Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 3. Tag

Aufgabe 1 Vererbung und Polymorphie

a) **Klasse *Person*** Schreibe eine Klasse *Person* (Diese Klasse hat nichts mit unserem Aufzugsimulator zu tun). *Person* soll ein *protected* Attribut *name* vom Typ *std::string* haben (**#include <string>**), welches den Namen speichert. Initialisiere den Namen im Konstruktor von *Person* und schreibe auch einen Destruktor. Implementiere außerdem die folgende Methodem, um Informationen über die Person, in unserem Fall den Namen, abzurufen:

```
std::string getInfo() const;
```

Hinweis: Um ein String-Literal an eine *std::string* Variable anzuhängen, musst du aus dem String-Literal zuerst ein *std::string*-Objekt machen. Beispiel:

```
std::string text = std::string("Name:␣" ) + name;
```

b) **Klasse *Student*** Schreibe eine Klasse *Student*, die von *Person* erbt und eine Person mit einer Matrikelnummer (ebenfalls *std::string*) modelliert. Rufe in der Initialisierungsliste den entsprechenden Konstruktor der Elternklasse *Person* mittels **Person(name)** auf. Implementiere auch einen Destruktor.

Überschreibe die Methode *getInfo()*, sodass zusätzlich zum Namen auch die Matrikelnummer zurückgegeben wird. Du kannst bei Bedarf die *getInfo()*-Implementierung der Elternklasse *Person* von *Student* aus mittels **Person::getInfo()** aufrufen.

Tipp: In C++ gibt es leider keine Möglichkeit zu deklarieren, dass eine Methode eine andere überschreibt (vergleichbar mit der Annotation *@Override* in Java).

Trotzdem zeigt dir Eclipse mit einem kleinen aufwärts zeigenden Dreieck links neben den Zeilennummern an, ob du eine Methode überschreibst oder nicht (siehe Abbildung 1)

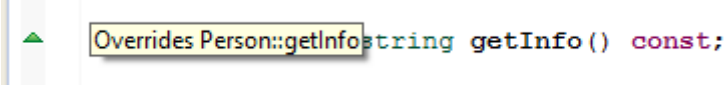

Abbildung 1: Hinweis in Eclipse, dass eine Methode überschrieben wird. Mit einem Klick auf das Dreieck navigiert man zur entsprechenden Methode in der Elternklasse.

c) Erstelle nun in *main()* je eine Person und einen Studenten und gib deren Daten auf der Konsole aus. Vergewissere dich, dass bei *Student* auch die Matrikelnummer ausgegeben wird. Schau dir auch die Ausgaben der Konstruktoren und Destruktoren an, und versuche, diese nachzuvollziehen.

d) Implementiere nun die Funktion

```
/** Prints person information on console */
void printPersonInfo(const Person *person);
```

Dadurch dass *person* als const Zeiger übergeben wird, können auch Unterklassen von *Person*, wie z.B. *Student*, übergeben werden.

Teste deine Implementierung. Rufe dazu *printPersonInfo()* sowohl mit beiden Personentypen auf.

e) **Dynamic Dispatch bei *printPersonInfo*** Du merkst, dass *printPersonInfo()* unabhängig von übergebenem Personentyp immer nur den Namen der Person ausgibt, aber nicht die Matrikelnummer. Der Grund dafür ist, dass *getInfo()* nicht als **virtual** deklariert wurde und deshalb auch kein dynamischer Dispatch der Methode stattfindet. Deklariere daher *getInfo()* als **virtual**.

Teste deine Implementierung erneut und vergewissere dich, dass nun immer die richtige Methode aufgerufen wird.

f) **Virtueller Destruktor** Lege einen Studenten dynamisch auf dem Heap an und speichere die Adresse in einem Zeiger auf eine *Person*. Lösche die Person anschließend.

```
Person *pTim = new Student("Tim", "321654");
delete pTim;
```

Analysiere die Konsolenausgabe. Es wird nur der Destruktor von *Person* aufgerufen, obwohl es sich um ein Objekt vom Typ *Student* handelt. Auch hier liegt es daran, dass kein dynamischer Dispatch bei der Zerstörung erfolgt. Deklariere deshalb in beiden Klassen den Destruktor als **virtual** und teste die Korrektheit der Destruktoraufrufe.

Aufgabe 2 Pure Virtual

In dieser Aufgabe wollen wir Vererbung und Polymorphie dazu nutzen, um mathematische Ausdrücke als Bäume von Primitivoperationen zu modellieren. Dazu werden wir eine abstrakte Oberklasse *Expression* mit der abstrakten Methode *compute()* erstellen. Einzelne Knotentypen wie Addition und Subtraktion werden von *Expression* abgeleitet und implementieren *compute()*, um die jeweilige Operation zu realisieren.

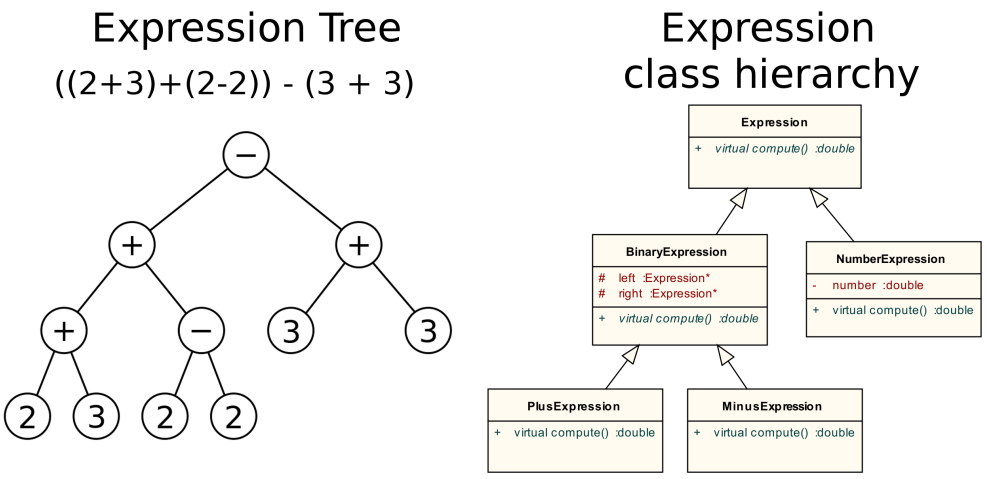

Abbildung 2: Abbildung: Beispielausdruck mit Ausdrucksbaum und Klassenhierarchie

a) **Klasse *Expression*** Schreibe die abstrakte Klasse *Expression*. Diese soll als Basisklasse für alle Ausdrücke dienen. Implementiere einen parameterlosen Konstruktor und einen virtuellen Destruktor, die je eine Meldung auf der Konsole ausgeben, sodass es bei der Ausführung ersichtlich wird, wann eine *Expression* erzeugt und wann zerstört wird. Deklariere außerdem eine abstrakte (*pure virtual*) Methode *virtual double compute() = 0;*, die das Ergebnis des Ausdrucks berechnen und zurückgeben soll.

b) **Klasse *NumberExpression*** Schreibe die Klasse *NumberExpression*, die ein (Baum-)Blatt mit einer Zahl darstellt. Dementsprechend soll *NumberExpression* von *Expression* erben und ein Attribut zum Speichern einer Zahl besitzen, das im Konstruktor initialisiert wird. Implementiere den Konstruktor und virtuellen Destruktor und versehe auch diese mit einer Konsolenausgabe. Die Methode *compute()* gibt die gespeicherte Zahl zurück.

c) **Klasse *BinaryExpression*** Schreibe die abstrakte Klasse *BinaryExpression* mit den *protected* Attributen *Expression *left, *right*. Implementiere den Konstruktor und virtuellen Destruktor mit entsprechender Ausgabe. Vergiss nicht, im Destruktor die beiden Zweige zu löschen.

d) **Klassen *Plus-* und *Minusexpression*** Schreibe die Klassen *PlusExpression* und *MinusExpression*, die von *BinaryExpression* erben und eine Addition bzw. Subtraktion realisieren. Implementiere die Kon- und Destruktoren sowie die *compute()* Methode.

e) **Test** Teste deine Implementierung. Ein gutes Beispiel findest du in Abbildung weiter oben. Schaue dir die Ausgabe genau an und versuche anhand der gegebenen Klassenhierarchie die Reihenfolge der Erzeugung und Zerstörung von Objekten nachzuvollziehen.

Aufgabe 3 Fortsetzung Aufzugsimulator

Unser bisheriger Aufzugsimulator hat eine feste Strategie, nach der die einzelnen Stockwerke abgefahren werden. Mithilfe von Polymorphie können wir den Simulator so erweitern, dass die Strategie austauschbar wird.

a) Lagere die bereits existierende Simulation des Aufzugs aus der *main*-Funktion in eine eigene Funktion *runSimulation()* aus. Die Funktion sollte das volle Gebäude als Parameter entgegennehmen und eine Liste (*std::list<int>*) der angefahrenen Stockwerke zurückgeben. Überlege dir, auf welche Art das Gebäude idealerweise übergeben werden sollte. Teste deine Implementierung.

b) **Klasse *ElevatorStrategy*** Implementiere die Klasse *ElevatorStrategy*. Diese soll die Basisklasse für verschiedene Aufzugstrategien sein.

```
/**
 * Elevator strategy base class. Determines to which floor the elevator should move next.
 */
class ElevatorStrategy {
public:
   virtual ~ElevatorStrategy();

   /**
    * Creates a plan for the simulation.
    * Default implementation does nothing but saving the building pointer.
    */
   virtual void createPlan(const Building*);

   /**
    * Gets the next floor to visit.
    */
   virtual int nextFloor() = 0;

protected:
   /** Pointer to current building, set by createPlan() */
   const Building *building;
};
```

Damit die Strategie das Gebäude nicht selbst modifizieren kann, wird *Building* per const Pointer übergeben.

c) **Eine einfache Aufzusstrategie** Implementiere eine einfache Aufzugstrategie. Diese soll folgendermaßen vorgehen: Falls der Aufzug momentan leer ist, soll zum tiefsten Stockwerk gefahren werden, wo sich noch Personen befinden. Falls der Aufzug nicht leer ist, wird das Zielstockwerk einer der Personen im Aufzug ausgewählt.

d) **Implementierung von *runSimulation*** Ändere nun *runSimulation()* entsprechend um, sodass die Simulation anhand der gegebenen Strategie durchgeführt wird. Folgender Pseudocode kann dir als Denkhilfe dienen:

while *People in Building or Elevator* **do**
  Calculate next floor;
  Move Elevator to next floor;
  Let all arrived people off;
  Let all people on floor into Elevator;
end

e) **Systemtest** Teste deine Implementierung mit der bisher erstellten, einfachen Strategie.

f) **Neue Aufzugstrategien (optional)** Entwickle eigene Aufzugstrategien. Versuche, verschiedene Größen zu optimieren, wie z.B. die Anzahl der Stopps oder die verbrauchte Energie. Hierfür könnte Backtracking verwenden[1], eine einfache Methode, um optimale Lösungen durch Ausprobieren zu finden. Beachte, dass der Aufzug auch kopiert werden kann, um verschiedene Strategien zu testen.

[1] Siehehttp://de.wikipedia.org/wiki/Backtracking