

Übung 05

Arbeitsaufwand insgesamt: 5h

Inhaltsverzeichnis

Inhaltsverzeichnis

Übung 05	1
Teil 1 – Flugreisen.....	2
Lösungsidee:	2
Lösung:	2
Source-Code:	3
Testfälle	11
Teil 2 – Stücklistenverwaltung.....	12
Lösungsidee:	12
Lösung:	12
Source-Code:	13
Testfälle	30

Teil 1 – Flugreisen

Ziel dieser Übung ist die Modellierung von Klassen für die Verwaltung eines Reisebüros. Es sollen mindestens Flüge, Personen und Flugreisen verwaltet werden können.

Lösungsidee:

Durch die Angabe sind bereits die meisten Komponenten und Funktionen vorgegeben.

Zusätzlich zu den angegebenen Datenkomponenten würde ich auch noch überladene Konstruktoren (mit den jeweiligen Datenkomponenten als Parametern) und eine Klasse DateTime implementieren. Die Klasse DateTime einzig und allein aus dem Grund, weil ich die Flugdauer dynamisch berechnen, sowie das Datum gespeichert speichern will. C++ sowie sämtliche Libraries die ich gefunden habe stellen keine anständige und leichte Implementierung bereit. Dabei wird jedoch nur das (zurzeit) notwendige implementiert.

Aufgrund der vagen Beschreibung des Beispiels werde ich nur die Klassen wie oben beschrieben, sowie eine anständige Ausgabe über den Operator, implementieren. Endprodukt sind für mich also lediglich die Klassen mit den Datenkomponenten, dem überladenen Konstruktor, sowie mit dem operator<<.

Lösung:

Als C++ Projekt „Teil_1“ im Archiv

Nachfolgender Source-Code

Source-Code:

Main.cpp

```
#include "DateTime.h"
#include "flight.h"
#include "flight_journey.h"
#include "Person.h"
#include<iostream>

void test_normal()
{
    Flight f1("F23000", "Dumbo Airlines", "Aegypten", "Berlin", "24.12.2020 08:50:00",
"24.12.2020 11:00:00");
    Flight f2("F23001", "Peter Pan Inc.", "London", "Moskau", "24.12.2020 12:00:00",
"24.12.2020 14:15:00");
    Flight f3("F23002", "Austrian Airlines", "Wien", "Hagenberg", "24.12.2020
15:10:00", "24.12.2020 16:00:00");

    Person p1("Stefan", "Raab", 44, "Koeln", "DE501203830");
    Person p2("Vladimir", "Klitschko", 10, "Novigrad", "RU2109302314");

    FlightJourney fj({ f1,f2,f3 }, { p1,p2 });
    std::cout << fj;
}
void test_empty()
{
    FlightJourney emptyfj({}, {});
    std::cout << emptyfj;
}
void test_date()
{
    DateTime test("24.12.2020 08:50:00");
    DateTime testfuture("24.12.2021 08:50:00");
    DateTime testpast("24.12.2020 10:00:00");
    std::cout << test << std::endl;
    std::cout << DateTime::diff(testpast, testfuture) << std::endl;
}

void main()
{
    test_normal();
    //test_empty();
    //test_date();
}
```

Flight.h

```
#ifndef FLIGHT_H
#define FLIGHT_H
#include<iostream>
#include<vector>
#include "datetime.h"
#include "person.h"
#include<iostream>
using std::string;
using std::vector;
using std::ostream;

class Flight
{
private:
    string flightno;
    string airline;
    string origin;
    string destination;
    DateTime departure;
    DateTime arrival;
    DateTime duration;

public:
    friend ostream& operator<<(ostream& lhs, Flight const& rhs);

    Flight(string const& flightno, string const& airline, string const& origin, string
const& destination, string const& dep, string const& arr)
    {
        this->flightno = flightno;
        this->airline = airline;
        this->origin = origin;
        this->destination = destination;
        this->departure = DateTime(dep);
        this->arrival = DateTime(arr);
        this->duration = DateTime::diff(departure, arrival);
    }
};

ostream& operator<<(ostream& lhs, Flight const& rhs)
{
    lhs << rhs.origin << " -> " << rhs.destination;
    return lhs;
}

#endif
```

Person.h

```
#ifndef PERSON_H
#define PERSON_H
#include<iostream>
using std::string;
using std::ostream;

class Person
{
    enum class gender { MALE, FEMALE, OTHER };

private:
    string firstname;
    string surname;
    size_t age;
    string city;
    string creditcardno;

public:
    friend ostream& operator<<(ostream& lhs, Person const& rhs);

    Person(string const& _firstname, string const& _surname, size_t const& _age, string
const& _city, string const& _creditcardno)
    {
        firstname = _firstname;
        surname = _surname;
        age = _age;
        city = _city;
        creditcardno = _creditcardno;
    }
};

ostream& operator<<(ostream& lhs, Person const& rhs)
{
    lhs << rhs.firstname << " " << rhs.surname;
    return lhs;
}

#endif
```

DateTime.h

```

#ifndef DATETIME_H
#define DATETIME_H
#include<iostream>
#include<string>
#include<regex>
#include<time.h>
using std::ostream;

using std::string;
using std::to_string;

class DateTime {
private:
    struct tm ti;

    std::vector<int>months = { 31, //January
                              59, //February
                              90, //March 31
                              120, //April 30
                              151, //May 31
                              181, //June 30
                              212, //July 31
                              243, //August 31
                              273, //September 30
                              304, //October 31
                              334, //November 30
                              365 //December 31
    };
    bool isDiff;

    void strtodate(string const& datestr)
    {
        std::regex date_regex("^([0-9][0-9]?)[ ]([0-9][0-9]?)[ ]([0-9][0-9]|([0-9][0-9][0-9][0-9])|([0-9][0-9]?):([0-9][0-9]?):([0-9][0-9]?))");
        std::smatch matches;

        if (std::regex_search(datestr, matches, date_regex) && matches.size()==7)
        {
            ti.tm_mday = std::stoi(matches[1].str());
            ti.tm_mon = std::stoi(matches[2].str())-1;
            string yearStr = matches[3].str();
            if (yearStr.length() == 2)
            {
                yearStr = "20" + yearStr;
            }
            ti.tm_year = std::stoi(yearStr)-1900;
            ti.tm_hour = std::stoi(matches[4].str());
            ti.tm_min = std::stoi(matches[5].str());
            ti.tm_sec = std::stoi(matches[6].str());
        }
        else
        {
            std::cout << "Error parsing date.\n";
            *this = DateTime();
        }
    }
    string formatdd(short const num) const
    {

```

```

        if (num < 10)
        {
            return "0" + to_string(num);
        }
        return to_string(num);
    }

public:
    friend ostream& operator<<(ostream& lhs, DateTime const& rhs);

    DateTime()
    {
        std::time_t t = std::time(0);
        gmtime_s(&ti, &t);
        isDiff = false;
    }

    DateTime(short const day, short const month, short const year, short const hour,
short const min, short const sec)
    {
        struct tm tmp;
        tmp.tm_mday = day;
        tmp.tm_mon = month - 1;
        tmp.tm_year = year - 1900;
        tmp.tm_hour = hour;
        tmp.tm_min = min;
        tmp.tm_sec = sec;
        if (mktime(&tmp) == -1)
        {
            std::cout << "Invalid date given.\n";
            *this = DateTime();
        }
        ti = tmp;
        isDiff = false;
    }

    DateTime(string const& s)
    {
        strtodate(s);
    }

    string toStr() const
    {
        return isDiff ? std::to_string(ti.tm_mday) + " days, " +
std::to_string(ti.tm_hour) + " hours, " + std::to_string(ti.tm_min) + " minutes, " +
std::to_string(ti.tm_sec) + " seconds" : (formatdd(ti.tm_mday) + "." +
formatdd(ti.tm_mon+1) + "." + to_string(ti.tm_year+1900) + " " + formatdd(ti.tm_hour) +
":" + formatdd(ti.tm_min) + ":" + formatdd(ti.tm_sec));
    }

    static DateTime diff(DateTime& time1, DateTime& time2)
    {
        DateTime tmp = DateTime();

        int seconds = std::abs(mktime(&(time1.ti)) - mktime(&(time2.ti)));
        tmp.ti.tm_sec = seconds % 60;
        tmp.ti.tm_min = (seconds / 60) % 60;
        tmp.ti.tm_hour = (seconds / 60 / 60) % 24;
        tmp.ti.tm_mday = (seconds / 60 / 60 / 24);
    }

```

```
        tmp.isDiff = true;
        return tmp;
    }

};

ostream& operator<<(ostream& lhs, DateTime const& rhs)
{
    lhs << rhs.toStr();
    return lhs;
}
#endif
```


Flight_journey.h

```

#ifndef FLIGHT_JOURNEY_H
#define FLIGHT_JOURNEY_H
#include "flight.h"
#include "person.h"
#include<vector>
#include<iostream>
using std::vector;
using std::ostream;

class FlightJourney{
private:
    vector<Flight> flights;
    vector<Person> people;

public:
    friend ostream& operator<<(ostream& lhs, FlightJourney const& rhs);

    FlightJourney(vector<Flight> const& flights, vector<Person> const& people)
    {
        this->flights = flights;
        this->people = people;
    }

    void printFlights(ostream& out = std::cout) const
    {
        if(flights.size() == 0)
        {
            out << "No flights found for this journey.\n";
        }
        else
        {
            for (int i(0); i < flights.size(); i++)
            {
                if (i > 0)
                {
                    out << " -> ";
                }
                out << flights[i];
            }
            out << std::endl;
        }
    }

    void printPeople(ostream& out = std::cout) const
    {
        if(people.size() == 0)
        {
            out << "No people attending this journey.\n";
        }
        else
        {
            for (int i(0); i < people.size(); i++)
            {
                if (i > 0)
                {
                    out << ", ";
                }
                out << people[i];
            }
        }
    }
}

```

```
        out << std::endl;
    }
}

};

ostream& operator<<(ostream& lhs, FlightJourney const& rhs)
{
    rhs.printFlights(lhs);
    rhs.printPeople(lhs);
    return lhs;
}

#endif
```

Testfälle

- Normale Benützung
- Leere Flugreise (keine Flüge, keine Personen)
- DateTime Funktionalität

Alle Testfälle sind zusätzlich als Funktionen in main.cpp enthalten.

Normale Benützung:

Aegypten -> Berlin -> London -> Moskau -> Wien -> Hagenberg
Stefan Raab, Vladimir Klitschko

Leere Flugreise:

No flights found for this journey.
No people attending this journey.

DateTime Funktionalität:

Error parsing date.
23.12.2020 22:10:01
364 days, 22 hours, 50 minutes, 0 seconds

Selbst bei Eingabe eines falschen Datums funktioniert das Beispiel. Es wird dann ein DateTime mit dem momentanen Datum erzeugt.

Teil 2 – Stücklistenverwaltung

Ziel der Übung ist die Implementierung einer Stücklistenverwaltung.

Lösungsidee:

Aus dem UML ist ersichtlich, dass es grundsätzlich 2 Klassen „Part“ und „CompositePart“ geben soll.

Ein CompositePart wird abgeleitet von einem Part und speichert zudem auch noch eine Liste von weiteren Teilen, aus denen es besteht.

Zusätzlich dazu soll es noch zwei Klassen SetFormatter und HierarchyFormatter, welche von der Abstrakten Klasse Formatter erben, geben. Für den SetFormatter würde ich eine Map anlegen aus String (für den Namen) und einem Int zum Zählen. Danach würde ich rekursiv die Liste/Vektor der Elemente für das Part durchgehen und dementsprechend jedes Element hochzählen. Nachdem Initialisieren der Map können die Elemente angenehm ausgegeben werden.

Hierfür ist jedoch eine polymorphe Klasse, also mindestens eine virtuelle Methode, notwendig. Dabei habe ich mich für die Funktion „equals“ entschieden. Ein Part wird durch den Namen identifiziert. Ein CompositePart jedoch aus dem Namen sowie dem gesamten Vektor bzw. allen Unterteilen.

Beim HierarchyFormatter würde ich ähnlich vorgehen und rekursiv durch die Liste iterieren, jedoch die Elemente sofort ausgeben.

Zum Speichern der Elemente habe ich mich für das CSV Format entschieden. Für jeden Part in der Liste bzw. für das Part selbst wird eine Zeile im CSV erzeugt und ähnlich der Ausgabe vom HierarchyFormatter gespeichert. Eine Zeile enthält: „Ebene,0 oder 1 (Part oder CompositePart),Name“. Die Ebene beschreibt dabei, das wievielte Unterkind in der Liste das Element ist. „Sitzgarnitur -> Sessel -> Bein“ wäre zum Beispiel die Tiefe 2, während Sitzgarnitur der „Wurzelknoten“ bzw. das ursprüngliche Part ist und den Eintrag 0 enthält.

Beim Speichern wird die Liste rekursiv in der gleichen Reihenfolge wie der HierarchyFormatter durchlaufen und dann für jeden Part eine Zeile erzeugt.

Umgekehrt wird beim Wiederherstellen (load) dann geschaut, ob es sich um die gleiche Ebene handelt (Kindobjekt oder nicht), ob es ein Part oder CompositePart ist und welchen Namen das Element hat. Mithilfe von diesen Informationen kann dann das ursprüngliche Objekt wiederhergestellt werden.

Lösung:

Als C++ Projekt „Teil_2“ im Archiv

Nachfolgender Source-Code

Source-Code:

Main.cpp

```
#include "part.h"
#include "formatter.h"
using PartsLists::Part;
using PartsLists::CompositePart;

void test_reference(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    setf.printParts(&sitzgarnitur);
    hierf.printParts(&sitzgarnitur);
}

void test_alter(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
```

```

    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    Part polster("Polster");
    CompositePart polsterSessel("Sessel");
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&sitzf);
    polsterSessel.addPart(&polster);

    sitzgarnitur.getParts()[0] = &polsterSessel;
    CompositePart* sub = dynamic_cast<CompositePart*>(sitzgarnitur.getParts()[0]);
    sub->getParts()[0] = &tisch;

    setf.printParts(&sitzgarnitur);
    hierf.printParts(&sitzgarnitur);
}

void test_empty(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    CompositePart test("null_part");
    test.addPart(nullptr);

    setf.printParts(&test);
    hierf.printParts(&test);
}

void test_equals(formatting::HierarchyFormatter hierf)
{
    Part p1("Sessel");
    Part p2("Sessel");
    Part p3("Polster");
    if(p1.equals(&p2))
    {
        std::cout << "Parts are equal.\n";
    }
    else
    {
        std::cout << "Parts are unequal.\n";
    }
    std::cout << "Comparison:\n";
    hierf.printParts(&p1);
    hierf.printParts(&p2);

    CompositePart sesselpolster1("Sesselpolster");
    sesselpolster1.addPart(&p2);
    sesselpolster1.addPart(&p3);

    Part p4("Stuhl");
    CompositePart sesselpolster2("Sesselpolster");

```

```

    sesselpolster2.addPart(&p4);
    sesselpolster2.addPart(&p3);

    if(sesselpolster1.equals(&sesselpolster2))
    {
        std::cout << "Compositeparts are equal.\n";
    }
    else
    {
        std::cout << "Compositeparts are unequal.\n";
    }
    std::cout << "Comparison:\n";
    hierf.printParts(&sesselpolster1);
    hierf.printParts(&sesselpolster2);

    std::cout << "Swapping 'Stuhl' for 'Sessel':\n";
    sesselpolster2.getParts()[0] = &p2;

    hierf.printParts(&sesselpolster1);
    hierf.printParts(&sesselpolster2);

    if (sesselpolster1.equals(&sesselpolster2))
    {
        std::cout << "Now they are equal!\n";
    }
}

void test_store(formatting::HierarchyFormatter hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    sitzgarnitur.store("sitzgarnitur.csv");

    hierf.printParts(&sitzgarnitur);
}

void test_load(formatting::HierarchyFormatter hierf)

```

```

{
    CompositePart copy("Platzhalter");
    copy.load("sitzgarnitur.csv");

    hierf.printParts(&copy);
}

void main()
{
    formatting::SetFormatter setf;
    formatting::HierarchyFormatter hierf;

    test_reference(setf, hierf);
    //test_alter(setf, hierf);
    //test_empty(setf, hierf);
    //test_equals(setf);
    //test_store(hierf);
    //test_load(hierf);
}

```

Part.h

```

#ifndef PART_H
#define PART_H

#include<iostream>
#include<string>
#include<vector>
#include "storable.h"
#include<fstream>

using std::string;
using std::cout;
using std::vector;
using std::ofstream;
using std::ifstream;

namespace PartsLists {
    class Part : Storable {
    private:
        string name;
    protected:
        // Set the name to the desired string.
        void setName(string const& s);
    public:
        // Vector of Part-Pointers.
        typedef vector<Part*> partList;
        // Overloaded constructor: name
        Part(string const& name);
        // Returns the name of the part.
        string getName() const;
        // Returns whether or not a part is equal to another.
        virtual bool equals(Part* const o) const;

        // Stores the file to the disk under the designated name.
        void store(string const& fname) const override;
        // Loads the part from a file with the given name.
        void load(string const& fname) override;
    };
}

```



```

    };

    class CompositePart : public Part {
    private:
        partList parts;
        // Stores the file to the disk under the designated name.
        void storeWorker(ofstream& out, size_t const level,
vector<PartsLists::Part*> const& list) const;
        // Loads the part from a file with the given name.
        void loadWorker(istream& in, size_t const level, string& currentline, Part*
p);
    protected:
        // Returns whether or not two partLists are equal
        bool cmpPartList(partList const& p1, partList const& p2) const;
    public:
        // Overloaded Constructor: name
        CompositePart(string const& name);
        // Adds a part to the partlist
        void addPart(Part* const p);
        // Returns all subparts.
        partList getParts() const;
        // Returns all subparts (editable)
        partList& getParts();
        // Returns whether or not two Parts/Compositeparts are equal.
        bool equals(Part* const o) const override;

        // Stores the current CompositePart to the disk under the designated
filename.
        void store(string const& fname) const override;
        // Restores the compositepart from disk using the given file.
        void load(string const& fname) override;
    };
}

#endif

```

Formatter.h

```

#ifndef FORMATTER_H
#define FORMATTER_H
#include "part.h"
#include<vector>
#include<map>
using std::map;
using std::vector;

namespace formatting {
    class Formatter {
    public:
        virtual void printParts(PartsLists::Part* const p) const = 0;
    };

    class SetFormatter : public Formatter {
    private:
        // Creates a map to output for printparts
        void buildSet(PartsLists::Part* const part, std::map<string, size_t>&
partsSet) const;
    public:
        // Prints the amount of each part. Only counts leafs.

```

```

        void printParts(PartsLists::Part* const p) const override;
    };

    class HierarchyFormatter : public Formatter {
    private:
        // Recursive worker for printparts.
        void printPartsWorker(PartsLists::Part* const p, size_t const indent) const;
    public:
        // Prints each part as a hierarchy.
        void printParts(PartsLists::Part* const p) const override;
    };
}

#endif

Storable.h
#ifndef FORMATTER_H
#define FORMATTER_H
#include "part.h"
#include<vector>
#include<map>
using std::map;
using std::vector;

namespace formatting {
    class Formatter {
    public:
        virtual void printParts(PartsLists::Part* const p) const = 0;
    };

    class SetFormatter : public Formatter {
    private:
        // Creates a map to output for printparts
        void buildSet(PartsLists::Part* const part, std::map<string, size_t>&
partsSet) const;
    public:
        // Prints the amount of each part. Only counts leafs.
        void printParts(PartsLists::Part* const p) const override;
    };

    class HierarchyFormatter : public Formatter {
    private:
        // Recursive worker for printparts.
        void printPartsWorker(PartsLists::Part* const p, size_t const indent) const;
    public:
        // Prints each part as a hierarchy.
        void printParts(PartsLists::Part* const p) const override;
    };
}

#endif

```

Helper_funcs.h

```

#ifndef HELPER_FUNCS_H
#define HELPER_FUNCS_H
#include<iostream>
#include<vector>
using std::string;
using std::vector;

// Compares two strings. Returns 0 if equal, -1 if smaller and +1 if greater than the
// other string. (lexicographically)
int strcmp(string const& name1, string const& name2)
{
    if (name1.length() != name2.length())
    {
        return -1;
    }

    string tmp1 = name1;
    string tmp2 = name2;

    for (size_t n(0); n < name1.length(); n++)
    {
        tmp1[n] = std::tolower(name1[n]);
        tmp2[n] = std::tolower(name2[n]);
    }

    return tmp1.compare(tmp2);
}

// Returns a vector of elements separated by the specified delimiter
vector<string> split_string(string const& str, char const delimiter)
{
    int offset(0);
    int element_end(0);
    std::vector<string> elements;

    while (str.find(delimiter, offset) != string::npos)
    {
        //calculate where the next delimiter is and get the substring
        int element_end = str.find(delimiter, offset);
        elements.push_back(str.substr(offset, element_end - offset));
        offset = element_end + 1;
    }

    //if there is no more delimiter, but there are still characters after the last
    //delimiter, add the remaining string to the vector
    if (str.length() - 1 >= offset)
    {
        elements.push_back(str.substr(offset, str.length() - offset));
    }

    return elements;
}

#endif

```

Part.h

```
#include "part.h"
#include "helper_funcs.h"
#include<fstream>
#include<vector>
using std::vector;
using std::ofstream;
using std::ifstream;

// Part

PartsLists::Part::Part(string const& name) {
    this->name = name;
}
string PartsLists::Part::getName() const {
    return name;
}
void PartsLists::Part::setName(string const& s)
{
    name = s;
}
bool PartsLists::Part::equals(Part* const o) const {
    // Normal parts are identified by their name.
    return strcmp(name, o->name)==0;
}

void PartsLists::Part::store(string const& fname) const
{
    // For normal parts one line is sufficient, since they don't have an embedded
    partslist.
    ofstream out(fname);
    out << 0 << "," << 0 << "," << getName() << std::endl;
}

void PartsLists::Part::load(string const& fname)
{
    // For normal parts reading one line is sufficient since they only carry so much
    data.
    ifstream in(fname, 'w');
    if(in)
    {
        string line;
        std::getline(in, line);
        vector<string> elements(split_string(line, ','));
        if(elements.size()==3)
        {
            name = elements[2];
        }
        else
        {
            std::cout << "Error loading from file. Invalid format.\n";
        }
    }
}

// CompositePart
//Delegate to Part(name)
PartsLists::CompositePart::CompositePart(string const& name) : Part(name) { }
```

```

void PartsLists::CompositePart::addPart(Part* const p)
{
    parts.push_back(p);
}

PartsLists::Part::partList PartsLists::CompositePart::getParts() const {
    return parts;
}
PartsLists::Part::partList& PartsLists::CompositePart::getParts() {
    return parts;
}

bool PartsLists::CompositePart::cmpPartList(partList const& p1, partList const& p2)
const
{
    // If they differ in size, they surely aren't equal.
    if (p1.size() != p2.size())
    {
        return false;
    }

    // If they have the same size, we need to check if they contain the same items.
    bool allOk(true);
    size_t id1(0);
    size_t id2(0);
    // We iterate through all parts or til there is a mismatch.
    while(allOk && id1<p1.size())
    {
        // We check for each item in p1 if it is present in p2.
        bool curOkay(false);
        while(!curOkay && id2<p2.size())
        {
            if (p1[id1] == nullptr && p2[id2] == nullptr)
                curOkay = true;
            else if(p1[id1] != nullptr && p2[id2] != nullptr)
            {
                PartsLists::CompositePart* cpart1 =
dynamic_cast<PartsLists::CompositePart*>(p1[id1]);
                PartsLists::CompositePart* cpart2 =
dynamic_cast<PartsLists::CompositePart*>(p2[id2]);

                if (cpart1 == nullptr && cpart2 == nullptr && p1[id1]-
>equals(p2[id2]))
                    curOkay = true;
                else if (cpart1 != nullptr && cpart2 != nullptr &&
cmpPartList(cpart1->getParts(), cpart2->getParts()))
                    curOkay = true;
            }
            id2++;
        }
        id1++;
        // if we fail to find one item, we set AllOk to false and leave the loop.
One item difference = unequal.
        allOk = curOkay;
    }

    return allOk;
}

```

```

bool PartsLists::CompositePart::equals(Part* const o) const {
    // CompositeParts are equal, if they have the same name and the same subitems in
    the partslist.
    PartsLists::CompositePart* cpart = dynamic_cast<PartsLists::CompositePart*>(o);
    if(cpart != nullptr)
    {
        return strcmp(getName(), o->getName())==0 && cmpPartList(getParts(),cpart-
>getParts());
    }
    else
    {
        return false;
    }
}

void PartsLists::CompositePart::store(string const& fname) const
{
    // To store a compositePart, store the baseItem and iterate recursively through all
    items.
    ofstream out(fname);
    out << "0,1," << getName() << std::endl;
    storeWorker(out, 1, getParts());
    out.close();
}

void PartsLists::CompositePart::storeWorker(ofstream& out, size_t const level,
vector<PartsLists::Part*> const& list) const
{
    for (int i(0); i < list.size(); i++)
    {
        if (list[i] != nullptr)
        {
            // For each item, store the sublevel (how deep it is into the list),
            the type (part or compositepart/ 0 or 1) and it's name.
            PartsLists::CompositePart* cpart =
dynamic_cast<PartsLists::CompositePart*>(list[i]);
            out << level << "," << std::to_string(cpart != nullptr) << "," <<
list[i]->getName() << std::endl;
            if (cpart != nullptr)
            {
                storeWorker(out, level + 1, cpart->getParts());
            }
        }
    }
}

void PartsLists::CompositePart::load(string const& fname)
{
    // To load it back, delegate to recursive worker.
    ifstream in(fname);
    string currentline("");
    std::getline(in, currentline);
    loadWorker(in, 0, currentline, this);
    in.close();
}

void PartsLists::CompositePart::loadWorker(ifstream& in, size_t const level, string&
currentline, Part* p)
{
    while(!currentline.empty())
    {

```

```

// First, split the string from the file into its elements.
vector<string> elements = split_string(currentline, ',');
if (elements.size() == 3)
{
    // If the level fits, save the content.
    if (level == std::stoi(elements[0]))
    {
        if(level == 0)
        {
            this->setName(elements[2]);
            if(std::getline(in,currentline))
            {
                loadWorker(in, level + 1, currentline, p);
            }
        }
        else
        {
            // If it isnt root part, create a new part and save it.
            PartsLists::CompositePart* cpart =
dynamic_cast<PartsLists::CompositePart*>(p);
            if(cpart != nullptr)
            {
                // Check if its a part or compositepart and
depending on that, save it and recurse.
                if(elements[1][0]=='0')
                {
                    Part* newPart = new Part(elements[2]);
                    cpart->addPart(newPart);
                    if (std::getline(in, currentline))
                    {
                        loadWorker(in, level, currentline,
p);
                    }
                }
                else
                {
                    CompositePart* newPart = new
CompositePart(elements[2]);
                    cpart->addPart(newPart);
                    if (std::getline(in, currentline))
                    {
                        loadWorker(in, level+1,
currentline, newPart);
                    }
                }
            }
        }
    }
    // If the level of the function and the line is unequal, wind out of
recursion.
    else
    {
        return;
    }
}
}
}

```

Formatter.cpp

```
#include "formatter.h"
#include "part.h"

void formatting::SetFormatter::buildSet(PartsLists::Part* const part,
std::map<string, size_t>& partsSet) const
{
    if (part != nullptr)
    {
        PartsLists::CompositePart* cpart =
dynamic_cast<PartsLists::CompositePart*>(part);
        if (cpart != nullptr)
        {
            PartsLists::Part::partList subparts(cpart->getParts());

            // Iterate through all parts in the list
            for (int i(0); i < subparts.size(); i++)
            {
                if (subparts[i] != nullptr)
                {
                    bool isCpart =
dynamic_cast<PartsLists::CompositePart*>(subparts[i]) != nullptr;
                    if (isCpart)
                    {
                        // recurse
                        buildSet(subparts[i], partsSet);
                    }
                    else
                    {
                        string name = subparts[i]->getName();
                        partsSet[name] ? partsSet[name]++ :
partsSet[name] = 1;
                    }
                }
            }
        }
        else
        {
            std::cout << "Invalid part specified.\n";
        }
    }
}

void formatting::SetFormatter::printParts(PartsLists::Part* const p) const
{
    if (p != nullptr)
    {
        // First build a map/set
        std::map<string, size_t> partsSet;
        buildSet(p, partsSet);
        // Then iterate through it and print each item with its amount.
        std::cout << p->getName() << ":" << std::endl;
        for (auto it(partsSet.begin()); it != partsSet.end(); ++it)
        {
            std::cout << "\t" << it->second << " " << it->first << std::endl;
        }
        std::cout << std::endl;
        partsSet.clear();
    }
}
```



```

    }
    else
    {
        std::cout << "Invalid part specified.\n";
    }
}

void formatting::HierarchyFormatter::printParts(PartsLists::Part* const p) const
{
    if(p != nullptr)
    {
        // To create a hierarchy, recursively iterate through the partslist and
        instantly output each part.
        std::cout << p->getName() << std::endl;
        printPartsWorker(p, 1);
        std::cout << std::endl;
    }
    else
    {
        std::cout << "Invalid part specified.\n";
    }
}

void formatting::HierarchyFormatter::printPartsWorker(PartsLists::Part* const p, size_t
const indent) const
{
    if (p != nullptr)
    {
        // Try to cast to a compositepart
        PartsLists::CompositePart* cpart =
dynamic_cast<PartsLists::CompositePart*>(p);
        if (cpart != nullptr)
        {
            //
            PartsLists::Part::partList subparts = cpart->getParts();
            for (int i(0); i < subparts.size(); i++)
            {
                for (size_t in(0); in < indent; in++)
                {
                    std::cout << "\t";
                }
                std::cout << subparts[i]->getName() << std::endl;
                printPartsWorker(subparts[i], indent + 1);
            }
        }
        else
        {
            std::cout << "Invalid part specified.\n";
        }
    }
}

```

Main.cpp

```
#include "part.h"
#include "formatter.h"
using PartsLists::Part;
using PartsLists::CompositePart;

void test_reference(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    setf.printParts(&sitzgarnitur);
    hierf.printParts(&sitzgarnitur);
}

void test_alter(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
```

```

    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    Part polster("Polster");
    CompositePart polsterSessel("Sessel");
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&beink);
    polsterSessel.addPart(&sitzf);
    polsterSessel.addPart(&polster);

    sitzgarnitur.getParts()[0] = &polsterSessel;
    CompositePart* sub = dynamic_cast<CompositePart*>(sitzgarnitur.getParts()[0]);
    sub->getParts()[0] = &tisch;

    setf.printParts(&sitzgarnitur);
    hierf.printParts(&sitzgarnitur);
}

void test_empty(formatting::SetFormatter const& setf, formatting::HierarchyFormatter
hierf)
{
    CompositePart test("null_part");
    test.addPart(nullptr);

    setf.printParts(&test);
    hierf.printParts(&test);
}

void test_equals(formatting::HierarchyFormatter hierf)
{
    Part p1("Sessel");
    Part p2("Sessel");
    Part p3("Polster");
    if(p1.equals(&p2))
    {
        std::cout << "Parts are equal.\n";
    }
    else
    {
        std::cout << "Parts are unequal.\n";
    }
    std::cout << "Comparison:\n";
    hierf.printParts(&p1);
    hierf.printParts(&p2);

    CompositePart sesselpolster1("Sesselpolster");
    sesselpolster1.addPart(&p2);
    sesselpolster1.addPart(&p3);

    Part p4("Stuhl");
    CompositePart sesselpolster2("Sesselpolster");
    sesselpolster2.addPart(&p4);
    sesselpolster2.addPart(&p3);

```

```

        if(&sesselpolster1.equals(&sesselpolster2))
        {
            std::cout << "Compositeparts are equal.\n";
        }
        else
        {
            std::cout << "Compositeparts are unequal.\n";
        }
        std::cout << "Comparison:\n";
        hierf.printParts(&sesselpolster1);
        hierf.printParts(&sesselpolster2);

        std::cout << "Swapping 'Stuhl' for 'Sessel':\n";
        sesselpolster2.getParts()[0] = &p2;

        hierf.printParts(&sesselpolster1);
        hierf.printParts(&sesselpolster2);

        if (&sesselpolster1.equals(&sesselpolster2))
        {
            std::cout << "Now they are equal!\n";
        }
    }

void test_store(formatting::HierarchyFormatter hierf)
{
    Part beink("Bein (klein)");
    Part sitzf("Sitzflaeche");

    CompositePart sessel("Sessel");
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&beink);
    sessel.addPart(&sitzf);

    Part being("Bein (gross)");
    Part tischf("Tischflaeche");

    CompositePart tisch("Tisch");
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&being);
    tisch.addPart(&tischf);

    CompositePart sitzgarnitur("Sitzgarnitur");
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&sessel);
    sitzgarnitur.addPart(&tisch);

    sitzgarnitur.store("sitzgarnitur.csv");

    hierf.printParts(&sitzgarnitur);
}

void test_load(formatting::HierarchyFormatter hierf)
{
    CompositePart copy("Platzhalter");

```

```
        copy.load("sitzgarnitur.csv");

        hierf.printParts(&copy);
    }

void main()
{
    formatting::SetFormatter setf;
    formatting::HierarchyFormatter hierf;

    test_reference(setf, hierf);
    //test_alter(setf, hierf);
    //test_empty(setf, hierf);
    //test_equals(setf);
    //test_store(hierf);
    //test_load(hierf);
}
```

Testfälle

- Referenzbeispiel
- PartsList verändern
- Leerer Part/Nullpointer.
- Vergleich/Equals
- Store
- Load

Alle Testfälle sind zusätzlich als Funktionen in main.cpp enthalten.

Referenzbeispiel:

Sitzgarnitur:

- 4 Bein (gross)
- 8 Bein (klein)
- 2 Sitzflaeche
- 1 Tischflaeche

Sitzgarnitur

Sessel

- Bein (klein)
- Bein (klein)
- Bein (klein)
- Bein (klein)
- Sitzflaeche

Sessel

- Bein (klein)
- Bein (klein)
- Bein (klein)
- Bein (klein)
- Sitzflaeche

Tisch

- Bein (gross)
- Bein (gross)
- Bein (gross)
- Bein (gross)
- Tischflaeche

PartsList verändern:**Sitzgarnitur:**

8 Bein (gross)
7 Bein (klein)
1 Polster
2 Sitzflaeche
2 Tischflaeche

Sitzgarnitur**Sessel****Tisch**

Bein (gross)
Bein (gross)
Bein (gross)
Bein (gross)
Tischflaeche

Bein (klein)
Bein (klein)
Bein (klein)
Sitzflaeche
Polster

Sessel

Bein (klein)
Bein (klein)
Bein (klein)
Bein (klein)
Sitzflaeche

Tisch

Bein (gross)
Bein (gross)
Bein (gross)
Bein (gross)
Tischflaeche

Leeres Element (Nullpointer):

```
null_part:
```

```
null_part
```

Vergleich/Equals:

```
Parts are equal.
```

```
Comparison:
```

```
Sessel
```

```
Sessel
```

```
Compositeparts are unequal.
```

```
Comparison:
```

```
Sesselpolster
```

```
    Sessel
```

```
    Polster
```

```
Sesselpolster
```

```
    Stuhl
```

```
    Polster
```

```
Swapping 'Stuhl' for 'Sessel':
```

```
Sesselpolster
```

```
    Sessel
```

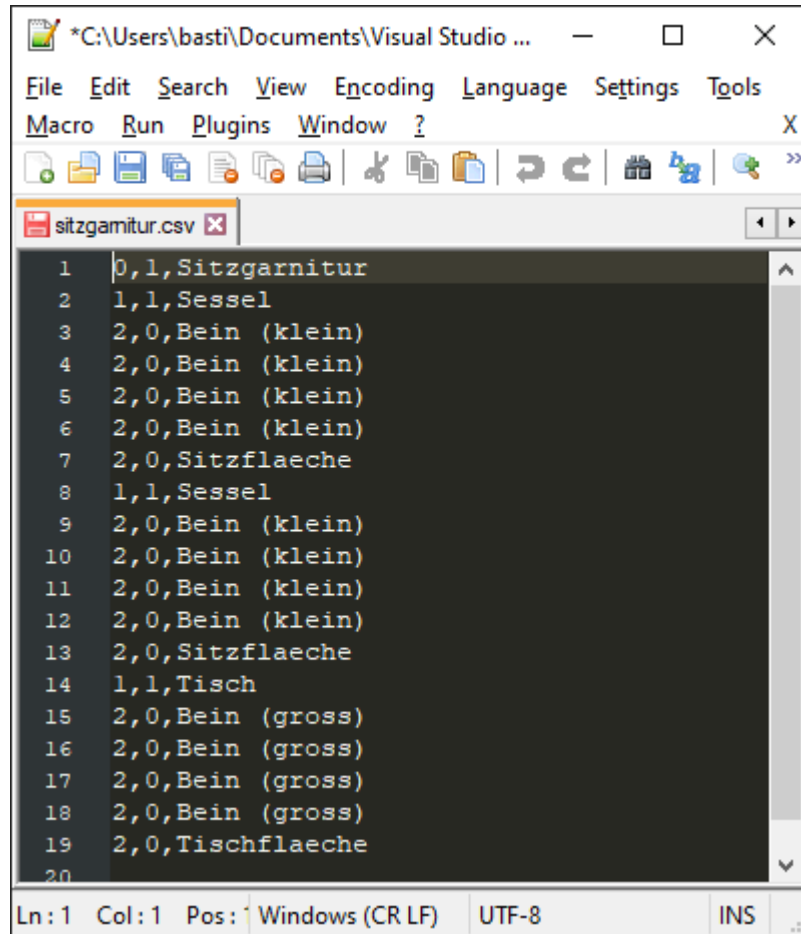
```
    Polster
```

```
Sesselpolster
```

```
    Sessel
```

```
    Polster
```

```
Now they are equal!
```


Store:


```

1 0,1,Sitzgarnitur
2 1,1,Sessel
3 2,0,Bein (klein)
4 2,0,Bein (klein)
5 2,0,Bein (klein)
6 2,0,Bein (klein)
7 2,0,Sitzflaeche
8 1,1,Sessel
9 2,0,Bein (klein)
10 2,0,Bein (klein)
11 2,0,Bein (klein)
12 2,0,Bein (klein)
13 2,0,Sitzflaeche
14 1,1,Tisch
15 2,0,Bein (gross)
16 2,0,Bein (gross)
17 2,0,Bein (gross)
18 2,0,Bein (gross)
19 2,0,Tischflaeche
20

```

Ln: 1 Col: 1 Pos: Windows (CR LF) UTF-8 INS

Load:

Sitzgarnitur	
Sessel	
Bein (klein)	
Bein (klein)	
Bein (klein)	
Bein (klein)	
Sitzflaeche	
Sessel	
Bein (klein)	
Bein (klein)	
Bein (klein)	
Bein (klein)	
Sitzflaeche	
Tisch	
Bein (gross)	
Bein (gross)	
Bein (gross)	
Bein (gross)	
Tischflaeche	