SWE3 Übung 05 Aufwand: ca. 6h

Bsp. 1: Flugreisen

Lösungsidee:

Die drei Klassen (Person, Flug, Flugreise) stehen in keiner Vererbungsbeziehung. Flugreise steht jedoch quasi über Person und Flug, da diese Klassen in den Komponenten verwendet werden.

Flugreise enthält einen Vektor aus Flügen und einen Vektor mit allen Passagieren.

Zum Speichern der Abflugs- und Ankunftszeit, wurde eine eigene Klasse erstellt, welche den output operator << überladen hat, um das Formatieren der Ausgabe leichter zu gestalten.

Außerdem wird ein enum für das Geschlecht der Personen verwendet.

Der Operator << wird von allen drei Klassen überladen.

Da die Klassen alle recht klein und unkompliziert sind, habe ich diese inline im header file implementiert.

Quellcode:

Flugreisen.h

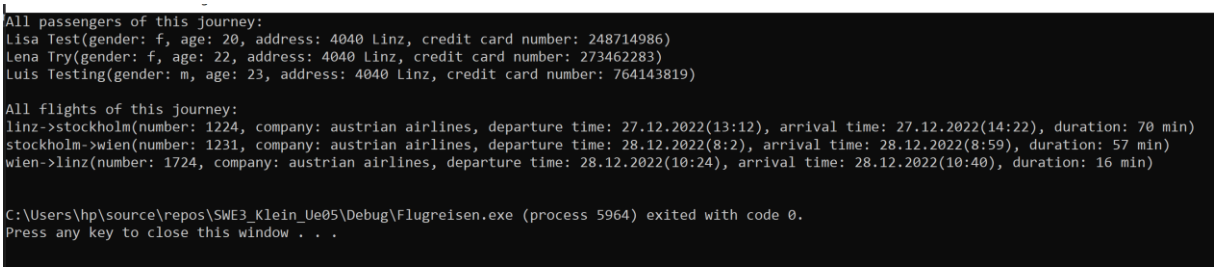
```
#if ! defined FLUGREISEN_H
#define FLUGREISEN_H

#include <iostream>
#include <string>
#include <vector>

// gender_t is used in person
enum gender_t {
	female,
	male,
	diverse
};

class person {
private:
	std::string first_name;
	std::string last_name;
	gender_t gender;
	int age;
	std::string address;
	std::string credit_card;

	friend inline std::ostream& operator<<(std::ostream& lhs, person const& rhs) {

		// formatting the gender to one letter
		std::string gender_string;

		switch (rhs.gender) {
		case female:
			gender_string = "f";
			break;
		case male:
			gender_string = "m";
			break;
```

```
                case diverse:
                        gender_string = "d";
                        break;
                }

                lhs << rhs.first_name << " " << rhs.last_name << "(gender: " <<
gender_string << ", age: " << rhs.age << ", address: "
                        << rhs.address << ", credit card number: " << rhs.credit_card <<
")";

                return lhs;
        }

public:
        person(std::string const& fn, std::string const& ln, gender_t const& g, int
const a, std::string const& ad, std::string const& cr) {
                first_name = fn;
                last_name = ln;
                gender = g;
                age = a;
                address = ad;
                credit_card = cr;
        }
};



class date_t {
public:
        int year;
        int month;
        int day;
        int hour;
        int min;

        // default constructor is required for flight class
        date_t() : year(0), month(0), day(0), hour(0), min(0) {

        }

        date_t(int y, int m, int d, int h, int mi) : year(y), month(m), day(d), hour(h),
min(mi) {

        }
};

std::ostream& operator<<(std::ostream& lhs, date_t const& rhs) {
        lhs << rhs.day << "." << rhs.month << "." << rhs.year << "(" << rhs.hour << ":"
<< rhs.min << ")";
        return lhs;
}


class flight {
private:
        int flight_number;
        std::string company;
        std::string departure;
        std::string destination;
        date_t departure_time;
        date_t arrival_time;
        int flight_time;

        friend inline std::ostream& operator<<(std::ostream& lhs, flight const& rhs) {
```

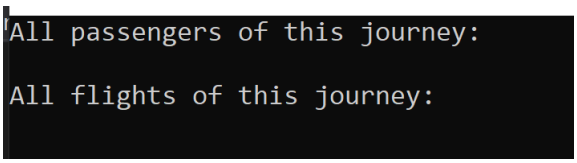
```
		// formatting output for flight
		lhs << rhs.departure << "->" << rhs.destination << "(number: " <<
rhs.flight_number << ", company: " << rhs.company
			<< ", departure time: " << rhs.departure_time << ", arrival time:
" << rhs.arrival_time << ", duration: "
			<< rhs.flight_time << " min)";
		return lhs;
	}

public:
	flight(int const num, std::string const& co, std::string const& dep, std::string
const& des, date_t const& dep_t, date_t const& arr_t, int const f_t) {
		flight_number = num;
		company = co;
		departure = dep;
		destination = des;
		departure_time = dep_t;
		arrival_time = arr_t;
		flight_time = f_t;
	}
};


class air_travel {
private:
	std::vector<flight> flights;
	std::vector<person> passengers;

	friend inline std::ostream& operator<<(std::ostream& lhs, air_travel const& rhs)
{
		// formatting output for air travel
		lhs << "All passengers of this journey: \n";
		for (size_t i{ 0 }; i < rhs.passengers.size(); ++i) {
			lhs << rhs.passengers[i] << "\n";
		}
		lhs << "\n";
		lhs << "All flights of this journey: \n";
		for (size_t i{ 0 }; i < rhs.flights.size(); ++i) {
			lhs << rhs.flights[i] << "\n";
		}
		lhs << "\n";
		return lhs;
	}
public:
	air_travel(std::vector<flight> const& fl, std::vector<person> const& p) {
		flights = fl;
		passengers = p;
	}
};



#endif
```

Testfälle:

Test der gesamten Funktion:



Flugreise ohne Flüge und Passagiere:



Bsp. 2: Stücklistenverwaltung

Lösungsidee:

Die Basisklasse ist Part. Composite Part ist eine Spezialisierung davon und übernimmt alle Methoden/Komponenten. Composite Part enthält zusätzlich noch einen Vektor aus Parts, aus denen es besteht und entsprechende Methoden um den Vektor aufzurufen und zu füllen.

Vorsicht: Theoretisch wäre es möglich bei addPart das Teil selbst hinzuzufügen -> führt zu Loop

Zusätzlich steht über der Basisklasse Part, Storeable (Interface). Dieses Interface gibt die Methoden store und load vor -> diese Methoden sind in Part zu implementieren.

Store speichert alle verwendeten Parts eines Composite Parts in einem Vektor, Load schreibt diesen Vektor auf die Konsole

Formatter:

Formatter ist eine abstrakte Klasse mit Spezialisierung SetFormatter und HierarchyFormatter.

Funktionsweise SetFormatter:

> Durch die Composite Parts wird rekursiv iteriert. Jedes Part, das kein Composite Part ist, wird in einer Map gezählt (beim ersten Vorkommen wird der Eintrag gesetzt und mit 1 initialisiert, danach wird bei jedem Vorkommen um 1 inkrementiert.)

Funktionsweise HierarchyFormatter:

> Es wird rekursiv durch alle Parts iteriert, bei jeder Rekursionstiefe wird weiter eingerückt.

Quellcode:

Part.h

```
#if ! defined PARTS_H
#define PARTS_H

#include <string>
#include <vector>
#include <iostream>

namespace PartsLists {

	// interface
	class Storable {
	public:
		virtual void store() = 0;
		virtual void load() = 0;

	protected:
		std::vector<std::string> part_list;
	};

	// base class
	class Part: public Storable {
	public:
		Part(std::string const& name): name(name){}

		virtual ~Part(){}

		std::string getName() {
			return name;
		}

		bool equals(Part const& o) {
			return name == o.name;
		}

		// implemented in the cpp
		void store();
		void load();

	protected:
		std::string name;

	private:
		// implemented in the cpp
		void store_rec(PartsLists::Part* p);
	};

	// specialized class
	class CompositePart :public Part {
	public:
		CompositePart(std::string const& name): Part(name){
		}
		~CompositePart() {
		}
		void addPart(Part* p) {
			parts.push_back(p);
		}

		std::vector<Part*> getParts()const {
			return parts;
		}
```

```
	private:
		std::vector<Part*> parts;
	};
}


#endif
```

Part.cpp

```
#include "part.h"

void PartsLists::Part::store() {
	// calling recursive function
	store_rec(this);
}

void PartsLists::Part::store_rec(PartsLists::Part* p) {
	// saving the parts in a vector
	part_list.push_back(p->getName());

	PartsLists::CompositePart* cp = dynamic_cast<PartsLists::CompositePart*>(p);

	// recursion to get to all parts that are needed for the composite parts
	if (cp != nullptr) {
		std::vector< PartsLists::Part*> parts = cp->getParts();
		for (size_t i{ 0 }; i < parts.size(); ++i) {
			store_rec(parts[i]);
		}
	}
}

void PartsLists::Part::load() {
	// output of the list, that was stored before
	for (size_t i{ 0 }; i < part_list.size(); ++i) {
		std::cout << part_list[i] << "\n";
	}
}
```

Formatter.h

```
#if ! defined FORMATTER_H
#define FORMATTER_H

#include "part.h"
#include <iostream>
#include <map>

// abstract class
class Formatter {
public:
	// implemented in the specialized classes
	virtual void printParts(PartsLists::Part& p) const = 0;
	virtual ~Formatter(){}
};



class SetFormatter : public Formatter {
```

```
public:
	void printParts(PartsLists::Part& p) const override;

	// default constructor -> calls constructor of abstract class
	SetFormatter() :Formatter() {}
private:
	void count_parts_rec(std::map<std::string, int>& parts_count, PartsLists::Part*
p)const;
};



class HierarchyFormatter : public Formatter {
public:
	void printParts(PartsLists::Part& p) const override;

	// default constructor -> calls constructor of abstract class
	HierarchyFormatter() :Formatter() {}
private:
	void print_parts_rec(PartsLists::Part* p, int const indent)const;
};



#endif
```

Formatter.cpp

```
#include "formatter.h"

void SetFormatter::printParts(PartsLists::Part& p) const {
	std::map<std::string, int> parts_count;

	// counts the parts and saves it in the map
	count_parts_rec(parts_count, &p);

	std::cout << p.getName() << ": \n";

	// iterating through the map to print all the parts with their corresponding
count
	for (auto it = parts_count.begin(); it != parts_count.end(); it++) {
		std::cout<<"    " << it->second << " " << it->first << "\n";
	}
}

void SetFormatter::count_parts_rec(std::map<std::string, int>& parts_count,
PartsLists::Part* p) const {

	PartsLists::CompositePart* cp = dynamic_cast<PartsLists::CompositePart*>(p);

	// going in recursion for composite parts
	if (cp != nullptr) {
		std::vector< PartsLists::Part*> parts = cp->getParts();
		for (size_t i{ 0 }; i < parts.size(); ++i) {

			count_parts_rec(parts_count, parts[i]);
		}
	}
	// only counting the elementary parts -> parts that aren't composite parts
	else {
		if (parts_count.count(p->getName())) {
			parts_count[p->getName()]++;
```

```
			}
			else {
				parts_count[p->getName()] = 1;
			}
		}

}

void HierarchyFormatter::printParts(PartsLists::Part& p) const {
		// calling recursive function
		print_parts_rec(&p, 0);
}

void HierarchyFormatter::print_parts_rec(PartsLists::Part* p, int const indent)const {
		// printing the indent for a nice formatted output
		for (int i{ 0 }; i < indent; ++i) {
			std::cout << "    ";
		}

		// printing the name of the current part
		std::cout << p->getName()<<"\n";


		PartsLists::CompositePart* cp = dynamic_cast<PartsLists::CompositePart*>(p);
		// going in recursion for composite parts
		if (cp != nullptr) {
			std::vector< PartsLists::Part*> parts = cp->getParts();
			for (size_t i{ 0 }; i < parts.size(); ++i) {
				print_parts_rec(parts[i], indent + 1);
			}
		}

}
```

Testfälle:

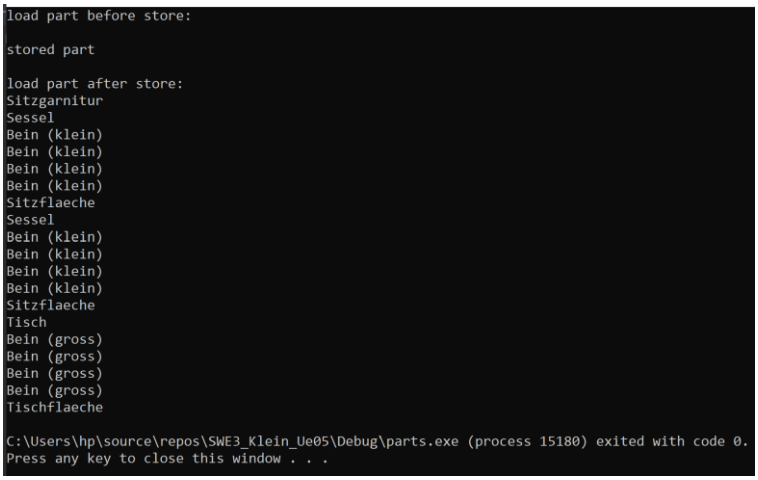
Mit Beispieldaten:

SetFormatter:

```
Sitzgarnitur:
    4 Bein (gross)
    8 Bein (klein)
    2 Sitzflaeche
    1 Tischflaeche

C:\Users\hp\source\repos\SWE3_Klein_Ue05\Debug\parts.exe (process 5040) exited with code 0.
Press any key to close this window . . .
```

HierarchyFormatter:

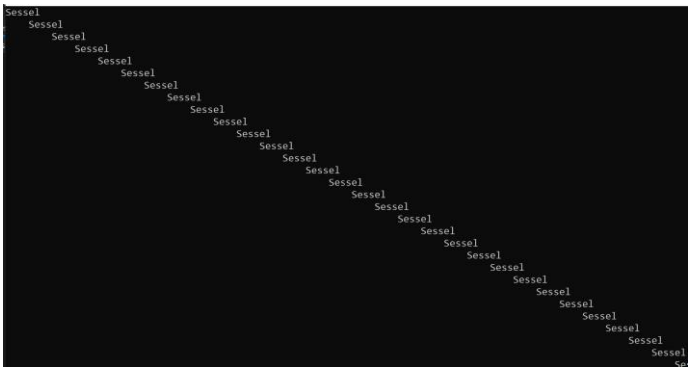
Load und Store:



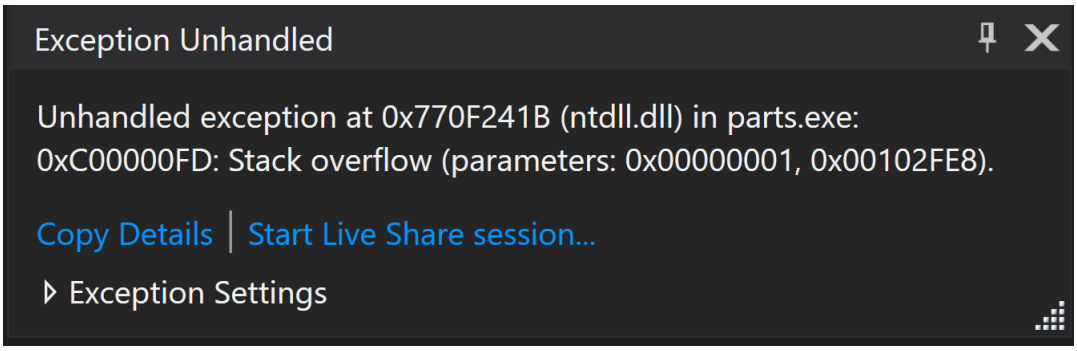
Man muss bei einem Teil die Setliste erst mit store speichern, bevor man sie laden kann.

Risiko:

Es ist möglich einen Loop zu bilden (wie in Lösungsidee erwähnt):



Mit Hierarchy ^



Mit set ^ Stackoverflow