



HSD

FH-HAGENBERG

Systemdokumentation Projekt Gehaltsberechnung

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 21. Oktober 2025

Inhaltsverzeichnis

1	Organisatorisches	3
1.1	Team	3
1.2	Aufteilung der Verantwortlichkeitsbereiche	3
1.3	Aufwand	4
2	Anforderungsdefinition (Systemspezifikation)	5
3	Systementwurf	6
3.1	Klassendiagramm	6
3.2	Designentscheidungen	7
4	Dokumentation der Komponenten (Klassen)	7
5	Testprotokollierung	8
6	Quellcode	9
6.1	Object.hpp	9
6.2	Client.hpp	9
6.3	Client.cpp	9
6.4	IComp.hpp	16
6.5	Company.hpp	16
6.6	Company.cpp	17
6.7	TWorker.hpp	19
6.8	Employee.hpp	19
6.9	Employee.cpp	21
6.10	Boss.hpp	22
6.11	Boss.cpp	23
6.12	HourlyWorker.hpp	23
6.13	HourlyWorker.cpp	24
6.14	PieceWorker.hpp	25
6.15	PieceWorker.cpp	26
6.16	ComissionWorker.hpp	27
6.17	ComissionWorker.cpp	28
6.18	main.cpp	29

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 x Ph
- Simon Vogelhuber: geschätzt x Ph / tatsächlich x 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

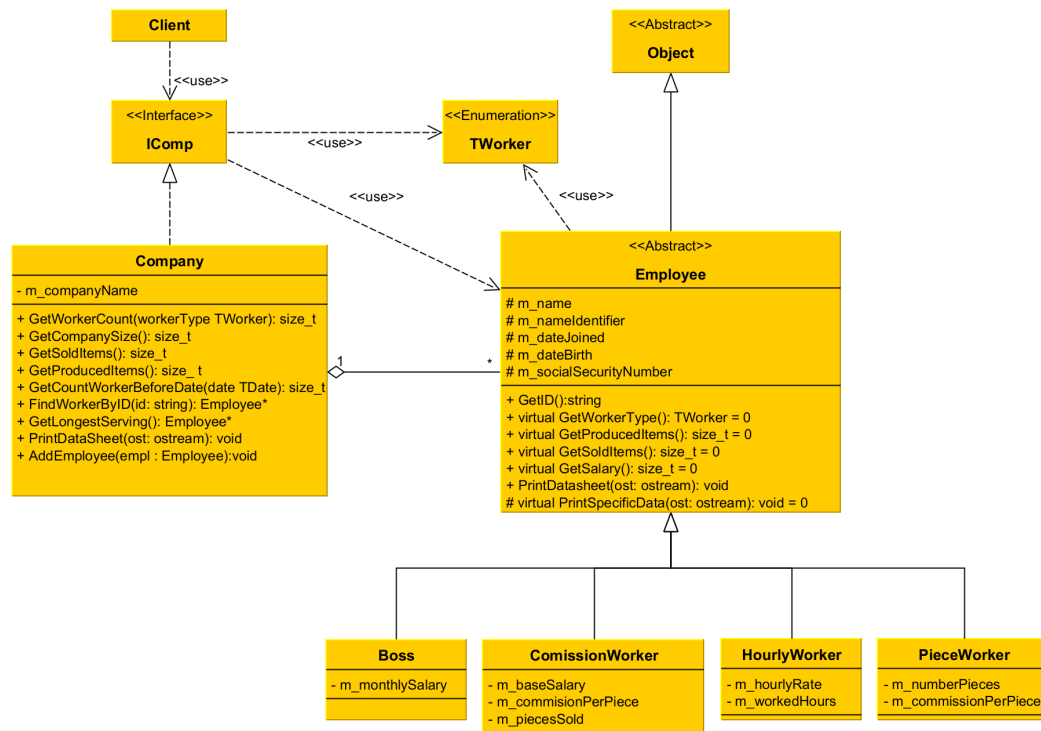
- Verwalten von Mitarbeitern verschiedener Arten.
- Ausgabe von Firmen und Mitarbeiterinformationen.
- Anlegen und Entfernen von Mitarbeitern.

Funktionen der Mitarbeiter

- Speichern von Mitarbeiterdaten.
- Berechnung des Gehalts.
- Ausgabe von Mitarbeiterinformationen.

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 ist ein polymorpher 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.

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.

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

4 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis [../doxy/html/index.html](http://doxy/html/index.html)

5 Testprotokollierung

6 Quellcode

6.1 Object.hpp

```
1 #ifndef OBJECT_HPP
2 #define OBJECT_HPP
3
4 class Object {
5 public:
6
7     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";
8     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";
9     inline static const std::string ERROR_NULLPTR = "ERROR:_Passed_in_Nullptr!";
10
11 protected:
12     Object() = default;
13
14     virtual ~Object() = default;
15 };
16
17
18 #endif // !OBJECT_HPP
```

6.2 Client.hpp

```
1 #ifndef CLIENT_HPP
2 #define CLIENT_HPP
3
4 #include <iostream>
5 #include "IComp.hpp"
6
7 class Client {
8 public:
9
10     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";
11     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";
12
13     bool TestCompanyGetter(std::ostream & ost, IComp& company) const;
14     bool TestEmptyCompanyGetter(std::ostream & ost, IComp& company) const;
15     bool TestCompanyCopyCTOR(std::ostream & ost, const IComp& company, const IComp& companyCopy) const;
16     bool TestCompanyAssignOp(std::ostream & ost, const IComp& company, const IComp& companyAss) const;
17     bool TestCompanyPrint(std::ostream & ost, const IComp& company) const;
18
19     bool TestEmployeeBoss(std::ostream& ost);
20     bool TestEmployeeHourlyWorker(std::ostream& ost);
21     bool TestEmployeePieceWorker(std::ostream& ost);
22     bool TestEmployeeComissionWorker(std::ostream& ost);
23
24 };
25
26 #endif // !CLIENT_HPP
```

6.3 Client.cpp

```
1 #include "Client.hpp"
2 #include "Test.hpp"
```

```
3 #include "ComissionWorker.hpp"
4 #include "HourlyWorker.hpp"
5 #include "Boss.hpp"
6 #include "PieceWorker.hpp"
7 #include <sstream>
8 #include <fstream>
9
10 using namespace std;
11 using namespace std::chrono;
12
13 bool Client::TestCompanyGetter(std::ostream& ost, IComp & company) const
14 {
15     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
16
17     TestStart(ost);
18
19     bool TestOK = true;
20     string error_msg = "";
21
22
23     try {
24
25         ComissionWorker* cWork = new ComissionWorker{ "Simon_1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711221100", 25000 };
26         ComissionWorker* cWork2 = new ComissionWorker{ "Simon_6", "Si6", { 2022y,November,23d }, { 2000y,November,22d }, "4711221100", 25000 };
27         HourlyWorker* hWork = new HourlyWorker{ "Simon_2", "Si2", { 2022y,November,23d }, { 1934y,November,23d }, "4712231100", 20, 25 };
28         Boss* boss = new Boss{ "Simon_3", "Si3", { 2000y,November,23d }, { 1950y,November,23d }, "4712231100", 35000 };
29         PieceWorker* pWork= new PieceWorker{ "Simon_4", "Si4", { 2022y,November,23d }, { 2010y,November,23d }, "4712231100", 25, 25 };
30         PieceWorker* pWork2= new PieceWorker{ "Simon_5", "Si5", { 2022y,November,23d }, { 2011y,November,23d }, "4712231100", 25, 25 };
31
32         company.AddEmployee(cWork);
33         company.AddEmployee(cWork2);
34         company.AddEmployee(hWork);
35         company.AddEmployee(boss);
36         company.AddEmployee(pWork);
37         company.AddEmployee(pWork2);
38
39         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Comission_Worker_Cnt_&_Add_Empl", static_cast<size_t>(2), company.GetWorkerCnt());
40         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Houerly_Worker_Cnt_&_Add_Empl", static_cast<size_t>(1), company.GetWorkerCnt());
41         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Boss_Cnt_&_Add_Empl", static_cast<size_t>(1), company.GetWorkerCnt());
42         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Piece_Worker_Cnt_&_Add_Empl", static_cast<size_t>(2), company.GetWorkerCnt());
43
44
45         TestOK = TestOK && check_dump(ost, "Test_Company_FindWorker_by_ID", static_cast<const Employee*>(cWork), company.FindWorkerByID());
46         TestOK = TestOK && check_dump(ost, "Test_Company_FindWorker_by_Empty_ID", static_cast<const Employee*>(nullptr), company.FindWorkerByID());
47
48         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Size", static_cast<size_t>(6), company.GetCompanySize());
49
50         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Count_worker_bevor_1930_date", static_cast<size_t>(0), company.GetCountWorkerBeforeDate(1930));
51         TestOK = TestOK && check_dump(ost, "Test_Company_Get_Count_worker_bevor_1951_date", static_cast<size_t>(2), company.GetCountWorkerBeforeDate(1951));
52
53         TestOK = TestOK && check_dump(ost, "Test_Company_Get_longest_serving_employee", static_cast<const Employee*>(boss), company.GetLongestServingEmployee());
54
55
56         TestOK = TestOK && check_dump(ost, "Test_Company_Get_total_pieces_produced", static_cast<size_t>(50), company.GetProducedItems());
57
58         TestOK = TestOK && check_dump(ost, "Test_Company_Get_total_pieces_sold", static_cast<size_t>(2700), company.GetSoldItems());
59
60     }
61     catch (const string& err) {
62         error_msg = err;
63         TestOK = false;
64     }
65     catch (bad_alloc const& error) {
66         error_msg = error.what();
67         TestOK = false;
68     }
69     catch (const exception& err) {
70         error_msg = err.what();
71         TestOK = false;
72     }
73     catch (...) {
74         error_msg = "Unhandelt_Exception";
75         TestOK = false;
76     }
77 }
```

```
78     TestEnd(ost);
79
80     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
81
82     return TestOK;
83 }
84
85 bool Client::TestEmptyCompanyGetter(std::ostream& ost, IComp& company) const
86 {
87     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
88
89     TestStart(ost);
90
91     bool TestOK = true;
92     string error_msg = "";
93
94
95     try {
96
97         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Comission_Worker_Cnt_&Add_Empl", static_cast<size_t>(0), company.GetWorke
98         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Houerly_Worker_Cnt_&Add_Empl", static_cast<size_t>(0), company.GetWor
99         TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Boss_Cnt_&Add_Empl", static_cast<size_t>(0), company.GetWorkerCount(T
100        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Piece_Worker_Cnt_&Add_Empl", static_cast<size_t>(0), company.GetWorke
101
102
103        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_FindWorker_by_ID", static_cast<const Employee *>(nullptr), company.FindWor
104        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_FindWorker_by_ID_empty_ID", static_cast<const Employee *>(nullptr), compan
105
106
107        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Size", static_cast<size_t>(0), company.GetCompanySize());
108
109        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Count_worker_bevor_1930_date", static_cast<size_t>(0), company.GetCoun
110        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_Count_worker_bevor_1951_date", static_cast<size_t>(0), company.GetCoun
111
112        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_longest_serving_employee", static_cast<const Employee*>(nullptr), comp
113
114
115        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_total_pieces_produced", static_cast<size_t>(0), company.GetProducedIt
116
117        TestOK = TestOK && check_dump(ost, "Test_Empty_Company_Get_total_pieces_sold", static_cast<size_t>(0), company.GetSoldItems());
118
119    }
120    catch (const string& err) {
121        error_msg = err;
122        TestOK = false;
123    }
124    catch (bad_alloc const& error) {
125        error_msg = error.what();
126        TestOK = false;
127    }
128    catch (const exception& err) {
129        error_msg = err.what();
130        TestOK = false;
131    }
132    catch (...) {
133        error_msg = "Unhandelt_Exception";
134        TestOK = false;
135    }
136
137    try {
138
139        company.AddEmployee(nullptr);
140    }
141    catch (const string& err) {
142        error_msg = err;
143    }
144    catch (bad_alloc const& error) {
145        error_msg = error.what();
146    }
147    catch (const exception& err) {
148        error_msg = err.what();
149    }
150    catch (...) {
151        error_msg = "Unhandelt_Exception";
152    }
}
```

```
153
154
155     TestOK = TestOK && check_dump(ost, "Test_Company_Add_nullptr", Object::ERROR_NULLPTR, error_msg);
156
157     TestEnd(ost);
158
159     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
160
161     return TestOK;
162 }
163
164 bool Client::TestCompanyCopyCTOR(std::ostream& ost, const IComp& company, const IComp& companyCopy) const
165 {
166     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
167
168     TestStart(ost);
169
170     bool TestOK = true;
171     string error_msg = "";
172
173     try {
174         stringstream result;
175         stringstream expected;
176
177         company.PrintDataSheet(expected);
178         companyCopy.PrintDataSheet(result);
179
180         TestOK = TestOK && check_dump(ost, "Test_Company_Copy_Ctor", true, expected.str() == result.str());
181     }
182     catch (const string& err) {
183         error_msg = err;
184         TestOK = false;
185     }
186     catch (bad_alloc const& error) {
187         error_msg = error.what();
188         TestOK = false;
189     }
190     catch (const exception& err) {
191         error_msg = err.what();
192         TestOK = false;
193     }
194     catch (...) {
195         error_msg = "Unhandelt_Exception";
196         TestOK = false;
197     }
198
199     TestEnd(ost);
200
201     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
202
203     return TestOK;
204
205     return false;
206 }
207
208
209
210
211 bool Client::TestCompanyAssignOp(std::ostream& ost, const IComp& company, const IComp& companyAss) const
212 {
213     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
214
215     TestStart(ost);
216
217     bool TestOK = true;
218     string error_msg = "";
219
220     try {
221         stringstream result;
222         stringstream expected;
223
224         company.PrintDataSheet(expected);
225         companyAss.PrintDataSheet(result);
226     }
227 }
```

```
228         TestOK = TestOK && check_dump(ost, "Test_Company_Assign_Operator", true, expected.str() == result.str());
229     }
230 }
231 catch (const string& err) {
232     error_msg = err;
233     TestOK = false;
234 }
235 catch (bad_alloc const& error) {
236     error_msg = error.what();
237     TestOK = false;
238 }
239 catch (const exception& err) {
240     error_msg = err.what();
241     TestOK = false;
242 }
243 catch (...) {
244     error_msg = "Unhandelt_Exception";
245     TestOK = false;
246 }
247
248 TestEnd(ost);
249
250 if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
251
252 return TestOK;
253
254 return false;
255 }
256
257 bool Client::TestCompanyPrint(std::ostream& ost, const IComp& company) const
258 {
259     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
260
261     TestStart(ost);
262
263     bool TestOK = true;
264     string error_msg = "";
265
266     fstream badstream;
267     badstream.setstate(ios::badbit);
268
269     try {
270
271         company.PrintDataSheet(badstream);
272     }
273     catch (const string& err) {
274         error_msg = err;
275     }
276     catch (bad_alloc const& error) {
277         error_msg = error.what();
278     }
279     catch (const exception& err) {
280         error_msg = err.what();
281     }
282     catch (...) {
283         error_msg = "Unhandelt_Exception";
284     }
285
286     TestOK = TestOK && check_dump(ost, "Test_Company_Print_Exception", Client::ERROR_BAD_OSTREAM, error_msg);
287
288     TestEnd(ost);
289
290     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
291
292     return TestOK;
293
294     return false;
295 }
296
297
298 bool Client::TestEmployeeBoss(std::ostream& ost)
299 {
300     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
301
302     TestStart(ost);
```

```
303
304     bool TestOK = true;
305     string error_msg = "";
306
307     try {
308         size_t testSalary = 7800;
309         string svr = "4711221100";
310         TDate dateBorn = { 2000y,November,22d };
311         TDate dateJoined = { 2022y,November,23d };
312         string name = "Max_Musterman";
313         string id = "MAX";
314
315         Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary};
316
317         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetSalary()", testSalary, testBoss.GetSalary());
318         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetSoldItems()", static_cast<size_t>(0), testBoss.GetSoldItems());
319         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetProducedItems()", static_cast<size_t>(0), testBoss.GetProducedItems());
320         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetWorkerType()", E_Boss, testBoss.GetWorkerType());
321         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetDateBirth()", dateBorn, testBoss.GetDateBirth());
322         TestOK = TestOK && check_dump(ost, "Test_/_Boss.GetDateJoined()", dateJoined, testBoss.GetDateJoined());
323     }
324     catch (const string& err) {
325         error_msg = err;
326     }
327     catch (bad_alloc const& error) {
328         error_msg = error.what();
329     }
330     catch (const exception& err) {
331         error_msg = err.what();
332     }
333     catch (...) {
334         error_msg = "Unhandelt_Exception";
335     }
336
337     TestOK = TestOK && check_dump(ost, "Test_/_error_buffer", error_msg.empty(), true);
338     TestEnd(ost);
339     return TestOK;
340 }
341
342 bool Client::TestEmployeeHourlyWorker(std::ostream& ost)
343 {
344     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
345
346     TestStart(ost);
347
348     bool TestOK = true;
349     string error_msg = "";
350
351     try {
352         size_t hourlyRate = 21;
353         size_t workedHours = 160;
354         string svr = "4711221100";
355         TDate dateBorn = { 2000y,November,22d };
356         TDate dateJoined = { 2022y,November,23d };
357         string name = "Max_Musterman";
358         string id = "MAX";
359
360         HourlyWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, hourlyRate, workedHours };
361
362         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetSalary()", hourlyRate * workedHours, testHourlyWorker.GetSalary());
363         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
364         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProducedItems());
365         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetWorkerType()", E_HourlyWorker, testHourlyWorker.GetWorkerType());
366         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
367         TestOK = TestOK && check_dump(ost, "Test_/_HourlyWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
368     }
369     catch (const string& err) {
370         error_msg = err;
371     }
372     catch (bad_alloc const& error) {
373         error_msg = error.what();
374     }
375     catch (const exception& err) {
376         error_msg = err.what();
377     }
```

```
378     catch (...) {
379         error_msg = "Unhandelt_Exception";
380     }
381
382     TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
383     TestEnd(ost);
384     return TestOK;
385 }
386
387 bool Client::TestEmployeePieceWorker(std::ostream& ost)
388 {
389     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
390
391     TestStart(ost);
392
393     bool TestOK = true;
394     string error_msg = "";
395
396     try {
397         size_t piecesProduced = 950;
398         size_t comissionPerPiece = 2;
399         string svr = "4711221100";
400         TDate dateBorn = { 2000y,November,22d };
401         TDate dateJoined = { 2022y,November,23d };
402         string name = "Max_Musterman";
403         string id = "MAX";
404
405         PieceWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };
406
407         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetSalary()", piecesProduced * comissionPerPiece, testHourlyWorker.GetSalary());
408         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
409         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetProducedItems()", piecesProduced, testHourlyWorker.GetProducedItems());
410         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetWorkerType()", E_PieceWorker, testHourlyWorker.GetWorkerType());
411         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
412         TestOK = TestOK && check_dump(ost, "Test_-_PieceWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
413     }
414     catch (const string& err) {
415         error_msg = err;
416     }
417     catch (bad_alloc const& error) {
418         error_msg = error.what();
419     }
420     catch (const exception& err) {
421         error_msg = err.what();
422     }
423     catch (...) {
424         error_msg = "Unhandelt_Exception";
425     }
426
427     TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
428     TestEnd(ost);
429     return TestOK;
430 }
431
432 bool Client::TestEmployeeComissionWorker(std::ostream& ost)
433 {
434     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
435
436     TestStart(ost);
437
438     bool TestOK = true;
439     string error_msg = "";
440
441     try {
442         size_t baseSalary = 2300;
443         size_t piecesSold = 300;
444         size_t comissionPerPiece = 2;
445         string svr = "4711221100";
446         TDate dateBorn = { 2000y,November,22d };
447         TDate dateJoined = { 2022y,November,23d };
448         string name = "Max_Musterman";
449         string id = "MAX";
450
451         ComissionWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };
452     }
```

```

453     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetSalary()", baseSalary + piecesSold * comissionPerPiece, testHourlyWorker.GetSalary());
454     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetSoldItems()", piecesSold, testHourlyWorker.GetSoldItems());
455     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProducedItems());
456     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetWorkerType()", E_CommissionWorker, testHourlyWorker.GetWorkerType());
457     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
458     TestOK = TestOK && check_dump(ost, "Test_-_ComissionWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
459 }
460 catch (const string& err) {
461     error_msg = err;
462 }
463 catch (bad_alloc const& error) {
464     error_msg = error.what();
465 }
466 catch (const exception& err) {
467     error_msg = err.what();
468 }
469 catch (...) {
470     error_msg = "Unhandelt_Exception";
471 }
472
473 TestOK = TestOK && check_dump(ost, "Test_-_error_buffer", error_msg.empty(), true);
474 TestEnd(ost);
475 return TestOK;
476 }

```

6.4 IComp.hpp

```

1  #ifndef ICOMP_HPP
2  #define ICOMP_HPP
3
4  #include <string>
5  #include "TWorker.hpp"
6  #include "Employee.hpp"
7
8  class IComp{
9  public:
10
11     virtual size_t GetCompanySize() const = 0;
12
13     virtual size_t GetWorkerCount(const TWorker & workerType) const = 0;
14
15     virtual size_t GetSoldItems() const = 0;
16
17     virtual size_t GetProducedItems() const = 0;
18
19     virtual size_t GetCountWorkerBeforDate(const TDate & date) const = 0;
20
21     virtual Employee const * FindWorkerByID(const std::string & id) const = 0;
22
23     virtual Employee const * GetLongestServing(void) const = 0;
24
25     virtual std::ostream& PrintDataSheet(std::ostream& ost) const = 0;
26
27     virtual void AddEmployee(Employee const* empl) = 0;
28
29     virtual ~IComp() = default;
30
31 };
32
33 #endif // !ICOMP_HPP

```

6.5 Company.hpp


```
1 #ifndef COMPANY_HPP
2 #define COMPANY_HPP
3
4 #include <map>
5 #include <string>
6 #include "Object.hpp"
7 #include "IComp.hpp"
8
9 using TContEmployee = std::map<const std::string, Employee const *>;
10
11 class Company : public Object, public IComp{
12 public:
13
14     Company(std::string name) : m_companyName{ name } {}
15
16     Company(const Company & comp);
17
18     void operator=(Company comp);
19
20     virtual void AddEmployee(Employee const* empl) override;
21
22     virtual size_t GetCompanySize() const override;
23
24     virtual size_t GetWorkerCount(const TWorker& workerType) const override;
25
26     virtual size_t GetSoldItems() const override;
27
28     virtual size_t GetProducedItems() const override;
29
30     virtual size_t GetCountWorkerBeforDate(const TDate& date) const override;
31
32     virtual Employee const * FindWorkerByID(const std::string& id) const override;
33
34     virtual Employee const * GetLongestServing(void) const override;
35
36     virtual std::ostream& PrintDataSheet(std::ostream& ost) const override;
37
38     ~Company();
39
40 private:
41     std::string m_companyName;
42     TContEmployee m_Employees;
43 };
44
45 #endif // !COMPANY_HPP
46
```

6.6 Company.cpp

```
1 #include <algorithm>
2 #include <numeric>
3 #include <iostream>
4 #include "Company.hpp"
5 #include "Employee.hpp"
6 using namespace std;
7
8 /**
9  * \brief Ostream manipulator for creating a horizontal line.
10  *
11  * \return string
12  */
13 static ostream & hline(ostream & ost) {
14
15     ost << string(60, '-') << endl;
16     return ost;
17 }
18
19 /**
```

```
20 * \brief ostream manipulator for creating a horizontal line.
21 *
22 * \return string
23 */
24 static ostream & hstar(ostream & ost) {
25
26     ost << string(60, '*') << endl;
27     return ost;
28 }
29
30 void Company::AddEmployee(Employee const* empl)
31 {
32     if (empl == nullptr) throw Object::ERROR_NULLPTR;
33     else m_Employees.insert({empl->GetID(), empl});
34 }
35
36 Company::Company(const Company& comp)
37 {
38     // copy Company name
39     m_companyName = comp.m_companyName;
40
41     // clone all employees from one company to the other
42     for_each(
43         comp.m_Employees.cbegin(), comp.m_Employees.cend(),
44         [&](auto& e) {AddEmployee(e.second->Clone());});
45 }
46
47
48 void Company::operator=(Company comp)
49 {
50     std::swap(m_Employees, comp.m_Employees);
51     std::swap(m_companyName, comp.m_companyName);
52 }
53
54 size_t Company::GetCompanySize() const
55 {
56     return m_Employees.size();
57 }
58
59 size_t Company::GetWorkerCount(const TWorker& workerType) const
60 {
61     return count_if(m_Employees.cbegin(), m_Employees.cend(),
62         [&](auto& e) {return e.second->GetWorkerType() == workerType;});
63 }
64
65 size_t Company::GetSoldItems() const
66 {
67     return accumulate(m_Employees.cbegin(), m_Employees.cend(), static_cast<size_t>(0),
68         [](size_t val, const auto& e) { return val + e.second->GetSoldItems();});
69 }
70
71 size_t Company::GetProducedItems() const
72 {
73     return accumulate(m_Employees.cbegin(), m_Employees.cend(), static_cast<size_t>(0),
74         [](size_t val, const auto& e) { return val + e.second->GetProducedItems();});
75 }
76
77 size_t Company::GetCountWorkerBeforDate(const TDate& date) const
78 {
79     return count_if(m_Employees.cbegin(), m_Employees.cend(),
80         [&](const auto& e) {return e.second->GetDateBirth() < date;});
81 }
82
83 Employee const* Company::FindWorkerByID(const std::string& id) const
84 {
85     auto empl = m_Employees.find(id);
86
87     if (empl == m_Employees.end()) return nullptr;
88     else return empl->second;
89 }
90
91 // longest serving ist glaub ich auf das Dienstalster und nicht auf den
92 // Geburtstag bezogen - TDate Employee::GetDateJoined()
93 Employee const* Company::GetLongestServing(void) const
94 {
```

```
95     auto minElem = min_element(m_Employees.cbegin(), m_Employees.cend(),
96                               [](const auto& lhs, const auto& rhs) { return lhs.second->GetDateJoined() < rhs.second->GetDateJoined();});
97
98
99     if (minElem == m_Employees.end()) return nullptr;
100     else return minElem->second;
101
102 }
103
104 std::ostream& Company::PrintDataSheet(std::ostream& ost) const
105 {
106
107     // convert system clock.now to days -> this can be used in CTOR for year month day
108     std::chrono::year_month_day date{ floor<std::chrono::days>(std::chrono::system_clock::now()) };
109
110     if (!ost.good()) throw Object::ERROR_BAD_OSTREAM;
111
112     ost << hstar;
113     ost << m_companyName << endl;
114     ost << hstar;
115
116     for_each(m_Employees.cbegin(), m_Employees.cend(), [&](const auto& e) { e.second->PrintDataSheet(ost);});
117
118     ost << hline;
119     ost << date.month() << "_" << date.year() << endl;
120     ost << hline;
121
122     if (ost.fail()) throw Object::ERROR_FAIL_WRITE;
123
124     return ost;
125 }
126
127 Company::~Company()
128 {
129     for (auto & elem : m_Employees)
130     {
131         delete elem.second;
132     }
133
134     m_Employees.clear();
135 }
```

6.7 TWorker.hpp

```
1  #ifndef TWORKER_HPP
2  #define TWORKER_HPP
3
4  // changed naming convention because of
5  // name clashes with the actual classes
6  // that had the same name.
7  enum TWorker
8  {
9      E_Boss,
10     E_CommissionWorker,
11     E_HourlyWorker,
12     E_PieceWorker
13 };
14
15 #endif // !TWORKER_HPP
```

6.8 Employee.hpp

```
1 #ifndef EMPLOYEE_H
2 #define EMPLOYEE_H
3
4 #include <string>
5 #include <chrono>
6 #include "Object.hpp"
7 #include "TWorker.hpp"
8
9 using TDate = std::chrono::year_month_day;
10
11 class Employee : public Object
12 {
13 public:
14
15     inline static const std::string ERROR_BAD_ID = "ERROR:_An_employees_ID_is_limited_to_3_characters.";
16
17     std::string GetID() const;
18
19     /**
20      * \brief Constructor needs every
21      * member set to be called.
22      * \return TWorker enum
23      */
24     Employee(
25         std::string name,
26         std::string nameID,
27         TDate dateJoined,
28         TDate TDateBirthdateBirth,
29         std::string socialSecurityNumber
30     );
31
32     /**
33      * \brief Gives Information about what kind
34      * of Worker it is.
35      * \return TWorker enum
36      */
37     virtual TWorker GetWorkerType() const = 0;
38
39     /** Pure Virtual Function
40      * \brief return produced items.
41      * \return size_t
42      */
43     virtual size_t GetProducedItems() const = 0;
44
45     /** Pure Virtual Function
46      * \brief returns sold items
47      * \return size_t
48      */
49     virtual size_t GetSoldItems() const = 0;
50
51     /** Pure Virtual Function
52      * \brief returns total pay a worker
53      * recieves.
54      * \return size_t
55      */
56     virtual size_t GetSalary() const = 0;
57
58     /** Pure Virtual Function
59      * \brief returns date of birth of a given worker.
60      * \return TDate
61      */
62     virtual TDate GetDateBirth() const;
63
64     /** Pure Virtual Function
65      * \brief returns the date a worker.
66      * has started working at the company.
67      * \return TDate
68      */
69     virtual TDate GetDateJoined() const;
70
71     /**
72      * \brief Prints information about a worker.
73      * \return std::ostream&
74      */
75 }
```

```
75     std::ostream& PrintDatasheet(std::ostream& ost) const;
76
77
78     /** Pure virtual function
79     * \brief Prints specific information for a type of worker.
80     * \return std::ostream&
81     */
82     virtual std::ostream& PrintSpecificData(std::ostream& ost) const = 0;
83
84     /** Pure virtual function
85     * \brief creates a copy of the worker and puts it on the heap.
86     * \return Employee*
87     */
88     virtual Employee* Clone() const = 0;
89
90 protected:
91
92     std::string m_name;
93     std::string m_nameIdentifier;
94     TDate m_dateJoined;
95     TDate m_dateBirth;
96     std::string m_socialSecurityNumber;
97 };
98
99 #endif // EMPLOYEE_H
100
```

6.9 Employee.cpp

```
1  #include "Employee.hpp"
2
3  Employee::Employee(
4      std::string name,
5      std::string nameID,
6      TDate dateJoined,
7      TDate dateBirth,
8      std::string socialSecurityNumber
9  ) : m_name{ name },
10     m_nameIdentifier{ nameID },
11     m_dateJoined{ dateJoined },
12     m_dateBirth{ dateBirth },
13     m_socialSecurityNumber{ socialSecurityNumber }
14 {
15     if (nameID.length() != 3)
16         throw ERROR_BAD_ID;
17 }
18
19 std::string Employee::GetID() const
20 {
21     return m_nameIdentifier;
22 }
23
24 TDate Employee::GetDateBirth() const
25 {
26     return m_dateBirth;
27 }
28
29 TDate Employee::GetDateJoined() const
30 {
31     return m_dateJoined;
32 }
33
34 std::ostream& Employee::PrintDatasheet(std::ostream& ost) const
35 {
36     if (ost.bad())
37     {
38         throw Object::ERROR_BAD_OSTREAM;
39         return ost;
40     }
41 }
```

```
40     }
41
42     ost << "Datenblatt\n-----\n";
43     ost << "Name:_" << m_name << std::endl;
44     ost << "Kuerzel:_" << m_nameIdentifier << std::endl;
45     ost << "Sozialversicherungsnummer:_" << m_socialSecurityNumber << std::endl;
46     ost << "Geburtstag:_" << m_dateBirth << std::endl;
47     ost << "Einstiegsjahr:_" << m_dateJoined.year() << std::endl;
48
49     PrintSpecificData(ost);
50
51     ost << std::endl;
52
53     return ost;
54 }
```

6.10 Boss.hpp

```
1  #ifndef BOSS_H
2  #define BOSS_H
3
4  #include "Employee.hpp"
5
6  class Boss : public Employee
7  {
8  public:
9
10     Boss() = default;
11
12     Boss(
13         std::string name,
14         std::string nameID,
15         TDate dateJoined,
16         TDate dateBirth,
17         std::string socialSecurityNumber,
18         size_t salary
19     );
20
21     /**
22      * \brief Prints worker specific information
23      * \param std::ostream& ost
24      * \return std::ostream&
25      */
26     std::ostream& PrintSpecificData(std::ostream& ost) const override;
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     size_t m_salary;
62 };
63
64 #endif // BOSS_H
```

6.11 Boss.cpp

```
1 #include "Boss.hpp"
2
3 Boss::Boss(
4     std::string name,
5     std::string nameID,
6     TDate dateJoined,
7     TDate dateBirth,
8     std::string socialSecurityNumber,
9     size_t salary
10 ) :
11     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
12     m_salary{ salary } {}
13
14 std::ostream& Boss::PrintSpecificData(std::ostream& ost) const
15 {
16     if (ost.bad())
17     {
18         throw Object::ERROR_BAD_OSTREAM;
19         return ost;
20     }
21     ost << "Role:_Boss" << std::endl;
22     ost << "Salary:_ " << m_salary << "_EUR" << std::endl;
23
24     return ost;
25 }
26
27 size_t Boss::GetSalary() const
28 {
29     return m_salary;
30 }
31
32 TWorker Boss::GetWorkerType() const
33 {
34     return E_Boss;
35 }
36
37 Employee* Boss::Clone() const
38 {
39     return new Boss{ *this };
40 }
```

6.12 HourlyWorker.hpp

```
1 #ifndef HOURLY_WORKER_HPP
```

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

6.13 HourlyWorker.cpp


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

6.14 PieceWorker.hpp

```
1 #ifndef PIECE_WORKER_H
2 #define PIECE_WORKER_H
3
4 #include "Employee.hpp"
5
6 class PieceWorker : public Employee
7 {
8 public:
9
10     PieceWorker() = default;
11
12     PieceWorker(
13         std::string name,
14         std::string nameID,
15         TDate dateJoined,
16         TDate dateBirth,
17         std::string socialSecurityNumber,
18         size_t m_numberPieces,
19         size_t m_commissionPerPiece
20     );
21 }
```

```
22  /**
23   * \brief Prints worker specific information
24   * \param std::ostream& ost
25   * \return std::ostream&
26   */
27  std::ostream& PrintSpecificData(std::ostream& ost) const override;
28
29  /**
30   * \brief Returns the number of pieces the
31   * worker has produced
32   */
33  size_t GetProducedItems() const override;
34
35  /**
36   * \brief Just here because of whacky class structure.
37   * Worker does not strictly sell items!
38   */
39  size_t GetSoldItems() const override { return 0; };
40
41  /**
42   * \brief Returns the total earnings for an
43   * worker in this month.
44   * \return size_t
45   */
46  size_t GetSalary() const override;
47
48  /**
49   * \brief Returns the type of worker.
50   * \return TWorker
51   */
52  TWorker GetWorkerType() const override;
53
54  /**
55   * \brief Creates a clone on the Heap
56   * and returns a pointer.
57   * \return Employee*
58   */
59  Employee* Clone() const override;
60
61 private:
62   size_t m_numberPieces;
63   size_t m_commissionPerPiece;
64 };
65
66 #endif // !PIECE_WORKER_H
```

6.15 PieceWorker.cpp

```
1  #include "PieceWorker.hpp"
2
3  PieceWorker::PieceWorker(
4      std::string name,
5      std::string nameID,
6      TDate dateJoined,
7      TDate dateBirth,
8      std::string socialSecurityNumber,
9      size_t m_numberPieces,
10     size_t m_commissionPerPiece
11 ) :
12     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
13     m_numberPieces{ m_numberPieces },
14     m_commissionPerPiece{ m_commissionPerPiece }{}
15
16 std::ostream& PieceWorker::PrintSpecificData(std::ostream& ost) const
17 {
18     if (ost.bad())
19     {
20         throw Object::ERROR_BAD_OSTREAM;
```

```
21         return ost;
22     }
23     ost << "Role:_PieceWorker" << std::endl;
24     ost << "Pieces_produced:_ " << m_numberPieces << std::endl;
25     ost << "Pay_per_piece:_ " << m_commissionPerPiece << "_EUR" << std::endl;
26
27     return ost;
28 }
29
30 size_t PieceWorker::GetProducedItems() const
31 {
32     return m_numberPieces;
33 }
34
35 size_t PieceWorker::GetSalary() const
36 {
37     return m_numberPieces * m_commissionPerPiece;
38 }
39
40 TWorker PieceWorker::GetWorkerType() const
41 {
42     return E_PieceWorker;
43 }
44
45 Employee* PieceWorker::Clone() const
46 {
47     return new PieceWorker{ *this };
48 }
```

6.16 ComissionWorker.hpp

```
1  #ifndef COMMISSION_WORKER_H
2  #define COMMISSION_WORKER_H
3
4  #include "Employee.hpp"
5
6  class ComissionWorker : public Employee
7  {
8  public:
9
10     ComissionWorker() = default;
11
12     ComissionWorker(
13         std::string name,
14         std::string nameID,
15         TDate dateJoined,
16         TDate dateBirth,
17         std::string socialSecurityNumber,
18         size_t baseSalary,
19         size_t commissionPerPiece,
20         size_t piecesSold
21     );
22
23     /**
24      * \brief Prints worker specific information
25      * \param std::ostream& ost
26      * \return std::ostream&
27      */
28     std::ostream& PrintSpecificData(std::ostream& ost) const override;
29
30     /**
31      * \brief Just here because of whacky class structure.
32      * Worker does not strictly produce items!
33      */
34     size_t GetProducedItems() const override { return 0; };
35
36     /**
37      * \brief returns how many items the commision worker has sold
```

```
38     * \return size_t sold items
39     */
40     size_t GetSoldItems() const override;
41
42     /**
43     * \brief Returns the total earnings for an
44     * worker in this month.
45     * \return size_t
46     */
47     size_t GetSalary() const override;
48
49     /**
50     * \brief Returns the type of worker.
51     * \return TWorker
52     */
53     TWorker GetWorkerType() const override;
54
55     /**
56     * \brief Creates a clone on the Heap
57     * and returns a pointer.
58     * \return Employee*
59     */
60     Employee* Clone() const override;
61
62 private:
63     size_t m_baseSalary;
64     size_t m_commissionPerPiece;
65     size_t m_piecesSold;
66 };
67
68 #endif // !COMMISSION_WORKER_H
```

6.17 ComissionWorker.cpp

```
1  #include "ComissionWorker.hpp"
2
3  ComissionWorker::ComissionWorker(
4      std::string name,
5      std::string nameID,
6      TDate dateJoined,
7      TDate dateBirth,
8      std::string socialSecurityNumber,
9      size_t baseSalary,
10     size_t commissionPerPiece,
11     size_t piecesSold
12 ) :
13     Employee(name, nameID, dateJoined, dateBirth, socialSecurityNumber),
14     m_baseSalary{ baseSalary },
15     m_commissionPerPiece{ commissionPerPiece },
16     m_piecesSold { piecesSold }
17 {}
18
19 std::ostream& ComissionWorker::PrintSpecificData(std::ostream & ost) const
20 {
21     if (ost.bad())
22     {
23         throw Object::ERROR_BAD_OSTREAM;
24         return ost;
25     }
26     ost << "Role:_ComissionWorker" << std::endl;
27     ost << "Base_salary:_ " << m_baseSalary << "_EUR" << std::endl;
28     ost << "Comission_per_piece:_ " << m_commissionPerPiece << "_EUR" << std::endl;
29     ost << "Pieces_sold:_ " << m_piecesSold << std::endl;
30
31     return ost;
32 }
33
34 size_t ComissionWorker::GetSoldItems() const
```

```
35 {
36     return m_piecesSold;
37 }
38
39 size_t ComissionWorker::GetSalary() const
40 {
41     return m_baseSalary + m_piecesSold * m_commissionPerPiece;
42 }
43
44 TWorker ComissionWorker::GetWorkerType() const
45 {
46     return E_CommissionWorker;
47 }
48
49 Employee* ComissionWorker::Clone() const
50 {
51     return new ComissionWorker{ *this };
52 }
```

6.18 main.cpp

```
1  #include "Company.hpp"
2  #include "Employee.hpp"
3  #include "HourlyWorker.hpp"
4  #include "vld.h"
5  #include "Client.hpp"
6  #include "Test.hpp"
7  #include "ComissionWorker.hpp"
8  #include "HourlyWorker.hpp"
9  #include "Boss.hpp"
10 #include "PieceWorker.hpp"
11 #include <iostream>
12 #include <fstream>
13
14 using namespace std;
15 using namespace std::chrono;
16
17 #define WRITE_OUTPUT false
18
19 int main(void){
20     bool TestOK = true;
21     ofstream testoutput;
22
23     if (WRITE_OUTPUT == true) {
24         testoutput.open("output.txt");
25     }
26
27     Company comp{"Offenberger_Devices"};
28     Client TestClient;
29
30
31     TestOK = TestOK && TestClient.TestCompanyGetter(cout, comp);
32     if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyGetter(testoutput, comp);
33
34     // Copy Ctor Call !
35     Company compCopy = comp;
36
37     TestOK = TestOK && TestClient.TestCompanyCopyCTOR(cout, comp, compCopy);
38     if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyCopyCTOR(testoutput, comp, compCopy);
39
40     // Test Assign Operator
41     Company compAss{"Assign_Company"};
42     compAss = comp;
43
44     TestOK = TestOK && TestClient.TestCompanyAssignOp(cout, comp, compAss);
45     if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyAssignOp(testoutput, comp, compAss);
46
47 }
```

```
48 TestOK = TestOK && TestClient.TestCompanyPrint(cout, comp);
49 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestCompanyPrint(testoutput, comp);
50
51 Company emptyComp{"empty"};
52
53 TestOK = TestOK && TestClient.TestEmptyCompanyGetter(cout, emptyComp);
54 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmptyCompanyGetter(testoutput, emptyComp);
55
56 // Test Boss
57 TestOK = TestOK && TestClient.TestEmployeeBoss(cout);
58 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeBoss(testoutput);
59
60 // Test Hourly Worker
61 TestOK = TestOK && TestClient.TestEmployeeHourlyWorker(cout);
62 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeHourlyWorker(testoutput);
63
64 // Test Piece Worker
65 TestOK = TestOK && TestClient.TestEmployeePieceWorker(cout);
66 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeePieceWorker(testoutput);
67
68 // Test Comission Worker
69 TestOK = TestOK && TestClient.TestEmployeeComissionWorker(cout);
70 if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeComissionWorker(testoutput);
71
72 if (WRITE_OUTPUT) {
73     if (TestOK) TestCaseOK(testoutput);
74     else TestCaseFail(testoutput);
75
76     testoutput.close();
77 }
78
79 if (TestOK) TestCaseOK(cout);
80 else TestCaseFail(cout);
81
82 }
```