(ost, "HourlyWorker Constructor bad SV - too many nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
	error_msg.clear();

	TestEnd(ost);
	return TestOK;
```

```
454 }
455
456 static bool TestEmployeePieceWorker(std::ostream& ost)
457 {
458         assert(ost.good());
459
460         TestStart(ost);
461
462         bool TestOK = true;
463         string error_msg = "";
464
465         try {
466                 size_t piecesProduced = 950;
467                 size_t comissionPerPiece = 2;
468                 string svr = "4711";
469                 TDate dateBorn = { 2000y,November,22d };
470                 TDate dateJoined = { 2022y,November,23d };
471                 string name = "Max Musterman";
472                 string id = "MAX";
473
474                 PieceWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
475
476                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetSalary()", piecesProduced * comissionPerPiece, testHourlyWorker.GetSalary());
477                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
478                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetProducedItems()", piecesProduced, testHourlyWorker.GetProducedItems());
479                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetWorkerType()", E_PieceWorker, testHourlyWorker.GetWorkerType());
480                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
481                 TestOK = TestOK && check_dump(ost, "Test - PieceWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
482         }
483         catch (const string& err) {
484                 error_msg = err;
485         }
486         catch (bad_alloc const& error) {
487                 error_msg = error.what();
488         }
489         catch (const exception& err) {
490                 error_msg = err.what();
491         }
492         catch (...) {
493                 error_msg = "Unhandelt Exception";
494         }
495
496         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
497         error_msg.clear();
498
499         //clone test
500         try {
501                 size_t piecesProduced = 950;
502                 size_t comissionPerPiece = 2;
503                 string svr = "4711";
504                 TDate dateBorn = { 2000y,November,22d };
505                 TDate dateJoined = { 2022y,November,23d };
506                 string name = "Max Musterman";
507                 string id = "MAX";
508
509                 PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
510                 Employee* pEmp = testPieceWorker.Clone();
511                 TestOK = TestOK && check_dump(ost, "Test testPieceWorker.Clone()", pEmp != nullptr && pEmp != &testPieceWorker, true);
512                 delete pEmp;
513         }
514         catch (const string& err) {
515                 error_msg = err;
516         }
517         catch (bad_alloc const& error) {
518                 error_msg = error.what();
519         }
520         catch (const exception& err) {
521                 error_msg = err.what();
522         }
523         catch (...) {
524                 error_msg = "Unhandelt Exception";
525         }
526
527         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
528         error_msg.clear();
529
```

```
530         // Unavialable ID
531         try {
532                 size_t piecesProduced = 950;
533                 size_t comissionPerPiece = 2;
534                 string svr = "4711";
535                 TDate dateBorn = { 2000y,November,22d };
536                 TDate dateJoined = { 2022y,November,23d };
537                 string name = "Max␣Musterman";
538                 string id = "MAXL";
539 
540                 PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
541         }
542         catch (const string& err) {
543                 error_msg = err;
544         }
545         catch (bad_alloc const& error) {
546                 error_msg = error.what();
547         }
548         catch (const exception& err) {
549                 error_msg = err.what();
550         }
551         catch (...) {
552                 error_msg = "Unhandelt␣Exception";
553         }
554 
555         TestOK = TestOK && check_dump(ost, "PieceWorker␣Constructor␣bad␣ID", error_msg, Employee::ERROR_BAD_ID);
556         error_msg.clear();
557 
558         // Constructor bad SV
559         try {
560                 size_t piecesProduced = 950;
561                 size_t comissionPerPiece = 2;
562                 string svr = "ARGH";
563                 TDate dateBorn = { 2000y,November,22d };
564                 TDate dateJoined = { 2022y,November,23d };
565                 string name = "Max␣Musterman";
566                 string id = "MAX";
567 
568                 PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
569         }
570         catch (const string& err) {
571                 error_msg = err;
572         }
573         catch (bad_alloc const& error) {
574                 error_msg = error.what();
575         }
576         catch (const exception& err) {
577                 error_msg = err.what();
578         }
579         catch (...) {
580                 error_msg = "Unhandelt␣Exception";
581         }
582 
583         TestOK = TestOK && check_dump(ost, "PieceWorker␣Constructor␣bad␣SV␣-␣invalid␣character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
584 
585         error_msg.clear();
586 
587         // Constructor bad SV - too many nums
588         try {
589                 size_t piecesProduced = 950;
590                 size_t comissionPerPiece = 2;
591                 string svr = "ARGH";
592                 TDate dateBorn = { 2000y,November,22d };
593                 TDate dateJoined = { 2022y,November,23d };
594                 string name = "Max␣Musterman";
595                 string id = "MAX";
596 
597                 PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
598         }
599         catch (const string& err) {
600                 error_msg = err;
601         }
602         catch (bad_alloc const& error) {
603                 error_msg = error.what();
604         }
605         catch (const exception& err) {
```

```
606                 error_msg = err.what();
607         }
608         catch (...) {
609                 error_msg = "Unhandelt Exception";
610         }
611
612         TestOK = TestOK && check_dump(ost, "PieceWorker Constructor bad SV - too many nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
613         error_msg.clear();
614
615
616         TestEnd(ost);
617         return TestOK;
618 }
619
620 static bool TestEmployeeComissionWorker(std::ostream& ost)
621 {
622         assert(ost.good());
623
624         TestStart(ost);
625
626         bool TestOK = true;
627         string error_msg = "";
628
629         try {
630                 size_t baseSalary = 2300;
631                 size_t piecesSold = 300;
632                 size_t comissionPerPiece = 2;
633                 string svr = "4711";
634                 TDate dateBorn = { 2000y,November,22d };
635                 TDate dateJoined = { 2022y,November,23d };
636                 string name = "Max Musterman";
637                 string id = "MAX";
638
639                 ComissionWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
640
641                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetSalary()", baseSalary + piecesSold * comissionPerPiece, testHourlyWorker.GetSalary());
642                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetSoldItems()", piecesSold, testHourlyWorker.GetSoldItems());
643                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProducedItems());
644                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetWorkerType()", E_CommisionWorker, testHourlyWorker.GetWorkerType());
645                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
646                 TestOK = TestOK && check_dump(ost, "Test - ComissionWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
647         }
648         catch (const string& err) {
649                 error_msg = err;
650         }
651         catch (bad_alloc const& error) {
652                 error_msg = error.what();
653         }
654         catch (const exception& err) {
655                 error_msg = err.what();
656         }
657         catch (...) {
658                 error_msg = "Unhandelt Exception";
659         }
660
661         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
662         error_msg.clear();
663
664         //clone test
665         try {
666                 size_t baseSalary = 2300;
667                 size_t piecesSold = 300;
668                 size_t comissionPerPiece = 2;
669                 string svr = "4711";
670                 TDate dateBorn = { 2000y,November,22d };
671                 TDate dateJoined = { 2022y,November,23d };
672                 string name = "Max Musterman";
673                 string id = "MAX";
674
675                 ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
676                 Employee* pEmp = testComissionWorker.Clone();
677                 TestOK = TestOK && check_dump(ost, "Test testPieceWorker.Clone()", pEmp != nullptr && pEmp != &testComissionWorker, true);
678                 delete pEmp;
679         }
680         catch (const string& err) {
681                 error_msg = err;
```

```
682         }
683         catch (bad_alloc const& error) {
684                 error_msg = error.what();
685         }
686         catch (const exception& err) {
687                 error_msg = err.what();
688         }
689         catch (...) {
690                 error_msg = "Unhandelt Exception";
691         }
692
693         TestOK = TestOK && check_dump(ost, "Test - error buffer", error_msg.empty(), true);
694         error_msg.clear();
695
696         // Unavialable ID
697         try {
698                 size_t baseSalary = 2300;
699                 size_t piecesSold = 300;
700                 size_t comissionPerPiece = 2;
701                 string svr = "4711";
702                 TDate dateBorn = { 2000y,November,22d };
703                 TDate dateJoined = { 2022y,November,23d };
704                 string name = "Max Musterman";
705                 string id = "MAXL";
706
707                 ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
708         }
709         catch (const string& err) {
710                 error_msg = err;
711         }
712         catch (bad_alloc const& error) {
713                 error_msg = error.what();
714         }
715         catch (const exception& err) {
716                 error_msg = err.what();
717         }
718         catch (...) {
719                 error_msg = "Unhandelt Exception";
720         }
721
722         TestOK = TestOK && check_dump(ost, "ComissionWorker Constructor bad ID ", error_msg, Employee::ERROR_BAD_ID);
723         error_msg.clear();
724
725         // Constructor bad SV - no numbers
726         try {
727                 size_t baseSalary = 2300;
728                 size_t piecesSold = 300;
729                 size_t comissionPerPiece = 2;
730                 string svr = "ARGH";
731                 TDate dateBorn = { 2000y,November,22d };
732                 TDate dateJoined = { 2022y,November,23d };
733                 string name = "Max Musterman";
734                 string id = "MAX";
735
736                 ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
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, "ComissionWorker Constructor bad SV - invalid character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
752
753         error_msg.clear();
754
755         // Constructor bad SV - too many nums
756         try {
757                 size_t baseSalary = 2300;
```

```
758                 size_t piecesSold = 300;
759                 size_t comissionPerPiece = 2;
760                 string svr = "47488888239874";
761                 TDate dateBorn = { 2000y,November,22d };
762                 TDate dateJoined = { 2022y,November,23d };
763                 string name = "Max Musterman";
764                 string id = "MAX";
765
766                 ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
767         }
768         catch (const string& err) {
769                 error_msg = err;
770         }
771         catch (bad_alloc const& error) {
772                 error_msg = error.what();
773         }
774         catch (const exception& err) {
775                 error_msg = err.what();
776         }
777         catch (...) {
778                 error_msg = "Unhandelt Exception";
779         }
780
781         TestOK = TestOK && check_dump(ost, "ComissionWorker Constructor bad SV - too many nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
782
783         error_msg.clear();
784
785         TestEnd(ost);
786         return TestOK;
787 }
788
789
790 static bool TestCompanyAdd(std::ostream& ost)
791 {
792         assert(ost.good());
793
794         TestStart(ost);
795
796         bool TestOK = true;
797         string error_msg = "";
798
799         try {
800
801                 ComissionWorker* cWork = new ComissionWorker{ "Simon 1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 2500 };
802
803                 Company comp{"Dup"};
804                 comp.AddEmployee(cWork);
805                 comp.AddEmployee(cWork);
806         }
807         catch (const string& err) {
808                 error_msg = err;
809         }
810         catch (bad_alloc const& error) {
811                 error_msg = error.what();
812         }
813         catch (const exception& err) {
814                 error_msg = err.what();
815         }
816         catch (...) {
817                 error_msg = "Unhandelt Exception";
818         }
819
820         TestOK = TestOK && check_dump(ost, "Test Exception in Company Add Duplicate", Company::ERROR_DUPLICATE_EMPL, error_msg);
821         error_msg.clear();
822
823         TestEnd(ost);
824         return TestOK;
825 }
```

6.19 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
	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 << expected << "␣==" << "␣Result:␣" << result << ")"  << std::
                                noboolalpha << std::endl << std::endl;
111                 }
112                 else {
113                         ostr << testcase << std::endl << colorRed << "[Test␣FAILED]␣" << colorWhite << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Result:␣" << result << ")"  << 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:␣" << result << ")" << std::noboolalpha << std::endl <<
                                std::endl;
118                 }
119                 else {
120                         ostr << testcase << std::endl  << "[Test␣FAILED]␣"  << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Result:␣" << result << ")" << std::noboolalpha << std::
                                endl << std::endl;
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
```