Systemdokumentation Projekt Gehaltsberechnung

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 21. Oktober 2025

Inhaltsverzeichnis

1 Organisatorisches

1.1 Team

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

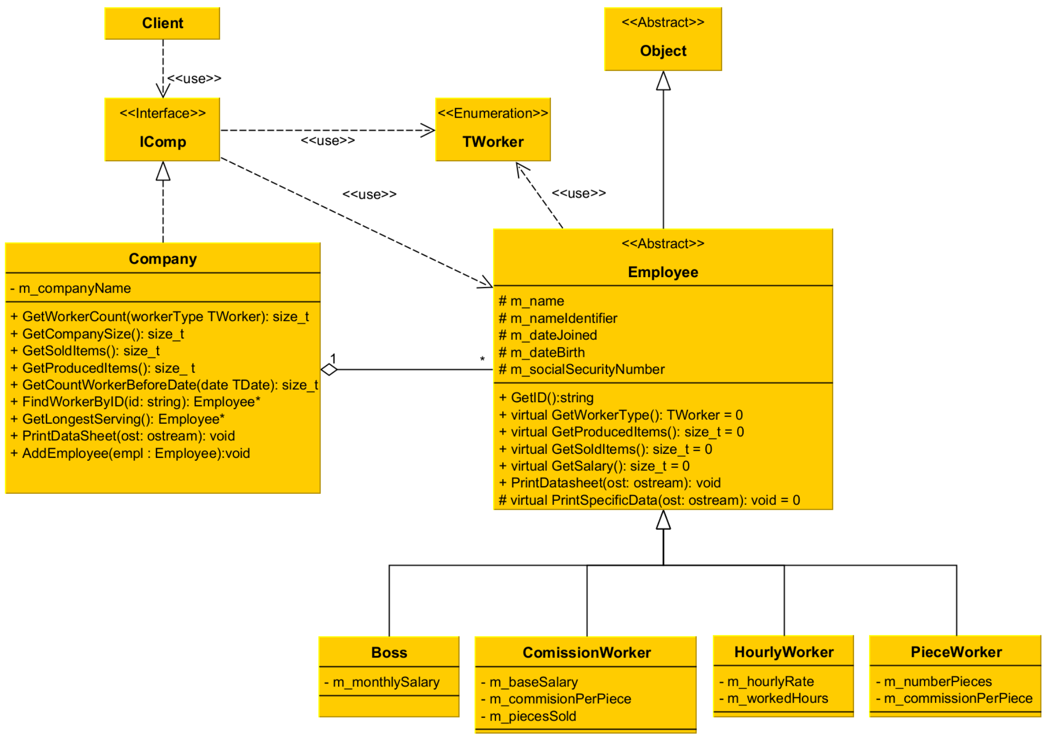
- Verwalten von Mitabeitern verschiedener Arten.
- Ausgabe von Firmen und Mitarbeiterinformationen.
- Anlegen und Entfernen von Mitarbeitern.

Funktionen der Mitarbeiter

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

3 Systementwurf

3.1 Klassendiagramm


Client
<<use>>
<<Interface>>
IComp
<<use>>
<<Enumeration>>
TWorker
<<use>>
<<use>>
<<Abstract>>
Object
Company
- m_companyName
+ GetWorkerCount(workerType TWorker): size_t
+ GetCompanySize(): size_t
+ GetSoldItems(): size_t
+ GetProducedItems(): size_ t
+ GetCountWorkerBeforeDate(date TDate): size_t
+ FindWorkerByID(id: string): Employee*
+ GetLongestServing(): Employee*
+ PrintDataSheet(ost: ostream): void
+ AddEmployee(empl : Employee):void
1
*
<<Abstract>>
Employee
# m_name
# m_nameIdentifier
# m_dateJoined
# m_dateBirth
# m_socialSecurityNumber
+ GetID():string
+ virtual GetWorkerType(): TWorker = 0
+ virtual GetProducedItems(): size_t = 0
+ virtual GetSoldItems(): size_t = 0
+ virtual GetSalary(): size_t = 0
+ PrintDatasheet(ost: ostream): void
# virtual PrintSpecificData(ost: ostream): void = 0
Boss
- m_monthlySalary
ComissionWorker
- m_baseSalary
- m_commisionPerPiece
- m_piecesSold
HourlyWorker
- m_hourlyRate
- m_workedHours
PieceWorker
- m_numberPieces
- m_commissionPerPiece

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

5 Testprotokollierung

6 Quellcode

6.1 Object.hpp

```
#ifndef OBJECT_HPP
#define OBJECT_HPP

class Object {
public:

   inline static const std::string ERROR_BAD_OSTREAM = "ERROR:␣Provided␣Ostream␣is␣bad";
   inline static const std::string ERROR_FAIL_WRITE = "ERROR:␣Fail␣to␣write␣on␣provided␣Ostream";
   inline static const std::string ERROR_NULLPTR = "ERROR:␣Passed␣in␣Nullptr!";

protected:
   Object() = default;

   virtual ~Object() = default;

};

#endif // !OBJECT_HPP
```

6.2 Client.hpp

```
#ifndef CLIENT_HPP
#define CLIENT_HPP

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

class Client {
public:

   inline static const std::string ERROR_BAD_OSTREAM = "ERROR:␣Provided␣Ostream␣is␣bad";
   inline static const std::string ERROR_FAIL_WRITE = "ERROR:␣Fail␣to␣write␣on␣provided␣Ostream";

   bool TestCompanyGetter(std::ostream & ost,IComp& company) const;
   bool TestEmptyCompanyGetter(std::ostream & ost,IComp& company) const;
   bool TestCompanyCopyCTOR(std::ostream & ost,const IComp& company,const IComp& companyCopy) const;
   bool TestCompanyAssignOp(std::ostream & ost,const IComp& company,const IComp& companyAss) const;
   bool TestCompanyPrint(std::ostream & ost,const IComp& company) const;

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

};

#endif // !CLIENT_HPP
```

6.3 Client.cpp

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

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

using namespace std;
using namespace std::chrono;

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

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";


   try {

      ComissionWorker* cWork = new ComissionWorker{ "Simon␣1", "Si1", { 2022y,November,23d }, { 2000y,November,22d }, "4711221100", 250
      ComissionWorker* cWork2 = new ComissionWorker{ "Simon␣6", "Si6", { 2022y,November,23d }, { 2000y,November,22d }, "4711221100", 25
      HourlyWorker* hWork = new HourlyWorker{ "Simon␣2", "Si2", { 2022y,November,23d }, { 1934y,November,23d },"4712231100",20,25};
      Boss* boss = new Boss{ "Simon␣3", "Si3", { 2000y,November,23d }, { 1950y,November,23d },"4712231100",35000};
      PieceWorker* pWork= new PieceWorker{ "Simon␣4", "Si4", { 2022y,November,23d }, { 2010y,November,23d },"4712231100",25,25};
      PieceWorker* pWork2= new PieceWorker{ "Simon␣5", "Si5", { 2022y,November,23d }, { 2011y,November,23d },"4712231100",25,25};

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

      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Comission␣Worker␣Cnt␣&␣Add␣Empl",  static_cast<size_t>(2), company.GetWorker
      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Houerly␣Worker␣Cnt␣&␣Add␣Empl",   static_cast<size_t>(1), company.GetWorkerC
      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Boss␣Cnt␣&␣Add␣Empl",          static_cast<size_t>(1), company.GetWorkerCoun
      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Piece␣Worker␣Cnt␣&␣Add␣Empl",     static_cast<size_t>(2), company.GetWorkerC


      TestOK = TestOK && check_dump(ost, "Test␣Company␣FindWorker␣by␣ID",        static_cast<const Employee *>(cWork), company.FindWork
      TestOK = TestOK && check_dump(ost, "Test␣Company␣FindWorker␣by␣empty␣ID",   static_cast<const Employee *>(nullptr), company.FindW

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

      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Count␣worker␣bevor␣1930␣date", static_cast<size_t>(0), company.GetCountWorke
      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣Count␣worker␣bevor␣1951␣date", static_cast<size_t>(2), company.GetCountWorke

      TestOK = TestOK && check_dump(ost, "Test␣Company␣Get␣longest␣serving␣employee", static_cast<const Employee*>(boss), company.GetLo


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

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

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

   TestEnd(ost);

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

   return TestOK;
}

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

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";


   try {

      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Comission␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetW
      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Houerly␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWor
      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Boss␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorkerCount(T
      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Piece␣Worker␣Cnt␣&␣Add␣Empl", static_cast<size_t>(0), company.GetWorke


      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣FindWorker␣by␣ID", static_cast<const Employee *>(nullptr), company.FindWor
      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣FindWorker␣by␣ID␣empty␣ID", static_cast<const Employee *>(nullptr), compan


      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Size", static_cast<size_t>(0), company.GetCompanySize());

      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Count␣worker␣bevor␣1930␣date", static_cast<size_t>(0), company.GetCoun
      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣Count␣worker␣bevor␣1951␣date", static_cast<size_t>(0), company.GetCoun

      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣longest␣serving␣employee", static_cast<const Employee*>(nullptr), comp


      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣total␣pieces␣produced", static_cast<size_t>(0), company.GetProducedIte

      TestOK = TestOK && check_dump(ost, "Test␣Empty␣Company␣Get␣total␣pieces␣sold", static_cast<size_t>(0), company.GetSoldItems());

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

   try {

      company.AddEmployee(nullptr);
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }


   TestOK = TestOK && check_dump(ost, "Test␣Company␣Add␣nullptr", Object::ERROR_NULLPTR, error_msg);

   TestEnd(ost);

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

   return TestOK;
}

bool Client::TestCompanyCopyCTOR(std::ostream& ost,const IComp& company,const IComp& companyCopy) const
{

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

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {

      stringstream result;
      stringstream expected;

      company.PrintDataSheet(expected);
      companyCopy.PrintDataSheet(result);

      TestOK = TestOK && check_dump(ost, "Test␣Company␣Copy␣Ctor", true ,expected.str() == result.str());

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

   TestEnd(ost);

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

   return TestOK;

   return false;
}

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

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {

      stringstream result;
      stringstream expected;

      company.PrintDataSheet(expected);
      companyAss.PrintDataSheet(result);

      TestOK = TestOK && check_dump(ost, "Test␣Company␣Assign␣Operator", true, expected.str() == result.str());

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

   TestEnd(ost);

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

   return TestOK;

   return false;
}

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

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

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

   try {

      company.PrintDataSheet(badstream);

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

   TestOK = TestOK && check_dump(ost, "Test␣Company␣Print␣Exception", Client::ERROR_BAD_OSTREAM, error_msg);

   TestEnd(ost);

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

   return TestOK;

   return false;
}

bool Client::TestEmployeeBoss(std::ostream& ost)
{
   if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {
      size_t testSalary = 7800;
      string svr = "4711221100";
      TDate dateBorn = { 2000y,November,22d };
      TDate dateJoined = { 2022y,November,23d };
      string name = "Max␣Musterman";
      string id = "MAX";

      Boss testBoss{ name, id, dateJoined, dateBorn, svr, testSalary};

      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetSalary()", testSalary, testBoss.GetSalary());
      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetSoldItems()", static_cast<size_t>(0), testBoss.GetSoldItems());
      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetProducedItems()", static_cast<size_t>(0), testBoss.GetProducedItems());
      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetWorkerType()", E_Boss, testBoss.GetWorkerType());
      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetDateBirth()", dateBorn, testBoss.GetDateBirth());
      TestOK = TestOK && check_dump(ost, "Test␣-␣Boss.GetDateJoined()", dateJoined, testBoss.GetDateJoined());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   TestOK = TestOK && check_dump(ost, "Test␣-␣error␣buffer", error_msg.empty(), true);
   TestEnd(ost);
   return TestOK;
}

bool Client::TestEmployeeHourlyWorker(std::ostream& ost)
{
   if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {
      size_t hourlyRate = 21;
      size_t workedHours = 160;
      string svr = "4711221100";
      TDate dateBorn = { 2000y,November,22d };
      TDate dateJoined = { 2022y,November,23d };
      string name = "Max␣Musterman";
      string id = "MAX";

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

      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetSalary()", hourlyRate * workedHours, testHourlyWorker.GetSalary());
      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems())
      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProduced
      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetWorkerType()", E_HourlyWorker, testHourlyWorker.GetWorkerType());
      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
      TestOK = TestOK && check_dump(ost, "Test␣-␣HourlyWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   TestOK = TestOK && check_dump(ost, "Test␣-␣error␣buffer", error_msg.empty(), true);
   TestEnd(ost);
   return TestOK;
}

bool Client::TestEmployeePieceWorker(std::ostream& ost)
{
   if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {
      size_t piecesProduced = 950;
      size_t comissionPerPiece = 2;
      string svr = "4711221100";
      TDate dateBorn = { 2000y,November,22d };
      TDate dateJoined = { 2022y,November,23d };
      string name = "Max␣Musterman";
      string id = "MAX";

      PieceWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, piecesProduced, comissionPerPiece };

      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetSalary()", piecesProduced * comissionPerPiece, testHourlyWorker.GetSala
      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetSoldItems()", static_cast<size_t>(0), testHourlyWorker.GetSoldItems());
      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetProducedItems()", piecesProduced, testHourlyWorker.GetProducedItems());
      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetWorkerType()", E_PieceWorker, testHourlyWorker.GetWorkerType());
      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
      TestOK = TestOK && check_dump(ost, "Test␣-␣PieceWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   TestOK = TestOK && check_dump(ost, "Test␣-␣error␣buffer", error_msg.empty(), true);
   TestEnd(ost);
   return TestOK;
}

bool Client::TestEmployeeComissionWorker(std::ostream& ost)
{
   if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;

   TestStart(ost);

   bool TestOK = true;
   string error_msg = "";

   try {
      size_t baseSalary = 2300;
      size_t piecesSold = 300;
      size_t comissionPerPiece = 2;
      string svr = "4711221100";
      TDate dateBorn = { 2000y,November,22d };
      TDate dateJoined = { 2022y,November,23d };
      string name = "Max␣Musterman";
      string id = "MAX";

      ComissionWorker testHourlyWorker{ name, id, dateJoined, dateBorn, svr, baseSalary, comissionPerPiece, piecesSold };

      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetSalary()", baseSalary + piecesSold * comissionPerPiece, testHourlyW
      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetSoldItems()", piecesSold, testHourlyWorker.GetSoldItems());
      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetProducedItems()", static_cast<size_t>(0), testHourlyWorker.GetProdu
      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetWorkerType()", E_CommisionWorker, testHourlyWorker.GetWorkerType())
      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetDateBirth()", dateBorn, testHourlyWorker.GetDateBirth());
      TestOK = TestOK && check_dump(ost, "Test␣-␣ComissionWorker.GetDateJoined()", dateJoined, testHourlyWorker.GetDateJoined());
   }
   catch (const string& err) {
      error_msg = err;
   }
   catch (bad_alloc const& error) {
      error_msg = error.what();
   }
   catch (const exception& err) {
      error_msg = err.what();
   }
   catch (...) {
      error_msg = "Unhandelt␣Exception";
   }

   TestOK = TestOK && check_dump(ost, "Test␣-␣error␣buffer", error_msg.empty(), true);
   TestEnd(ost);
   return TestOK;
}
```

6.4 IComp.hpp

```
#ifndef ICOMP_HPP
#define ICOMP_HPP

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

class IComp{
public:

   virtual size_t GetCompanySize() const = 0;

   virtual size_t GetWorkerCount(const TWorker & workerType) const = 0;

   virtual size_t GetSoldItems() const = 0;

   virtual size_t GetProducedItems() const = 0;

   virtual size_t GetCountWorkerBeforDate(const TDate & date) const = 0;

   virtual Employee const * FindWorkerByID(const std::string & id) const = 0;

   virtual Employee const * GetLongestServing(void) const = 0;

   virtual std::ostream& PrintDataSheet(std::ostream& ost) const = 0;

   virtual void AddEmployee(Employee const* empl) = 0;

   virtual ~IComp() = default;

};

#endif // !ICOMP_HPP
```

6.5 Company.hpp

```cpp
#ifndef COMPANY_HPP
#define COMPANY_HPP

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

using  TContEmployee = std::map<const std::string,Employee const *>;

class Company : public Object, public IComp{
public:

   Company(std::string name) : m_companyName{ name } {}

   Company(const Company & comp);

   void operator=(Company comp);

   virtual void AddEmployee(Employee const* empl) override;

   virtual size_t GetCompanySize() const override;

   virtual size_t GetWorkerCount(const TWorker& workerType) const override;

   virtual size_t GetSoldItems() const override;

   virtual size_t GetProducedItems() const override;

   virtual size_t GetCountWorkerBeforDate(const TDate& date) const override;

   virtual Employee const * FindWorkerByID(const std::string& id) const override;

   virtual Employee const * GetLongestServing(void) const override;

   virtual std::ostream& PrintDataSheet(std::ostream& ost) const override;

   ~Company();

private:

   std::string m_companyName;
   TContEmployee m_Employees;
};

#endif // !COMPANY_HPP
```

6.6 Company.cpp

```cpp
#include <algorithm>
#include <numeric>
#include <iostream>
#include "Company.hpp"
#include "Employee.hpp"
using namespace std;

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

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

/**
```

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

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

void Company::AddEmployee(Employee const* empl)
{
   if (empl == nullptr) throw Object::ERROR_NULLPTR;
   else m_Employees.insert({empl->GetID(),empl});
}

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

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

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

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

size_t Company::GetWorkerCount(const TWorker& workerType) const
{
   return count_if(m_Employees.cbegin(), m_Employees.cend(),
               [&](auto& e) {return e.second->GetWorkerType() == workerType;});
}

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

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

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

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

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

// longest serving ist glaub ich auf das Dienstalter und nicht auf den
// Geburtstag bezogen - TDate Employee::GetDateJoined()
Employee const * Company::GetLongestServing(void) const
{
   auto minElem = min_element(m_Employees.cbegin(), m_Employees.cend(),
      [](const auto& lhs, const auto& rhs) { return lhs.second->GetDateJoined() < rhs.second->GetDateJoined();});


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

}

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

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

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

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

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

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

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

   return ost;
}

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

   m_Employees.clear();
}
```

6.7 TWorker.hpp

```
#ifndef TWORKER_HPP
#define TWORKER_HPP

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

#endif // !TWORKER_HPP
```

6.8 Employee.hpp

```cpp
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

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

using TDate = std::chrono::year_month_day;

class Employee : public Object
{
public:

    inline static const std::string ERROR_BAD_ID = "ERROR:␣An␣employees␣ID␣is␣limited␣to␣3␣characters.";

    std::string GetID() const;

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

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

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

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

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

    /** Pure Virtual Function
     * \brief returns date of birth of a given worker.
     * \return TDate
     */
    virtual TDate GetDateBirth() const;

    /** Pure Virtual Function
     * \brief returns the date a worker.
     * has started working at the company.
     * \return TDate
     */
    virtual TDate GetDateJoined() const;

    /**
     * \brief Prints information about a worker.
     * \return std::ostream&
     */
    std::ostream& PrintDatasheet(std::ostream& ost) const;


    /** Pure virtual function
    * \brief Prints specific information for a type of worker.
    * \return std::ostream&
    */
    virtual std::ostream& PrintSpecificData(std::ostream& ost) const = 0;

    /** Pure virtual function
    * \brief creates a copy of the worker and puts it on the heap.
    * \return Employee*
    */
    virtual Employee* Clone() const = 0;


protected:

    std::string m_name;
    std::string m_nameIdentifier;
    TDate m_dateJoined;
    TDate m_dateBirth;
    std::string m_socialSecurityNumber;
};

#endif // EMPLOYEE_H
```

6.9 Employee.cpp

```
#include "Employee.hpp"

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

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

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

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

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

    ost << "Datenblatt\n---------------\n";
    ost << "Name:␣" << m_name << std::endl;
    ost << "Kuerzel:␣" << m_nameIdentifier << std::endl;
    ost << "Sozialversicherungsnummer:␣" << m_socialSecurityNumber << std::endl;
    ost << "Geburtstag:␣" << m_dateBirth << std::endl;
    ost << "Einstiegsjahr:␣" << m_dateJoined.year() << std::endl;

    PrintSpecificData(ost);

    ost << std::endl;

    return ost;
}
```

6.10 Boss.hpp

```
#ifndef BOSS_H
#define BOSS_H

#include "Employee.hpp"

class Boss : public Employee
{
public:

    Boss() = default;

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

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

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

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

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

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

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

private:
    size_t m_salary;
};

#endif // BOSS_H
```

6.11 Boss.cpp

```
#include "Boss.hpp"

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

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

    return ost;
}

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

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

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

6.12 HourlyWorker.hpp

```
#ifndef HOURLY_WORKER_HPP
```

```
#define HOURLY_WORKER_HPP

#include "Employee.hpp"

class HourlyWorker : public Employee
{
public:

    HourlyWorker() = default;

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

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

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

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

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

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

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

private:
    size_t m_hourlyRate;
    size_t m_workedHours;
};

#endif // !HOURLY_WORKER_HPP
```

6.13 HourlyWorker.cpp

```cpp
#include "HourlyWorker.hpp"

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

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

    return ost;
}

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

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

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

6.14 PieceWorker.hpp

```cpp
#ifndef PIECE_WORKER_H
#define PIECE_WORKER_H

#include "Employee.hpp"

class PieceWorker : public Employee
{
public:

    PieceWorker() = default;

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

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

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

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

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

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

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

private:
    size_t m_numberPieces;
    size_t m_commisionPerPiece;
};

#endif // !PIECE_WORKER_H
```

6.15 PieceWorker.cpp

```
#include "PieceWorker.hpp"

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

std::ostream& PieceWorker::PrintSpecificData(std::ostream& ost) const
{
    if (ost.bad())
    {
        throw Object::ERROR_BAD_OSTREAM;
```

```cpp
        return ost;
    }
    ost << "Role:␣PieceWorker" << std::endl;
    ost << "Pieces␣produced:␣" << m_numberPieces << std::endl;
    ost << "Pay␣per␣piece:␣" << m_commisionPerPiece << "␣EUR" << std::endl;

    return ost;
}

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

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

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

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

6.16 ComissionWorker.hpp

```cpp
#ifndef COMISSION_WORKER_H
#define COMISSION_WORKER_H

#include "Employee.hpp"

class ComissionWorker : public Employee
{
public:

    ComissionWorker() = default;

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

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

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

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

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

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

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

private:
    size_t m_baseSalary;
    size_t m_commisionPerPiece;
    size_t m_piecesSold;
};

#endif // !COMISSION_WORKER_H
```

6.17 ComissionWorker.cpp

```
#include "ComissionWorker.hpp"

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

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

    return ost;
}

size_t ComissionWorker::GetSoldItems() const
```

```
{
    return m_piecesSold;
}

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

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

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

6.18 main.cpp

```
#include "Company.hpp"
#include "Employee.hpp"
#include "HourlyWorker.hpp"
#include "vld.h"
#include "Client.hpp"
#include "Test.hpp"
#include "ComissionWorker.hpp"
#include "HourlyWorker.hpp"
#include "Boss.hpp"
#include "PieceWorker.hpp"
#include <iostream>
#include <fstream>

using namespace std;
using namespace std::chrono;

#define WRITE_OUTPUT false

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

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

    Company comp{"Offenberger_Devices"};
    Client TestClient;


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

    // Copy Ctor Call !
    Company compCopy = comp;

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

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

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


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

    Company emptyComp{"empty"};

    TestOK = TestOK && TestClient.TestEmptyCompanyGetter(cout, emptyComp);
    if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmptyCompanyGetter(testoutput, emptyComp);

    // Test Boss
    TestOK = TestOK && TestClient.TestEmployeeBoss(cout);
    if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeBoss(testoutput);

    // Test Hourly Worker
    TestOK = TestOK && TestClient.TestEmployeeHourlyWorker(cout);
    if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeHourlyWorker(testoutput);

    // Test Piece Worker
    TestOK = TestOK && TestClient.TestEmployeePieceWorker(cout);
    if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeePieceWorker(testoutput);

    // Test Comission Worker
    TestOK = TestOK && TestClient.TestEmployeeComissionWorker(cout);
    if (WRITE_OUTPUT) TestOK = TestOK && TestClient.TestEmployeeComissionWorker(testoutput);

    if (WRITE_OUTPUT){
       if (TestOK) TestCaseOK(testoutput);
       else TestCaseFail(testoutput);

       testoutput.close();
    }

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

}
```