

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!



HSD

FH-HAGENBERG

Systemdokumentation Projekt Gehaltsberechnung

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 30. 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)	11
5	Testprotokollierung	12
6	Quellcode	21
6.1	Object.hpp	21
6.2	Client.hpp	22
6.3	Client.cpp	24
6.4	IComp.hpp	28
6.5	Company.hpp	30
6.6	Company.cpp	32
6.7	TWorker.hpp	34
6.8	Employee.hpp	35
6.9	Employee.cpp	37
6.10	Boss.hpp	38
6.11	Boss.cpp	39
6.12	HourlyWorker.hpp	40
6.13	HourlyWorker.cpp	41
6.14	PieceWorker.hpp	42
6.15	PieceWorker.cpp	43
6.16	ComissionWorker.hpp	44
6.17	ComissionWorker.cpp	45
6.18	main.cpp	46
6.19	Test.hpp	59

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 8,5 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

- Abfrage nach der Anzahl der Mitarbeiter.
- Abfrage 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

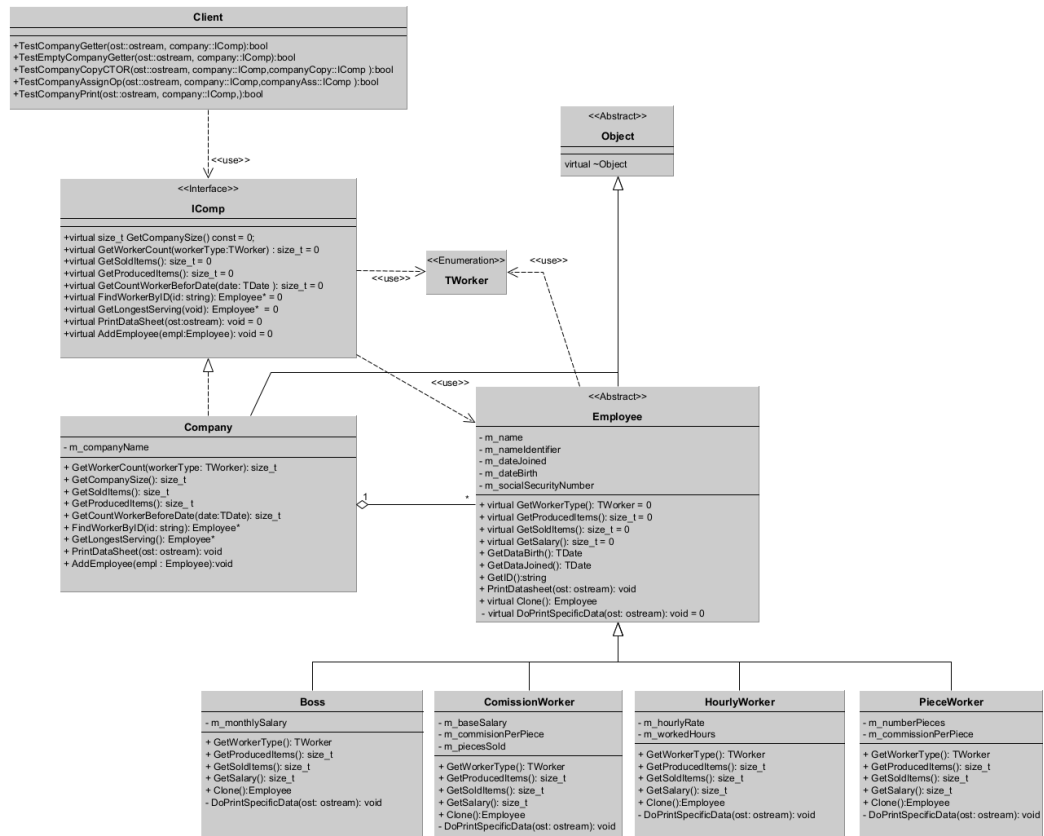
Funktionen der Mitarbeiter

- 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. Weiters wurden die konkreten Mitarbeiterklassen so gestaltet, dass sie ohne großen Aufwand zu testen sind. Aus diesem Grund werden alle Daten im Konstruktor übergeben und es gibt keine Setter-Methoden. Würde dieses Design in der Praxis verwendet werden, müsste man noch Setter Methoden hinzufügen. Da dies hier nicht im Fokus steht, wurde dies nicht umgesetzt.

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



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

```
247
248 Test - error buffer
249 [Test OK] Result: (Expected: true == Result: true)
250
251 PieceWorker Constructor bad ID
252 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪ to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
253
254 PieceWorker Constructor bad SV - invalid character
255 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
256
257 PieceWorker Constructor bad SV - too many nums
258 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
259
260 PieceWorker bad ostream
261 [Test OK] Result: (Expected: ERROR: Provided Ostream is bad ==
    ↪ Result: ERROR: Provided Ostream is bad)
262
263
264 *****
265
266
267 *****
268 TESTCASE START
269 *****
270
271 Test - ComissionWorker.GetSalary()
272 [Test OK] Result: (Expected: 2900 == Result: 2900)
273
274 Test - ComissionWorker.GetSoldItems()
275 [Test OK] Result: (Expected: 300 == Result: 300)
276
277 Test - ComissionWorker.GetProducedItems()
278 [Test OK] Result: (Expected: 0 == Result: 0)
279
280 Test - ComissionWorker.GetWorkerType()
281 [Test OK] Result: (Expected: 1 == Result: 1)
282
283 Test - ComissionWorker.GetDateBirth()
284 [Test OK] Result: (Expected: 2000-11-22 == Result: 2000-11-22)
285
```

```
286 Test - ComissionWorker.GetDateJoined()
287 [Test OK] Result: (Expected: 2022-11-23 == Result: 2022-11-23)
288
289 Test - error buffer
290 [Test OK] Result: (Expected: true == Result: true)
291
292 Test testPieceWorker.Clone()
293 [Test OK] Result: (Expected: true == Result: true)
294
295 Test - error buffer
296 [Test OK] Result: (Expected: true == Result: true)
297
298 ComissionWorker Constructor bad ID
299 [Test OK] Result: (Expected: ERROR: An employees ID is limited
    ↪ to 3 characters. == Result: ERROR: An employees ID is
    ↪ limited to 3 characters.)
300
301 ComissionWorker Constructor bad SV - invalid character
302 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
303
304 ComissionWorker Constructor bad SV - too many nums
305 [Test OK] Result: (Expected: ERROR: Invalid Sozial Security
    ↪ Number == Result: ERROR: Invalid Sozial Security Number)
306
307 ComissionWorker bad ostream
308 [Test OK] Result: (Expected: ERROR: Provided Ostream is bad ==
    ↪ Result: ERROR: Provided Ostream is bad)
309
310
311 *****
312
313
314 *****
315 TESTCASE START
316 *****
317
318 Test Exception in Company Add Duplicate
319 [Test OK] Result: (Expected: ERROR: Duplicate Employee! ==
    ↪ Result: ERROR: Duplicate Employee!)
320
321
322 *****
323
```

324 | TEST OK!!

6 Quellcode

6.1 Object.hpp

```
1  /*****
2  * \file   Object.hpp
3  * \brief  Root of all Objects
4  *
5  * \author Simon
6  * \date   October 2025
7  *****/
8  #ifndef OBJECT_HPP
9  #define OBJECT_HPP
10
11 class Object {
12 public:
13
14     /**
15     * \brief Constant for Exception Bad Ostream.
16     */
17     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";
18
19     /**
20     * \brief Constant for Exception Fail Write.
21     */
22     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";
23
24     /**
25     * \brief Constant for Exception Nullptr.
26     */
27     inline static const std::string ERROR_NULLPTR = "ERROR:_Passed_in_Nullptr!";
28
29 protected:
30
31     /**
32     * \brief protected CTOR -> abstract Object.
33     *
34     */
35     Object() = default;
36
37     /**
38     * \brief virtual DTOR -> once Virtual always virtual.
39     *
40     */
41     virtual ~Object() = default;
42 };
43
44 #endif // !OBJECT_HPP
45
```

6.2 Client.hpp

```
1  /*****
2  * \file   Client.hpp
3  * \brief  Client Class that uses the Class Company via the Interface IComp
4  *
5  * \author Simon Offenberger
6  * \date   October 2025
7  *****/
8  #ifndef CLIENT_HPP
9  #define CLIENT_HPP
10
11 #include <iostream>
12 #include "IComp.hpp"
13
14 class Client {
15 public:
16     /**
17     * Constant for Exception Bad Ostream.
18     */
19     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";
20
21     /**
22     * Constant for Exception Write Fail.
23     */
24     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";
25
26     /**
27     * \brief Test Methode for the Getter Methodes of the Company via the Interface.
28     *
29     * \param ost Refernce to an ostream where the Test results should be printed at
30     * \param company Reference to a company interface
31     * \return true -> Test OK
32     * \return false -> Test NOK
33     */
34     bool TestCompanyGetter(std::ostream & ost, const IComp& company) const;
35
36     /**
37     * \brief Test Methode for the Getter Methodes of an Empty Company via the Interface.
38     *
39     * \param ost Refernce to an ostream where the Test results should be printed at
40     * \param company Reference to a company interface
41     * \return true -> Test OK
42     * \return false -> Test NOK
43     */
44     bool TestEmptyCompanyGetter(std::ostream & ost, IComp& company) const;
45
46     /**
47     * \brief Test Methode for testing the Copy Ctor of the Company
48     *
49     * \param ost Refernce to an ostream where the Test results should be printed at
50     * \param company Reference to a company interface
51     * \param companyCopy Reference to the copy of company
52     * \return true -> Test OK
53     * \return false -> Test NOK
54     */
55     bool TestCompanyCopyCTOR(std::ostream & ost, const IComp& company, const IComp& companyCopy) const;
56
57     /**
58     * \brief Test Methode for the Assign Operator of Company
59     *
60     * \param ost Refernce to an ostream where the Test results should be printed at
61     * \param company Reference to a company interface
62     * \param companyAss Reference to the assigned Company should be Equal to company
63     * \return true -> Test OK
64     * \return false -> Test NOK
65     */
66     bool TestCompanyAssignOp(std::ostream & ost, const IComp& company, const IComp& companyAss) const;
67
68     /**
69     * \brief Test Methode for the Print Methode of Company
70     *
71     * \param ost Refernce to an ostream where the Test results should be printed at
72     * \param company Reference to a company interface
73     * \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

```
1  /*****
2  * \file   Client.hpp
3  * \brief  Client Class that uses the Class Company via the Interface IComp
4  *
5  * \author Simon Offenberger
6  * \date   October 2025
7  *****/
8
9  #include "Client.hpp"
10 #include "Test.hpp"
11 #include <sstream>
12 #include <fstream>
13
14 using namespace std;
15 using namespace std::chrono;
16
17 bool Client::TestCompanyGetter(std::ostream& ost, const IComp & company) const
18 {
19     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
20
21     TestStart(ost);
22
23     bool TestOK = true;
24     string error_msg = "";
25
26     try {
27
28         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Commission_Worker_Cnt_&_Add_Empl", static_cast<size_t>(2), company.GetWorkerCount(TWorker::E_CommissionWorker));
29         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Houerly_Worker_Cnt_&_Add_Empl", static_cast<size_t>(1), company.GetWorkerCount(TWorker::E_HourlyWorker));
30         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Boss_Cnt_&_Add_Empl", static_cast<size_t>(1), company.GetWorkerCount(TWorker::E_Boss));
31         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Piece_Worker_Cnt_&_Add_Empl", static_cast<size_t>(2), company.GetWorkerCount(TWorker::E_PieceWorker));
32
33
34         TestOK = TestOK && check_dump(ost, "Test_Company_FindWorker_by_ID", static_cast<std::string>("Si1"), company.FindWorkerByID("Si1")->GetID());
35         TestOK = TestOK && check_dump(ost, "Test_Company_FindWorker_by_empty_ID", static_cast<const Employee *>(nullptr), company.FindWorkerByID(""));
36
37         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Size", static_cast<size_t>(6), company.GetCompanySize());
38
39         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Count_worker_bevor_1930_date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1930y, November, 23d }));
40         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Count_worker_bevor_1951_date", static_cast<size_t>(2), company.GetCountWorkerBeforDate({ 1951y, November, 23d }));
41
42         TestOK = TestOK && check_dump(ost, "Test_Company_Get_longest_serving_employee", TWorker::E_Boss, company.GetLongestServing()->GetWorkerType());
43
44         TestOK = TestOK && check_dump(ost, "Test_Company_Get_total_pieces_produced", static_cast<size_t>(50), company.GetProducedItems());
45
46         TestOK = TestOK && check_dump(ost, "Test_Company_Get_total_pieces_sold", static_cast<size_t>(2700), company.GetSoldItems());
47
48     }
49     catch (const string& err) {
50         error_msg = err;
51         TestOK = false;
52     }
53     catch (bad_alloc const& error) {
54         error_msg = error.what();
55         TestOK = false;
56     }
57     catch (const exception& err) {
58         error_msg = err.what();
59         TestOK = false;
60     }
61     catch (...) {
62         error_msg = "Unhandelt_Exception";
63         TestOK = false;
64     }
65
66     TestEnd(ost);
67
68     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
69
70     return TestOK;
71 }
72
73
```



```
74 bool Client::TestEmptyCompanyGetter(std::ostream& ost, IComp& company) const
75 {
76     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
77
78     TestStart(ost);
79
80     bool TestOK = true;
81     string error_msg = "";
82
83
84     try {
85
86         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Commission_Worker_Cnt_&_Add_Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_CommissionWorker));
87         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Houerly_Worker_Cnt_&_Add_Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_HourlyWorker));
88         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Boss_Cnt_&_Add_Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_Boss));
89         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Piece_Worker_Cnt_&_Add_Empl", static_cast<size_t>(0), company.GetWorkerCount(TWorker::E_PieceWorker));
90
91
92         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_FindWorker_by_ID", static_cast<const Employee *>(nullptr), company.FindWorkerByID("Sil1"));
93         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_FindWorker_by_ID_empty_ID", static_cast<const Employee *>(nullptr), company.FindWorkerByID(""));
94
95
96         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Size", static_cast<size_t>(0), company.GetCompanySize());
97
98         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Count_worker_bevor_1930_date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1930y, November, 23d }));
99         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Count_worker_bevor_1951_date", static_cast<size_t>(0), company.GetCountWorkerBeforDate({ 1951y, November, 23d }));
100
101         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_longest_serving_employee", static_cast<const Employee*>(nullptr), company.GetLongestServing());
102
103
104         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_total_pieces_produced", static_cast<size_t>(0), company.GetProducedItems());
105
106         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_total_pieces_sold", static_cast<size_t>(0), company.GetSoldItems());
107
108     }
109     catch (const string& err) {
110         error_msg = err;
111         TestOK = false;
112     }
113     catch (bad_alloc const& error) {
114         error_msg = error.what();
115         TestOK = false;
116     }
117     catch (const exception& err) {
118         error_msg = err.what();
119         TestOK = false;
120     }
121     catch (...) {
122         error_msg = "Unhandelt_Exception";
123         TestOK = false;
124     }
125
126     try {
127
128         company.AddEmployee(nullptr);
129     }
130     catch (const string& err) {
131         error_msg = err;
132     }
133     catch (bad_alloc const& error) {
134         error_msg = error.what();
135     }
136     catch (const exception& err) {
137         error_msg = err.what();
138     }
139     catch (...) {
140         error_msg = "Unhandelt_Exception";
141     }
142
143
144     TestOK = TestOK && check_dump(ost, "Test_Company_Add_nullptr", Object::ERROR_NULLPTR, error_msg);
145
146     TestEnd(ost);
147
148     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
149
```

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

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

6.4 IComp.hpp

```
1  /*****
2  * \file    IComp.hpp
3  * \brief   Interface which is implemented by the company and used by the client
4  *
5  * \author  Simon Offenberger
6  * \date    October 2025
7  *****/
8  #ifndef ICOMP_HPP
9  #define ICOMP_HPP
10
11 #include <string>
12 #include "TWorker.hpp"
13 #include "Employee.hpp"
14
15 class IComp{
16 public:
17
18     /**
19     * \brief Gets the current size of the company.
20     *
21     * \return Size of the company
22     */
23     virtual size_t GetCompanySize() const = 0;
24
25     /**
26     * \brief Get the Count of a specific Worker Type.
27     *
28     * \param workerType Worker Type from which the count should be determined
29     * \return Count of the Worker Type in the Company
30     */
31     virtual size_t GetWorkerCount(const TWorker & workerType) const = 0;
32
33     /**
34     * \brief Get the amount of Sold Items in the whole company.
35     *
36     * \return Amout of Sold Items
37     */
38     virtual size_t GetSoldItems() const = 0;
39
40     /**
41     * \brief Get the amount of produced items.
42     *
43     * \return Amout of produced Items
44     */
45     virtual size_t GetProducedItems() const = 0;
46
47     /**
48     * \brief Get the of worker with birth date bevor date.
49     *
50     * \param date to get the employees which are older
51     * \return Amout of employees which are older than the passed in birthdate
52     */
53     virtual size_t GetCountWorkerBeforDate(const TDate & date) const = 0;
54
55     /**
56     * \brief Find a worker with a specific ID.
57     *
58     * \param id ID for which should be searched for
59     * \return nullptr if no Empl is found
60     * \return Pointer to Employee
61     */
62     virtual Employee const * FindWorkerByID(const std::string & id) const = 0;
63
64     /**
65     * \brief Get the Employee which has been the longest serving.
66     *
67     * \return nullptr if company is empty
68     * \return Pointer to Employee
69     */
70     virtual Employee const * GetLongestServing(void) const = 0;
71
72     /**
73     * \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 an 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

```
1  /*****
2  * \file    Company.hpp
3  * \brief   Company that holds Employees and provides information about the
4  * \brief   Employees of the company.
5  *
6  * \author  Simon Offenberger
7  * \date    October 2025
8  *****/
9  #ifndef COMPANY_HPP
10 #define COMPANY_HPP
11
12 #include <map>
13 #include <string>
14 #include "Object.hpp"
15 #include "IComp.hpp"
16
17 /**
18  * Declaration of an alias for the used Container.
19  */
20 using TContEmployee = std::map<const std::string, Employee const*>;
21
22 class Company : public Object, public IComp{
23 public:
24     /**
25      * Constant for the Exception of an Duplicate Employee.
26      */
27     inline static const std::string ERROR_DUPLICATE_EMPL = "ERROR:_Duplicate_Employee!";
28
29     /**
30      * \brief CTOR for a Company.
31      *
32      * \param name Name of the Company
33      */
34     Company(const std::string & name) : m_companyName{ name } {}
35
36     /**
37      * \brief Copy Ctor of the Company.
38      *
39      * \param comp Reference to the company that should be copied
40      */
41     Company(const Company & comp);
42
43     /**
44      * \brief Assignoperator for a company uses Copy and Swap.
45      *
46      * \param comp Copy of the company
47      */
48     void operator=(Company comp);
49
50     /**
51      * \brief Adds an Employee to the Company
52      * \brief The company now owns the Employee and is responsible for destructing of Employee.
53      *
54      * \param empl Employee that should be added to the Company
55      * \throw ERROR_DUPLICATE_EMPL if ID of Employee is already in the collection
56      * \throw ERROR_NULLPTR if an Nullptr is passed in
57      */
58     virtual void AddEmployee(Employee const* empl) override;
59
60     /**
61      * \brief Gets the current size of the company.
62      *
63      * \return Size of the company
64      */
65     virtual size_t GetCompanySize() const override;
66
67     /**
68      * \brief Get the Count of a specific Worker Type.
69      *
70      * \param workerType Worker Type from which the count should be determined
71      * \return Count of the Worker Type in the Company
72      */
73     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 Amount of Sold Items
79     */
80     virtual size_t GetSoldItems() const override;
81
82     /**
83     * \brief Get the amount of produced items.
84     *
85     * \return Amount 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 Amount 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

```
1  /*****
2  * \file    Company.cpp
3  * \brief   Company that holds Employees and provides information about the
4  * \brief   Employees of the company.
5  *
6  * \author  Simon Offenberger
7  * \date    October 2025
8  *****/
9  #include <algorithm>
10 #include <numeric>
11 #include <iostream>
12 #include "Company.hpp"
13 #include "Employee.hpp"
14 using namespace std;
15
16 /**
17  * \brief Ostream manipulator for creating a horizontal line.
18  *
19  * \return string
20  */
21 static ostream & hline(ostream & ost) {
22
23     ost << string(60, '-') << endl;
24     return ost;
25 }
26
27 /**
28  * \brief Ostream manipulator for creating a horizontal line.
29  *
30  * \return string
31  */
32 static ostream & hstar(ostream & ost) {
33
34     ost << string(60, '*') << endl;
35     return ost;
36 }
37
38 void Company::AddEmployee(Employee const* empl)
39 {
40     if (empl == nullptr) throw Object::ERROR_NULLPTR;
41     // insert returns a pair. First = Iterator, Second bool -> bool indicates if the insertion was successful.
42     if (!m_Employees.insert({ empl->GetID(), empl }).second) throw Company::ERROR_DUPLICATE_EMPL;
43 }
44
45 Company::Company(const Company& comp)
46 {
47     // copy Company name
48     m_companyName = comp.m_companyName;
49
50     // clone all employees from one company to the other
51     for_each(
52         comp.m_Employees.cbegin(), comp.m_Employees.cend(),
53         [&](auto& e) {AddEmployee(e.second->Clone());});
54 }
55
56
57 void Company::operator=(Company comp)
58 {
59     // copy and swap
60     std::swap(m_Employees, comp.m_Employees);
61     std::swap(m_companyName, comp.m_companyName);
62 }
63
64 size_t Company::GetCompanySize() const
65 {
66     return m_Employees.size();
67 }
68
69 size_t Company::GetWorkerCount(const TWorker& workerType) const
70 {
71     // Count all Employees where workerType is equal
72     return count_if(m_Employees.cbegin(), m_Employees.cend(),
73         [&](auto& e) {return e.second->GetWorkerType() == workerType;});
74 }
```



```
74 }
75
76 size_t Company::GetSoldItems() const
77 {
78     return accumulate(m_Employees.cbegin(), m_Employees.cend(), static_cast<size_t>(0),
79         [](size_t val, const auto& e) { return val + e.second->GetSoldItems(); });
80 }
81
82 size_t Company::GetProducedItems() const
83 {
84     return accumulate(m_Employees.cbegin(), m_Employees.cend(), static_cast<size_t>(0),
85         [](size_t val, const auto& e) { return val + e.second->GetProducedItems(); });
86 }
87
88 size_t Company::GetCountWorkerBeforDate(const TDate& date) const
89 {
90     return count_if(m_Employees.cbegin(), m_Employees.cend(),
91         [&](const auto& e) { return e.second->GetDateBirth() < date; });
92 }
93
94 Employee const * Company::FindWorkerByID(const std::string& id) const
95 {
96     auto empl = m_Employees.find(id);
97
98     if (empl == m_Employees.cend()) return nullptr;
99     else return empl->second;
100 }
101
102 Employee const * Company::GetLongestServing(void) const
103 {
104     auto minElem = min_element(m_Employees.cbegin(), m_Employees.cend(),
105         [](const auto& lhs, const auto& rhs) { return lhs.second->GetDateJoined() < rhs.second->GetDateJoined(); });
106
107     if (minElem == m_Employees.end()) return nullptr;
108     else return minElem->second;
109 }
110
111
112 std::ostream& Company::PrintDataSheet(std::ostream& ost) const
113 {
114
115     // convert system clock.now to days -> this can be used in CTOR for year month day
116     std::chrono::year_month_day date{ floor<std::chrono::days>(std::chrono::system_clock::now()) };
117
118     if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;
119
120     ost << hstar;
121     ost << m_companyName << endl;
122     ost << hstar;
123
124     for_each(m_Employees.cbegin(), m_Employees.cend(), [&](const auto& e) { e.second->PrintDatasheet(ost); });
125
126     ost << hline;
127     ost << date.month() << " " << date.year() << endl;
128     ost << hline;
129
130     if (ost.fail()) throw Object::ERROR_FAIL_WRITE;
131
132     return ost;
133 }
134
135 Company::~Company()
136 {
137     for (auto & elem : m_Employees)
138     {
139         delete elem.second;
140     }
141
142     m_Employees.clear();
143 }
```

6.7 TWorker.hpp

```
1  /*****
2  * \file   TWorker.hpp
3  * \brief  Enum for indicating the worker Type
4  *
5  * \author Simon
6  * \date   October 2025
7  *****/
8  #ifndef TWORKER_HPP
9  #define TWORKER_HPP
10
11 // changed naming convention because of
12 // name clashes with the actual classes
13 // that had the same name.
14 enum TWorker
15 {
16     E_Boss,
17     E_CommissionWorker,
18     E_HourlyWorker,
19     E_PieceWorker
20 };
21
22 #endif // !TWORKER_HPP
```

6.8 Employee.hpp

```
1  /***** Employee.hpp *****/
2  * \file Employee.hpp
3  * \brief Abstract Class for constructing Employees of all types
4  * \author Simon Vogelhuber
5  * \date October 2025
6  *****/
7  #ifndef EMPLOYEE_H
8  #define EMPLOYEE_H
9
10 #include <string>
11 #include <chrono>
12 #include "Object.hpp"
13 #include "TWorker.hpp"
14
15 using TDate = std::chrono::year_month_day;
16
17 class Employee : public Object
18 {
19 public:
20
21     inline static const std::string ERROR_BAD_ID = "ERROR:_An_employees_ID_is_limited_to_3_characters.";
22     inline static const std::string ERROR_BAD_SOZIAL_SEC_NUM = "ERROR:_Invalid_Sozial_Security_Number";
23
24     /**
25      * \brief Returns the ID of an Employee.
26      *
27      * \return String indication the ID
28      */
29     std::string GetID() const;
30
31     /**
32      * \brief Constructor needs every
33      * member set to be called.
34      * \return TWorker enum
35      */
36     Employee(
37         const std::string & name,
38         const std::string & nameID,
39         const TDate & dateJoined,
40         const TDate & TDateBirthdateBirth,
41         const std::string & socialSecurityNumber
42     );
43
44     /**
45      * \brief Gives Information about what kind
46      * of Worker it is.
47      * \return TWorker enum
48      */
49     virtual TWorker GetWorkerType() const = 0;
50
51     /** Pure Virtual Function
52      * \brief return produced items.
53      * \return size_t
54      */
55     virtual size_t GetProducedItems() const = 0;
56
57     /** Pure Virtual Function
58      * \brief returns sold items
59      * \return size_t
60      */
61     virtual size_t GetSoldItems() const = 0;
62
63     /** Pure Virtual Function
64      * \brief returns total pay a worker
65      * recieves.
66      * \return size_t
67      */
68     virtual size_t GetSalary() const = 0;
69
70     /**
71      * \brief returns date of birth of a given worker.
72      * \return TDate
73      */
74 }
```

```
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

```
1  /***** Employee.cpp *****/
2  * \file Employee.cpp
3  * \brief Abstract Class for constructing Employees of all types
4  * \author Simon Vogelhuber
5  * \date October 2025
6  *****/
7  #include "Employee.hpp"
8  #include <cctype>
9  #include <algorithm>
10
11 Employee::Employee(
12     const std::string & name,
13     const std::string & nameID,
14     const TDate & dateJoined,
15     const TDate & dateBirth,
16     const std::string & socialSecurityNumber
17 ) : m_name{ name },
18     m_nameIdentifier{ nameID },
19     m_dateJoined{ dateJoined },
20     m_dateBirth{ dateBirth }
21 {
22     if (nameID.length() != 3) throw ERROR_BAD_ID;
23
24     if (! std::all_of(socialSecurityNumber.begin(), socialSecurityNumber.end(), ::isdigit)) throw ERROR_BAD_SOZIAL_SEC_NUM;
25
26     if (! (socialSecurityNumber.size() == SozialSecNumLen) ) throw ERROR_BAD_SOZIAL_SEC_NUM;
27
28     m_socialSecurityNumber = socialSecurityNumber;
29 }
30
31
32 std::string Employee::GetID() const
33 {
34     return m_nameIdentifier;
35 }
36
37 TDate Employee::GetDateBirth() const
38 {
39     return m_dateBirth;
40 }
41
42 TDate Employee::GetDateJoined() const
43 {
44     return m_dateJoined;
45 }
46
47 std::ostream& Employee::PrintDatasheet(std::ostream& ost) const
48 {
49     if (ost.bad())
50     {
51         throw Object::ERROR_BAD_OSTREAM;
52     }
53
54     ost << "Datenblatt\n-----\n";
55     ost << "Name:_" << m_name << std::endl;
56     ost << "Kuerzel:_" << m_nameIdentifier << std::endl;
57     ost << "Sozialversicherungsnummer:_" << m_socialSecurityNumber;
58     ost << m_dateBirth.day() << static_cast<unsigned>(m_dateBirth.month()) << static_cast<int>(m_dateBirth.year())%100 << std::endl;
59     ost << "Geburtstag:_" << m_dateBirth << std::endl;
60     ost << "Einstiegsjahr:_" << m_dateJoined.year() << std::endl;
61
62     DoPrintSpecificData(ost);
63
64     ost << std::endl;
65
66     return ost;
67 }
```

6.10 Boss.hpp

```
1  /*****
2  * \file   Boss.hpp
3  * \brief  Boss Class - inherits from Employee
4  * \author Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #ifndef BOSS_H
8  #define BOSS_H
9
10 #include "Employee.hpp"
11
12 class Boss : public Employee
13 {
14 public:
15
16     Boss(
17         const std::string & name,
18         const std::string & nameID,
19         const TDate & dateJoined,
20         const TDate & dateBirth,
21         const std::string & socialSecurityNumber,
22         const size_t & salary
23     );
24
25
26     /**
27     * \brief Just here because of whacky class structure.
28     * Worker does not strictly produce items!
29     */
30     size_t GetProducedItems() const override { return 0; };
31
32     /**
33     * \brief Just here because of whacky class structure.
34     * Worker Does not sell items!
35     */
36     size_t GetSoldItems() const override { return 0; };
37
38     /**
39     * \brief Returns the total earnings for an
40     * worker in this month.
41     * \return size_t
42     */
43     size_t GetSalary() const override;
44
45     /**
46     * \brief Returns the type of worker.
47     * \return TWorker
48     */
49     TWorker GetWorkerType() const override;
50
51     /**
52     * \brief Creates a clone on the Heap
53     * and returns a pointer.
54     * \return Employee*
55     */
56     Employee* Clone() const override;
57
58 private:
59     /**
60     * \brief Prints worker specific information
61     * \param std::ostream& ost
62     * \return std::ostream&
63     */
64     std::ostream& DoPrintSpecificData(std::ostream& ost) const override;
65
66     size_t m_salary;
67 };
68
69 #endif // BOSS_H
```

6.11 Boss.cpp

```
1  /*****
2  * \file   Boss.cpp
3  * \brief  Boss Class - inherits from Employee
4  * \author Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #include "Boss.hpp"
8
9  Boss::Boss(
10     const std::string & name,
11     const std::string & nameID,
12     const TDate & dateJoined,
13     const TDate & dateBirth,
14     const std::string & socialSecurityNumber,
15     const size_t & salary
16 ) :
17     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
18     m_salary{ salary } {}
19
20 std::ostream& Boss::DoPrintSpecificData(std::ostream& ost) const
21 {
22     if (ost.bad())
23     {
24         throw Object::ERROR_BAD_OSTREAM;
25         return ost;
26     }
27     ost << "Role:_Boss" << std::endl;
28     ost << "Salary:_ " << m_salary << "_EUR" << std::endl;
29     return ost;
30 }
31
32
33 size_t Boss::GetSalary() const
34 {
35     return m_salary;
36 }
37
38 TWorker Boss::GetWorkerType() const
39 {
40     return E_Boss;
41 }
42
43 Employee* Boss::Clone() const
44 {
45     return new Boss( *this );
46 }
```

6.12 HourlyWorker.hpp

```
1  /*****
2  * \file   HourlyWorker.hpp
3  * \brief  HourlyWorker Class - Inherits from Employee
4  * \author Simon
5  * \date   October 2025
6  *****/
7  #ifndef HOURLY_WORKER_HPP
8  #define HOURLY_WORKER_HPP
9
10 #include "Employee.hpp"
11
12 class HourlyWorker : public Employee
13 {
14 public:
15
16     HourlyWorker(
17         const std::string & name,
18         const std::string & nameID,
19         const TDate & dateJoined,
20         const TDate & dateBirth,
21         const std::string & socialSecurityNumber,
22         const size_t & hourlyRate,
23         const size_t & workedHours
24     );
25
26
27
28     /**
29     * \brief Just here because of whacky class structure.
30     * Worker does not strictly produce items!
31     */
32     size_t GetProducedItems() const override { return 0; };
33
34     /**
35     * \brief Just here because of whacky class structure.
36     * Worker Does not sell items!
37     */
38     size_t GetSoldItems() const override { return 0; };
39
40     /**
41     * \brief Returns the total earnings for an
42     * worker in this month.
43     * \return size_t
44     */
45     size_t GetSalary() const override;
46
47     /**
48     * \brief Returns the type of worker.
49     * \return TWorker
50     */
51     TWorker GetWorkerType() const override;
52
53     /**
54     * \brief Creates a clone on the Heap
55     * and returns a pointer.
56     * \return Employee*
57     */
58     Employee* Clone() const override;
59
60 private:
61     /**
62     * \brief Prints worker specific information
63     * \param std::ostream& ost
64     * \return std::ostream&
65     */
66     std::ostream& DoPrintSpecificData(std::ostream& ost) const override;
67
68     size_t m_hourlyRate;
69     size_t m_workedHours;
70 };
71
72 #endif // !HOURLY_WORKER_HPP
```


6.13 HourlyWorker.cpp

```
1  /*****  
2  * \file   HourlyWorker.cpp  
3  * \brief  HourlyWorker Class - Inherits from Employee  
4  * \author  Simon  
5  * \date   October 2025  
6  *****/  
7  
8  #include "HourlyWorker.hpp"  
9  
10 HourlyWorker::HourlyWorker(  
11     const std::string & name,  
12     const std::string & nameID,  
13     const TDate & dateJoined,  
14     const TDate & dateBirth,  
15     const std::string & socialSecurityNumber,  
16     const size_t & hourlyRate,  
17     const size_t & workedHours  
18 ) :  
19     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),  
20     m_hourlyRate{ hourlyRate },  
21     m_workedHours{ workedHours }  
22 {}  
23  
24 std::ostream& HourlyWorker::DoPrintSpecificData(std::ostream& ost) const  
25 {  
26     if (ost.bad())  
27     {  
28         throw Object::ERROR_BAD_OSTREAM;  
29         return ost;  
30     }  
31     ost << "Role:_HourlyWorker" << std::endl;  
32     ost << "Hourly_rate:_ " << m_hourlyRate << "_EUR" << std::endl;  
33     ost << "Hours_worked:_ " << m_workedHours << "_EUR" << std::endl;  
34     return ost;  
35 }  
36  
37  
38 size_t HourlyWorker::GetSalary() const  
39 {  
40     return m_hourlyRate * m_workedHours;  
41 }  
42  
43 TWorker HourlyWorker::GetWorkerType() const  
44 {  
45     return E_HourlyWorker;  
46 }  
47  
48 Employee* HourlyWorker::Clone() const  
49 {  
50     return new HourlyWorker{*this};  
51 }
```

6.14 PieceWorker.hpp

```
1  /*****
2  * \file   PieceWorker.hpp
3  * \brief  PieceWorker Class - inherits from Employee
4  * \author  Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #ifndef PIECE_WORKER_H
8  #define PIECE_WORKER_H
9
10 #include "Employee.hpp"
11
12 class PieceWorker : public Employee
13 {
14 public:
15
16     PieceWorker(
17         const std::string & name,
18         const std::string & nameID,
19         const TDate & dateJoined,
20         const TDate & dateBirth,
21         const std::string & socialSecurityNumber,
22         const size_t & m_numberPieces,
23         const size_t & m_commissionPerPiece
24     );
25
26     /**
27     * \brief Returns the number of pieces the
28     * worker has produced
29     */
30     size_t GetProducedItems() const override;
31
32     /**
33     * \brief Just here because of whacky class structure.
34     * Worker does not strictly sell items!
35     */
36     size_t GetSoldItems() const override { return 0; };
37
38     /**
39     * \brief Returns the total earnings for an
40     * worker in this month.
41     * \return size_t
42     */
43     size_t GetSalary() const override;
44
45     /**
46     * \brief Returns the type of worker.
47     * \return TWorker
48     */
49     TWorker GetWorkerType() const override;
50
51     /**
52     * \brief Creates a clone on the Heap
53     * and returns a pointer.
54     * \return Employee*
55     */
56     Employee* Clone() const override;
57
58 private:
59     /**
60     * \brief Prints worker specific information
61     * \param std::ostream& ost
62     * \return std::ostream&
63     */
64     std::ostream& DoPrintSpecificData(std::ostream& ost) const override;
65
66     size_t m_numberPieces;
67     size_t m_commissionPerPiece;
68 };
69
70 #endif // !PIECE_WORKER_H
```

6.15 PieceWorker.cpp

```
1  /*****
2  * \file   PieceWorker.cpp
3  * \brief  PieceWorker Class - inherits from Employee
4  * \author Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #include "PieceWorker.hpp"
8
9  PieceWorker::PieceWorker(
10     const std::string & name,
11     const std::string & nameID,
12     const TDate & dateJoined,
13     const TDate & dateBirth,
14     const std::string & socialSecurityNumber,
15     const size_t & m_numberPieces,
16     const size_t & m_commissionPerPiece
17 ) :
18     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
19     m_numberPieces{ m_numberPieces },
20     m_commissionPerPiece{ m_commissionPerPiece }{}
21
22 std::ostream& PieceWorker::DoPrintSpecificData(std::ostream& ost) const
23 {
24     if (ost.bad())
25     {
26         throw Object::ERROR_BAD_OSTREAM;
27         return ost;
28     }
29     ost << "Role:_PieceWorker" << std::endl;
30     ost << "Pieces_produced:_ " << m_numberPieces << std::endl;
31     ost << "Pay_per_piece:_ " << m_commissionPerPiece << "_EUR" << std::endl;
32
33     return ost;
34 }
35
36 size_t PieceWorker::GetProducedItems() const
37 {
38     return m_numberPieces;
39 }
40
41 size_t PieceWorker::GetSalary() const
42 {
43     return m_numberPieces * m_commissionPerPiece;
44 }
45
46 TWorker PieceWorker::GetWorkerType() const
47 {
48     return E_PieceWorker;
49 }
50
51 Employee* PieceWorker::Clone() const
52 {
53     return new PieceWorker{ *this };
54 }
```

6.16 ComissionWorker.hpp

```
1  /*****
2  * \file    ComissionWorker.hpp
3  * \brief   ComissionWorker Class - inherits from Employee
4  * \author  Simon Vogelhuber
5  * \date    October 2025
6  *****/
7  #ifndef COMISSION_WORKER_H
8  #define COMISSION_WORKER_H
9
10 #include "Employee.hpp"
11
12 class ComissionWorker : public Employee
13 {
14 public:
15
16     ComissionWorker(
17         const std::string & name,
18         const std::string & nameID,
19         const TDate & dateJoined,
20         const TDate & dateBirth,
21         const std::string & socialSecurityNumber,
22         const size_t & baseSalary,
23         const size_t & commisionPerPiece,
24         const size_t & piecesSold
25     );
26
27     /**
28      * \brief Just here because of whacky class structure.
29      * Worker does not strictly produce items!
30      */
31     size_t GetProducedItems() const override { return 0; };
32
33     /**
34      * \brief returns how many items the commision worker has sold
35      * \return size_t sold items
36      */
37     size_t GetSoldItems() const override;
38
39     /**
40      * \brief Returns the total earnings for an
41      * worker in this month.
42      * \return size_t
43      */
44     size_t GetSalary() const override;
45
46     /**
47      * \brief Returns the type of worker.
48      * \return TWorker
49      */
50     TWorker GetWorkerType() const override;
51
52     /**
53      * \brief Creates a clone on the Heap
54      * and returns a pointer.
55      * \return Employee*
56      */
57     Employee* Clone() const override;
58
59 private:
60     /**
61      * \brief Prints worker specific information
62      * \param std::ostream& ost
63      * \return std::ostream&
64      */
65     std::ostream& DoPrintSpecificData(std::ostream& ost) const override;
66
67     size_t m_baseSalary;
68     size_t m_commissionPerPiece;
69     size_t m_piecesSold;
70 };
71
72 #endif // !COMISSION_WORKER_H
```

6.17 ComissionWorker.cpp

```
1  /*****
2  * \file    ComissionWorker.cpp
3  * \brief   ComissionWorker Class - inherits from Employee
4  * \author  Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #include "ComissionWorker.hpp"
8
9  ComissionWorker::ComissionWorker(
10     const std::string & name,
11     const std::string & nameID,
12     const TDate & dateJoined,
13     const TDate & dateBirth,
14     const std::string & socialSecurityNumber,
15     const size_t & baseSalary,
16     const size_t & commissionPerPiece,
17     const size_t & piecesSold
18 ) :
19     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
20     m_baseSalary{ baseSalary },
21     m_commissionPerPiece{ commissionPerPiece },
22     m_piecesSold { piecesSold }
23 {}
24
25 std::ostream& ComissionWorker::DoPrintSpecificData(std::ostream & ost) const
26 {
27     if (ost.bad())
28     {
29         throw Object::ERROR_BAD_OSTREAM;
30         return ost;
31     }
32     ost << "Role:_ComissionWorker" << std::endl;
33     ost << "Base_salary:_ " << m_baseSalary << "_EUR" << std::endl;
34     ost << "Comission_per_piece:_ " << m_commissionPerPiece << "_EUR" << std::endl;
35     ost << "Pieces_sold:_ " << m_piecesSold << std::endl;
36
37     return ost;
38 }
39
40 size_t ComissionWorker::GetSoldItems() const
41 {
42     return m_piecesSold;
43 }
44
45 size_t ComissionWorker::GetSalary() const
46 {
47     return m_baseSalary + m_piecesSold * m_commissionPerPiece;
48 }
49
50 TWorker ComissionWorker::GetWorkerType() const
51 {
52     return E_CommissionWorker;
53 }
54
55 Employee* ComissionWorker::Clone() const
56 {
57     return new ComissionWorker{ *this };
58 }
```

6.18 main.cpp

```
1  /*****
2  * \file    main.cpp
3  * \brief   Testdriver for the Company
4  *
5  * \author  Simon
6  * \date    October 2025
7  *****/
8  #include "Company.hpp"
9  #include "Employee.hpp"
10 #include "HourlyWorker.hpp"
11 #include "vld.h"
12 #include "Client.hpp"
13 #include "Test.hpp"
14 #include "ComissionWorker.hpp"
15 #include "Boss.hpp"
16 #include "PieceWorker.hpp"
17 #include <iostream>
18 #include <fstream>
19 #include <cassert>
20
21 using namespace std;
22 using namespace std::chrono;
23
24 static bool TestEmployeeBoss(std::ostream& ost);
25 static bool TestEmployeeHourlyWorker(std::ostream& ost);
26 static bool TestEmployeePieceWorker(std::ostream& ost);
27 static bool TestEmployeeComissionWorker(std::ostream& ost);
28 static bool TestCompanyAdd(std::ostream& ost);
29
30 #define WRITE_OUTPUT true
31
32 int main(void) {
33     bool TestOK = true;
34     ofstream testoutput;
35     try {
36
37         if (WRITE_OUTPUT == true) {
38             testoutput.open("TestOutput.txt");
39         }
40
41         Company comp{ "Offenberger_Devices" };
42         Client TestClient;
43         ComissionWorker* cWork = new ComissionWorker{ "Simon_1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 2500 };
44         ComissionWorker* cWork2 = new ComissionWorker{ "Simon_6", "Si6", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 200 };
45         HourlyWorker* hWork = new HourlyWorker{ "Simon_2", "Si2", { 2022y,November,23d }, { 1934y,November,23d }, "4712", 20, 25 };
46         Boss* boss = new Boss{ "Simon_3", "Si3", { 2000y,November,23d }, { 1950y,November,23d }, "4712", 35000 };
47         PieceWorker* pWork = new PieceWorker{ "Simon_4", "Si4", { 2022y,November,23d }, { 2010y,November,23d }, "4712", 25, 25 };
48         PieceWorker* pWork2 = new PieceWorker{ "Simon_5", "Si5", { 2022y,November,23d }, { 2011y,November,23d }, "4712", 25, 25 };
49
50         comp.AddEmployee(cWork);
51         comp.AddEmployee(cWork2);
52         comp.AddEmployee(hWork);
53         comp.AddEmployee(boss);
54         comp.AddEmployee(pWork);
55         comp.AddEmployee(pWork2);
56
57         TestOK = TestOK && TestClient.TestCompanyGetter(cout, comp);
58         if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyGetter(testoutput, comp);
59
60         // Copy Ctor Call !
61         Company compCopy = comp;
62
63         TestOK = TestOK && TestClient.TestCompanyCopyCTOR(cout, comp, compCopy);
64         if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyCopyCTOR(testoutput, comp, compCopy);
65
66         // Test Assign Operator
67         Company compAss{ "Assign_Company" };
68         compAss = comp;
69
70         TestOK = TestOK && TestClient.TestCompanyAssignOp(cout, comp, compAss);
71         if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyAssignOp(testoutput, comp, compAss);
72
73     }
```

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

```
150
151     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetSalary()", testSalary, testBoss.GetSalary());
152     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetSoldItems()", static_cast<size_t>(0), testBoss.GetSoldItems());
153     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetProducedItems()", static_cast<size_t>(0), testBoss.GetProducedItems());
154     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetWorkerType()", E_Boss, testBoss.GetWorkerType());
155     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetDateBirth()", dateBorn, testBoss.GetDateBirth());
156     TestOK = TestOK && check_dump(ost, "Test_-_Boss.GetDateJoined()", dateJoined, testBoss.GetDateJoined());
157 }
158 catch (const string& err) {
159     error_msg = err;
160 }
161 catch (bad_alloc const& error) {
162     error_msg = error.what();
163 }
164 catch (const exception& err) {
165     error_msg = err.what();
166 }
167 catch (...) {
168     error_msg = "Unhandelt_Exception";
169 }
170
171 TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
172 error_msg.clear();
173
174 //clone test
175 try {
176     size_t testSalary = 7800;
177     string svr = "4711";
178     TDate dateBorn = { 2000y,November,22d };
179     TDate dateJoined = { 2022y,November,23d };
180     string name = "Max_Musterman";
181     string id = "MAX";
182
183     Boss testBoss( name, id, dateJoined, dateBorn, svr, testSalary );
184     Employee* pEmp = testBoss.Clone();
185     TestOK = TestOK && check_dump(ost, "Test_Boss.Clone()", pEmp != nullptr && pEmp != &testBoss, true);
186     delete pEmp;
187 }
188 catch (const string& err) {
189     error_msg = err;
190 }
191 catch (bad_alloc const& error) {
192     error_msg = error.what();
193 }
194 catch (const exception& err) {
195     error_msg = err.what();
196 }
197 catch (...) {
198     error_msg = "Unhandelt_Exception";
199 }
200
201 TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
202 error_msg.clear();
203
204 // Unavailable ID
205 try {
206     size_t testSalary = 7800;
207     string svr = "4711";
208     TDate dateBorn = { 2000y,November,22d };
209     TDate dateJoined = { 2022y,November,23d };
210     string name = "Max_Musterman";
211     string id = "MAXL";
212
213     Boss testBoss( name, id, dateJoined, dateBorn, svr, testSalary );
214 }
215 catch (const string& err) {
216     error_msg = err;
217 }
218 catch (bad_alloc const& error) {
219     error_msg = error.what();
220 }
221 catch (const exception& err) {
222     error_msg = err.what();
223 }
224 catch (...) {
225     error_msg = "Unhandelt_Exception";
226 }
```



```
226 }
227
228 TestOK = TestOK && check_dump(ost, "Boss_Constructor_bad_ID", error_msg, Employee::ERROR_BAD_ID);
229 error_msg.clear();
230
231 // Constructor bad SV
232 try {
233     size_t testSalary = 7800;
234     string svr = "ARGH";
235     TDate dateBorn = { 2000y, November, 22d };
236     TDate dateJoined = { 2022y, November, 23d };
237     string name = "Max_Musterman";
238     string id = "MAX";
239
240     Boss testBoss( name, id, dateJoined, dateBorn, svr, testSalary );
241 }
242 catch (const string& err) {
243     error_msg = err;
244 }
245 catch (bad_alloc const& error) {
246     error_msg = error.what();
247 }
248 catch (const exception& err) {
249     error_msg = err.what();
250 }
251 catch (...) {
252     error_msg = "Unhandelt_Exception";
253 }
254
255 TestOK = TestOK && check_dump(ost, "Boss_Constructor_bad_SV_-_invalid_character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
256 error_msg.clear();
257
258
259 // Constructor bad SV - too many nums
260 try {
261     size_t testSalary = 7800;
262     string svr = "ARGH";
263     TDate dateBorn = { 2000y, November, 22d };
264     TDate dateJoined = { 2022y, November, 23d };
265     string name = "Max_Musterman";
266     string id = "MAX";
267
268     Boss testBoss( name, id, dateJoined, dateBorn, svr, testSalary );
269 }
270 catch (const string& err) {
271     error_msg = err;
272 }
273 catch (bad_alloc const& error) {
274     error_msg = error.what();
275 }
276 catch (const exception& err) {
277     error_msg = err.what();
278 }
279 catch (...) {
280     error_msg = "Unhandelt_Exception";
281 }
282
283 TestOK = TestOK && check_dump(ost, "Boss_Constructor_bad_SV_-_too_many_nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
284 error_msg.clear();
285
286 // Bad ostream
287 try {
288     size_t testSalary = 7800;
289     string svr = "4711";
290     TDate dateBorn = { 2000y, November, 22d };
291     TDate dateJoined = { 2022y, November, 23d };
292     string name = "Max_Musterman";
293     string id = "MAX";
294
295     Boss testBoss( name, id, dateJoined, dateBorn, svr, testSalary );
296     std::stringstream out_stream;
297     out_stream.setstate(ios::badbit);
298     testBoss.PrintDatasheet(out_stream);
299 }
300 catch (const string& err) {
301
```

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

```
378         delete pEmp;
379     }
380     catch (const string& err) {
381         error_msg = err;
382     }
383     catch (bad_alloc const& error) {
384         error_msg = error.what();
385     }
386     catch (const exception& err) {
387         error_msg = err.what();
388     }
389     catch (...) {
390         error_msg = "Unhandelt_Exception";
391     }
392
393     TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
394     error_msg.clear();
395
396     // Unavailable ID
397     try {
398         size_t hourlyRate = 21;
399         size_t workedHours = 160;
400         string svr = "4711";
401         TDate dateBorn = { 2000y,November,22d };
402         TDate dateJoined = { 2022y,November,23d };
403         string name = "Max_Musterman";
404         string id = "MAXL";
405
406         HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
407     }
408     catch (const string& err) {
409         error_msg = err;
410     }
411     catch (bad_alloc const& error) {
412         error_msg = error.what();
413     }
414     catch (const exception& err) {
415         error_msg = err.what();
416     }
417     catch (...) {
418         error_msg = "Unhandelt_Exception";
419     }
420
421     TestOK = TestOK && check_dump(ost, "HourlyWorker_Constructor_bad_ID", error_msg, Employee::ERROR_BAD_ID);
422     error_msg.clear();
423
424     // Constructor bad SV
425     try {
426         size_t hourlyRate = 21;
427         size_t workedHours = 160;
428         string svr = "ARGH";
429         TDate dateBorn = { 2000y,November,22d };
430         TDate dateJoined = { 2022y,November,23d };
431         string name = "Max_Musterman";
432         string id = "MAX";
433
434         HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
435     }
436     catch (const string& err) {
437         error_msg = err;
438     }
439     catch (bad_alloc const& error) {
440         error_msg = error.what();
441     }
442     catch (const exception& err) {
443         error_msg = err.what();
444     }
445     catch (...) {
446         error_msg = "Unhandelt_Exception";
447     }
448
449     TestOK = TestOK && check_dump(ost, "HourlyWorker_Constructor_bad_SV_-_invalid_character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
450
451     error_msg.clear();
452
453     // Constructor bad SV - too many nums
```

```
454     try {
455         size_t hourlyRate = 21;
456         size_t workedHours = 160;
457         string svr = "ARGH";
458         TDate dateBorn = { 2000y,November,22d };
459         TDate dateJoined = { 2022y,November,23d };
460         string name = "Max_Musterman";
461         string id = "MAX";
462
463         HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
464     }
465     catch (const string& err) {
466         error_msg = err;
467     }
468     catch (bad_alloc const& error) {
469         error_msg = error.what();
470     }
471     catch (const exception& err) {
472         error_msg = err.what();
473     }
474     catch (...) {
475         error_msg = "Unhandelt_Exception";
476     }
477
478     TestOK = TestOK && check_dump(ost, "HourlyWorker_Constructor_bad_SV_-_too_many_nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
479     error_msg.clear();
480
481     // Bad ostream
482     try {
483         size_t hourlyRate = 21;
484         size_t workedHours = 160;
485         string svr = "4711";
486         TDate dateBorn = { 2000y,November,22d };
487         TDate dateJoined = { 2022y,November,23d };
488         string name = "Max_Musterman";
489         string id = "MAX";
490
491         HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
492         std::stringstream out_stream;
493         out_stream.setstate(ios::badbit);
494         testHourlyWorker.PrintDatasheet(out_stream);
495     }
496     catch (const string& err) {
497         error_msg = err;
498     }
499     catch (bad_alloc const& error) {
500         error_msg = error.what();
501     }
502     catch (const exception& err) {
503         error_msg = err.what();
504     }
505     catch (...) {
506         error_msg = "Unhandelt_Exception";
507     }
508
509     TestOK = TestOK && check_dump(ost, "HourlyWorker_bad_ostream", error_msg, Object::ERROR_BAD_OSTREAM);
510     error_msg.clear();
511
512     TestEnd(ost);
513     return TestOK;
514 }
515
516 static bool TestEmployeePieceWorker(std::ostream& ost)
517 {
518     assert(ost.good());
519
520     TestStart(ost);
521
522     bool TestOK = true;
523     string error_msg = "";
524
525     try {
526         size_t piecesProduced = 950;
527         size_t comissionPerPiece = 2;
528         string svr = "4711";
529         TDate dateBorn = { 2000y,November,22d };
```

```
530     TDate dateJoined = { 2022y,November,23d };
531     string name = "Max_Musterman";
532     string id = "MAX";
533
534     PieceWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
535
536     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetSalary()", piecesProduced * comissionPerPiece, testHourlyWorker.GetSalary());
537     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
538     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetProducedItems()", piecesProduced, testHourlyWorker.GetProducedItems());
539     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetWorkerType()", E_PieceWorker, testHourlyWorker.GetWorkerType());
540     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
541     TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
542 }
543 catch (const string& err) {
544     error_msg = err;
545 }
546 catch (bad_alloc const& error) {
547     error_msg = error.what();
548 }
549 catch (const exception& err) {
550     error_msg = err.what();
551 }
552 catch (...) {
553     error_msg = "Unhandelt_Exception";
554 }
555
556 TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
557 error_msg.clear();
558
559 //clone test
560 try {
561     size_t piecesProduced = 950;
562     size_t comissionPerPiece = 2;
563     string svr = "4711";
564     TDate dateBorn = { 2000y,November,22d };
565     TDate dateJoined = { 2022y,November,23d };
566     string name = "Max_Musterman";
567     string id = "MAX";
568
569     PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
570     Employee* pEmp = testPieceWorker.Clone();
571     TestOK = TestOK && check_dump(ost, "Test_testPieceWorker.Clone()", pEmp != nullptr && pEmp != &testPieceWorker, true);
572     delete pEmp;
573 }
574 catch (const string& err) {
575     error_msg = err;
576 }
577 catch (bad_alloc const& error) {
578     error_msg = error.what();
579 }
580 catch (const exception& err) {
581     error_msg = err.what();
582 }
583 catch (...) {
584     error_msg = "Unhandelt_Exception";
585 }
586
587 TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
588 error_msg.clear();
589
590 // Unavailable ID
591 try {
592     size_t piecesProduced = 950;
593     size_t comissionPerPiece = 2;
594     string svr = "4711";
595     TDate dateBorn = { 2000y,November,22d };
596     TDate dateJoined = { 2022y,November,23d };
597     string name = "Max_Musterman";
598     string id = "MAXL";
599
600     PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
601 }
602 catch (const string& err) {
603     error_msg = err;
604 }
605 catch (bad_alloc const& error) {
```

```
606         error_msg = error.what();
607     }
608     catch (const exception& err) {
609         error_msg = err.what();
610     }
611     catch (...) {
612         error_msg = "Unhandelt_Exception";
613     }
614
615     TestOK = TestOK && check_dump(ost, "PieceWorker_Constructor_bad_ID", error_msg, Employee::ERROR_BAD_ID);
616     error_msg.clear();
617
618     // Constructor bad SV
619     try {
620         size_t piecesProduced = 950;
621         size_t comissionPerPiece = 2;
622         string svr = "ARGH";
623         TDate dateBorn = { 2000y,November,22d };
624         TDate dateJoined = { 2022y,November,23d };
625         string name = "Max_Musterman";
626         string id = "MAX";
627
628         PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
629     }
630     catch (const string& err) {
631         error_msg = err;
632     }
633     catch (bad_alloc const& error) {
634         error_msg = error.what();
635     }
636     catch (const exception& err) {
637         error_msg = err.what();
638     }
639     catch (...) {
640         error_msg = "Unhandelt_Exception";
641     }
642
643     TestOK = TestOK && check_dump(ost, "PieceWorker_Constructor_bad_SV_-_invalid_character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
644     error_msg.clear();
645
646     // Constructor bad SV - too many nums
647     try {
648         size_t piecesProduced = 950;
649         size_t comissionPerPiece = 2;
650         string svr = "ARGH";
651         TDate dateBorn = { 2000y,November,22d };
652         TDate dateJoined = { 2022y,November,23d };
653         string name = "Max_Musterman";
654         string id = "MAX";
655
656         PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
657     }
658     catch (const string& err) {
659         error_msg = err;
660     }
661     catch (bad_alloc const& error) {
662         error_msg = error.what();
663     }
664     catch (const exception& err) {
665         error_msg = err.what();
666     }
667     catch (...) {
668         error_msg = "Unhandelt_Exception";
669     }
670
671     TestOK = TestOK && check_dump(ost, "PieceWorker_Constructor_bad_SV_-_too_many_nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
672     error_msg.clear();
673
674     // Bad ostream
675     try {
676         size_t piecesProduced = 950;
677         size_t comissionPerPiece = 2;
678         string svr = "4711";
679         TDate dateBorn = { 2000y,November,22d };
680         TDate dateJoined = { 2022y,November,23d };
```

```
682         string name = "Max_Musterman";
683         string id = "MAX";
684
685         PieceWorker testPieceWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
686         std::stringstream out_stream;
687         out_stream.setstate(ios::badbit);
688         testPieceWorker.PrintDatasheet(out_stream);
689     }
690     catch (const string& err) {
691         error_msg = err;
692     }
693     catch (bad_alloc const& error) {
694         error_msg = error.what();
695     }
696     catch (const exception& err) {
697         error_msg = err.what();
698     }
699     catch (...) {
700         error_msg = "Unhandelt_Exception";
701     }
702
703     TestOK = TestOK && check_dump(ost, "PieceWorker_bad_ostream", error_msg, Object::ERROR_BAD_OSTREAM);
704     error_msg.clear();
705
706     TestEnd(ost);
707     return TestOK;
708 }
709
710 static bool TestEmployeeComissionWorker(std::ostream& ost)
711 {
712     assert(ost.good());
713
714     TestStart(ost);
715
716     bool TestOK = true;
717     string error_msg = "";
718
719     try {
720         size_t baseSalary = 2300;
721         size_t piecesSold = 300;
722         size_t comissionPerPiece = 2;
723         string svr = "4711";
724         TDate dateBorn = { 2000y, November, 22d };
725         TDate dateJoined = { 2022y, November, 23d };
726         string name = "Max_Musterman";
727         string id = "MAX";
728
729         ComissionWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
730
731         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetSalary()", baseSalary + piecesSold * comissionPerPiece, testHourlyWorker.GetSalary());
732         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetSoldItems()", piecesSold, testHourlyWorker.GetSoldItems());
733         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProducedItems());
734         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetWorkerType()", E_CommissionWorker, testHourlyWorker.GetWorkerType());
735         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
736         TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
737     }
738     catch (const string& err) {
739         error_msg = err;
740     }
741     catch (bad_alloc const& error) {
742         error_msg = error.what();
743     }
744     catch (const exception& err) {
745         error_msg = err.what();
746     }
747     catch (...) {
748         error_msg = "Unhandelt_Exception";
749     }
750
751     TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
752     error_msg.clear();
753
754     //clone test
755     try {
756         size_t baseSalary = 2300;
757         size_t piecesSold = 300;
```

```
758         size_t comissionPerPiece = 2;
759         string svr = "4711";
760         TDate dateBorn = { 2000y,November,22d };
761         TDate dateJoined = { 2022y,November,23d };
762         string name = "Max_Musterman";
763         string id = "MAX";
764
765         ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
766         Employee* pEmp = testComissionWorker.Clone();
767         TestOK = TestOK && check_dump(ost, "Test_TestPieceWorker.Clone()", pEmp != nullptr && pEmp != &testComissionWorker, true);
768         delete pEmp;
769     }
770     catch (const string& err) {
771         error_msg = err;
772     }
773     catch (bad_alloc const& error) {
774         error_msg = error.what();
775     }
776     catch (const exception& err) {
777         error_msg = err.what();
778     }
779     catch (...) {
780         error_msg = "Unhandelt_Exception";
781     }
782
783     TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
784     error_msg.clear();
785
786     // Unavialable ID
787     try {
788         size_t baseSalary = 2300;
789         size_t piecesSold = 300;
790         size_t comissionPerPiece = 2;
791         string svr = "4711";
792         TDate dateBorn = { 2000y,November,22d };
793         TDate dateJoined = { 2022y,November,23d };
794         string name = "Max_Musterman";
795         string id = "MAXL";
796
797         ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
798     }
799     catch (const string& err) {
800         error_msg = err;
801     }
802     catch (bad_alloc const& error) {
803         error_msg = error.what();
804     }
805     catch (const exception& err) {
806         error_msg = err.what();
807     }
808     catch (...) {
809         error_msg = "Unhandelt_Exception";
810     }
811
812     TestOK = TestOK && check_dump(ost, "ComissionWorker_Constructor_bad_ID_", error_msg, Employee::ERROR_BAD_ID);
813     error_msg.clear();
814
815     // Constructor bad SV - no numbers
816     try {
817         size_t baseSalary = 2300;
818         size_t piecesSold = 300;
819         size_t comissionPerPiece = 2;
820         string svr = "ARGH";
821         TDate dateBorn = { 2000y,November,22d };
822         TDate dateJoined = { 2022y,November,23d };
823         string name = "Max_Musterman";
824         string id = "MAX";
825
826         ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
827     }
828     catch (const string& err) {
829         error_msg = err;
830     }
831     catch (bad_alloc const& error) {
832         error_msg = error.what();
833     }
```



```
834     catch (const exception& err) {
835         error_msg = err.what();
836     }
837     catch (...) {
838         error_msg = "Unhandelt_Exception";
839     }
840
841     TestOK = TestOK && check_dump(ost, "ComissionWorker_Constructor_bad_SV_-_invalid_character", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
842
843     error_msg.clear();
844
845     // Constructor bad SV - too many nums
846     try {
847         size_t baseSalary = 2300;
848         size_t piecesSold = 300;
849         size_t comissionPerPiece = 2;
850         string svr = "47488888239874";
851         TDate dateBorn = { 2000y,November,22d };
852         TDate dateJoined = { 2022y,November,23d };
853         string name = "Max_Musterman";
854         string id = "MAX";
855
856         ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
857     }
858     catch (const string& err) {
859         error_msg = err;
860     }
861     catch (bad_alloc const& error) {
862         error_msg = error.what();
863     }
864     catch (const exception& err) {
865         error_msg = err.what();
866     }
867     catch (...) {
868         error_msg = "Unhandelt_Exception";
869     }
870
871     TestOK = TestOK && check_dump(ost, "ComissionWorker_Constructor_bad_SV_-_too_many_nums", Employee::ERROR_BAD_SOZIAL_SEC_NUM, error_msg);
872     error_msg.clear();
873
874     // Bad ostream
875     try {
876         size_t baseSalary = 2300;
877         size_t piecesSold = 300;
878         size_t comissionPerPiece = 2;
879         string svr = "4711";
880         TDate dateBorn = { 2000y,November,22d };
881         TDate dateJoined = { 2022y,November,23d };
882         string name = "Max_Musterman";
883         string id = "MAX";
884
885         ComissionWorker testComissionWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
886         std::stringstream out_stream;
887         out_stream.setstate(ios::badbit);
888         testComissionWorker.PrintDatasheet(out_stream);
889     }
890     catch (const string& err) {
891         error_msg = err;
892     }
893     catch (bad_alloc const& error) {
894         error_msg = error.what();
895     }
896     catch (const exception& err) {
897         error_msg = err.what();
898     }
899     catch (...) {
900         error_msg = "Unhandelt_Exception";
901     }
902
903     TestOK = TestOK && check_dump(ost, "ComissionWorker_bad_ostream", error_msg, Object::ERROR_BAD_OSTREAM);
904     error_msg.clear();
905
906     TestEnd(ost);
907     return TestOK;
908 }
909
```

```
910
911 static bool TestCompanyAdd(std::ostream& ost)
912 {
913     assert(ost.good());
914     TestStart(ost);
915
916     bool TestOK = true;
917     string error_msg = "";
918
919     try {
920
921         ComissionWorker* cWork = new ComissionWorker( "Simon_1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711", 2500, 25, 2500 );
922
923         Company comp{"Dup"};
924         comp.AddEmployee(cWork);
925         comp.AddEmployee(cWork);
926     }
927     catch (const string& err) {
928         error_msg = err;
929     }
930     catch (bad_alloc const& error) {
931         error_msg = error.what();
932     }
933     catch (const exception& err) {
934         error_msg = err.what();
935     }
936     catch (...) {
937         error_msg = "Unhandelt_Exception";
938     }
939
940     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Company_Add_Duplicate", Company::ERROR_DUPLICATE_EMPL, error_msg);
941     error_msg.clear();
942
943     TestEnd(ost);
944     return TestOK;
945 }
946
```

6.19 Test.hpp

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

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

```
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, "\n"));
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, "\n"));
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, "\n"));
165     return ost;
166 }
167
168
169 #endif // !TEST_HPP
```