ILV Datenstrukturen und Algorithmen

02: Vererbung und Komposition, Exception handling und assert

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Kapitel

Vererbung und Komposition

Beziehungen unter Klassen

- OOP-Ansatz: Klassen modellieren Objekte der realen Welt oder der Anschauung.
- Klassen besitzen ...
  - Attribute, Eigenschaften: member variable
  - Methoden, Fähigkeiten: member function
- Klassen stehen untereinander in Beziehung:
  - Komposition: Ein Vehicle **hat ein** SteeringWheel, ein Vehicle **hat einen** Motor.
  - Spezialisierung: Ein Car **ist ein** Vehicle, ein Bus **ist ein** Vehicle.

Komposition: Motivation

```
1  class Circle
2  {
3      double x, y, radius;
4    public:
5      Circle(double x, double y, double radius);
6      void move(double x2, double y2) { x+=x2; y+=y2 };
7  };
8  class Square
9  {
10     double x, y, width;
11   public:
12     Square(double x, double y, double width);
13     void move(double x2, double y2) { x+=x2; y+=y2 };
14 };
```

- Codeduplizierung: fehleranfällig, Refactoring schwierig, schwerfälliger source code, schlechter Programmierstil!
- Idee: `Circle` und `Square` **hat ein** `center` vom Typ `Vector`. Ein `Vector` hat Koordinaten `x, y`.
- Ziel: Wiederverwenden von Code, der getestet wurde. Code nicht immer wieder neu implementieren.

Komposition: Anwendung

```
1  class Vector
2  {
3      double x, y;
4    public:
5      Vector(double x, double y);
6      void add(const Vector& p) { x+=p.x; y+=p.y; };
7  };
8  class Circle
9  {
10     Vector center; double radius;
11   public:
12     Circle(const Vector& center, double radius);
13     void move(const Vector& diff) { center.add(diff); };
14 };
15 class Square
16 {
17     Vector center; double width;
18   public:
19     Square(const Vector& center, double width);
20     void move(const Vector& diff) { center.add(diff); };
21 };
```

Komposition: Anwendung

```
1  class Circle
2  {
3      Vector center; double radius;
4    public:
5      Circle(const Vector& center, double radius);
6      void move(const Vector& diff) { center.add(diff); };
7  };
8  class Square
9  {
10     Vector center; double width;
11   public:
12     Square(const Vector& center, double width);
13     void move(const Vector& diff) { center.add(diff); };
14 };
```

- Taucht ein Fehler in der Implementierung von `Vector` auf, so muss dieser nicht in `Circle` und `Square` extra behoben werden.
- Ist die Klasse `Vector` einmal ordentlich implementiert und getestet, so kann sie an vielen Stellen im restlichen Code verwendet werden.

Komposition: Anwendung

```
1  class Circle
2  {
3      Vector center; double radius;
4    public:
5      Circle(const Vector& center, double radius);
6      void move(const Vector& diff) { center.add(diff); };
7  };
8  class Square
9  {
10     Vector center; double width;
11   public:
12     Square(const Vector& center, double width);
13     void move(const Vector& diff) { center.add(diff); };
14 };
```

- Motto 1: Taucht Code, der die gleiche Funktionalität umsetzt, mehrmals auf $\Rightarrow$ in eine Klasse (oder Funktion) kapseln!
- Motto 2: „Separation of concerns“, trenne Dinge, die nicht zusammen gehören. Die Klasse `Square` sollte sich nicht um die Implementierung von Vektor-Addition kümmern müssen!

Vererbung: Motivation

```
1  class Vehicle
2  {
3    private:
4      Motor motor;
5    public:
6      void startMotor() { motor.start(); };
7  };
8  class Car
9  {
10   private:
11     Motor motor;
12   public:
13     void startMotor() { motor.start(); };
14 };
```

- Nochmal: Codeduplizierung ist schlecht!
- Idee: `Vehicle` hat schon Methode `startMotor` und `Car` **ist ein** `Vehicle`!
- Wie kann man dem Compiler sagen, dass `Car` eh ein `Vehicle` ist?

Vererbung: Syntax & Semantik

```
1  class Vehicle
2  {
3    private:
4      Motor motor;
5    public:
6      Vehicle();
7      void startMotor() { motor.start(); };
8  };
9  class Car : public Vehicle
10 {
11   public:
12     Car();
13 };
14
15 void test()
16 {
17   Car a;
18   /* Car "inherts" the method "startMotor" from Vehicle */
19   a.startMotor();
20 }
```

Vererbung: Syntax & Semantik

```
1  class Vehicle
2  {
3    private:
4      Motor motor;
5    public:
6      Vehicle();
7      void startMotor() { motor.start(); };
8  };
9  class Car : public Vehicle
10 {
11   public:
12     Car();
13 };
```

- Man sagt...
  - `Vehicle` ist eine „superclass (base class, parent class, Basisklasse)“ von `Car`,
  - `Car` ist die „sub class (derived class, Unterklasse)“ von `Vehicle`, `Car` wurde von `Vehicle` „abgeleitet“.
- Idee: Attribute und Methoden der Basisklasse sind in der Unterklasse verfügbar.

Vererbung: Zugriffsschutz

```
1  class A
2  {
3    private:
4      int x;
5    protected:
6      int y;
7    public:
8      int z;
9  };
10 class B : public A
11 {
12 };
```

- `A::x` ist nur für `A` sichtbar.
- `A::y` ist nur für `A` und dessen Unterklassen (und dessen Unterklassen und dessen ...) sichtbar. Also ist `A::y` von `B` aus zugreifbar.
- `A::z` ist überall, auch außerhalb von `A` und `B`, sichtbar.

Vererbung: Quadratur des Kreises

```
1  class Circle
2  {
3      Vector center; double radius;
4    public:
5      Circle(const Vector& center, double radius);
6      void move(const Vector& diff) { center.add(diff); };
7  };
8  class Square
9  {
10     Vector center; double width;
11   public:
12     Square(const Vector& center, double width);
13     void move(const Vector& diff) { center.add(diff); };
14 };
```

- Trotz Kapselung von `Vector` noch immer Codeduplizierung!
- Aber: `Circle` ist kein `Square` und `Square` ist kein `Circle`.
- Idee: Einführen einer gemeinsamen Basisklasse.

Vererbung: Quadratur des Kreises

```
1  class GeometryObject
2  {
3      Vector position;
4    public:
5      GeometryObject(const Vector& position);
6      void move(const Vector& diff) { position.add(diff); };
7  };
8  class Circle : public GeometryObject
9  {
10     double radius;
11   public:
12     Circle(const Vector& position, double radius);
13 };
14 class Square : public GeometryObject
15 {
16     double width;
17   public:
18     Square(const Vector& position, double width);
19 };
```

- Damit ist `move` auch für `Circle` und `Square` verfügbar.
- Motto: Manchmal eine gemeinsame Basisklasse einführen!

Lebenszyklus eines Objektes

Erinnerung:

- Konstruktor: wird beim Erzeugen eines Objektes aufgerufen.
- Destruktor: wird beim „Sterben“ eines Objektes aufgerufen.

Wie ist das bei `Square`?

- Wer initialisiert die Attribute die von `GeometryObject` geerbt wurden?
- Wer räumt ggf. Speicher auf, auf den durch einen Pointer in `GeometryObject` verwiesen wurde?

Auf **keinen** Fall sämtliche Unterklassen von GeometryObject (Codeduplizierung)!

Antwort: Man ruft den Konstruktor/Destruktor von `GeometryObject` auf.

Konstruktor der Unterklasse

```
1 class GeometryObject
2 {
3     Vector position;
4   public:
5     GeometryObject(const Vector& position);
6     void move(const Vector& diff) { position.add(diff); };
7 };
```

```
1 class Square : public GeometryObject
2 {
3     double width;
4   public:
5     Square(const Vector& position, double width);
6 };
```

```
1 Square::Square(const Vector& position, double width) :
2   /* First: call the base class constructor */
3   GeometryObject(position),
4   width(width)
5 {
6 }
```

Konstruktor/Destruktor Reihenfolgen

```
1  class A
2  {...};
3  class B : public A
4  {...};
5  class C : public B
6  {...};
7
8  void test()
9  {
10   /* Calls constructor of A,
11    *     then constr. of B,
12    *     then constr. of C */
13   C c;
14
15   /* Calls destructor of C,
16    *     then destr. of B,
17    *     then destr. of A */
18 }
```

Live-Demo...

Casting

- Casting bezeichnet die Umwandlung eines Typs in einen anderen:

```
1 double d = 3.1;
2 unsigned u = d;                       // Implicit type conversion
3 double* pd = &d;
4 unsigned* pu = (unsigned*) pd;        // Requires casting operator
```

- Ist B eine Unterklasse von A, so kann ein Objekt vom Typ B in ein Objekt vom Typ A gecastet werden.

Casting between classes

```
1  class A
2  {
3    public:
4      int x;
5      void print()  { cout << x << endl; };
6  };
7  class B : public A
8  {
9    public:
10     int y;
11 };
12 void func(A &obj) // Accepts objects of type B, too
13 {
14   obj.print();  //Prints 3
15   obj.x = 2;
16   obj.y = 1;    //Error: there is no A::y defined
17 }
18 int main()
19 {
20   B b;
21   b.x = 3;
22   func(b);
23   b.print();    //Prints 2
24 }
```

Casting

```
1  class Vehicle
2  {
3    public:
4      void removeMotor();
5      void removeWheels();
6  };
7  class Car : public Vehicle
8  {
9  };
10 class Bus : public Vehicle
11 {
12 };
13
14 void repair(Vehicle& v)
15 {
16   v.removeMotor();
17   v.removeWheels();
18 }
```

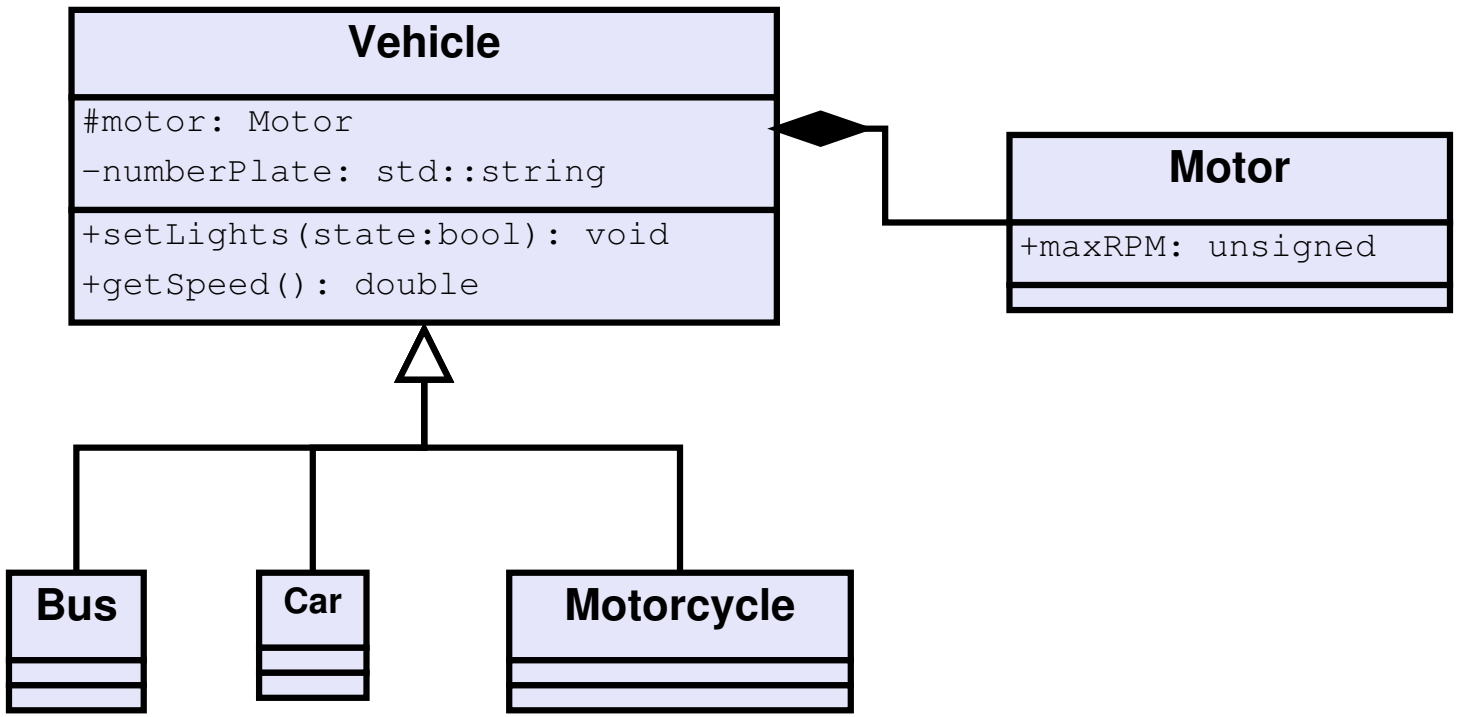
- Neuen Typ von Vehicle erfunden? — Neue Unterklasse erstellen, keine Änderung für `repair`!

Zugreifen auf Funktionen/Attribute der Basisklasse

```
1  class A
2  {
3    protected:
4      int x;
5    public:
6      A();
7      void func();
8  };
9  class B : public A
10 {
11   protected:
12     int x;
13   public:
14     B();
15     void func();
16 };
17 void B::func()
18 {
19   A::x = 2; // Accesses the member x of A
20   B::x = 3; // This is the same as: x = 3;
21   A::func();  // Calls the function of A
22   B::func();  // This is the same as: func();
23 }
```

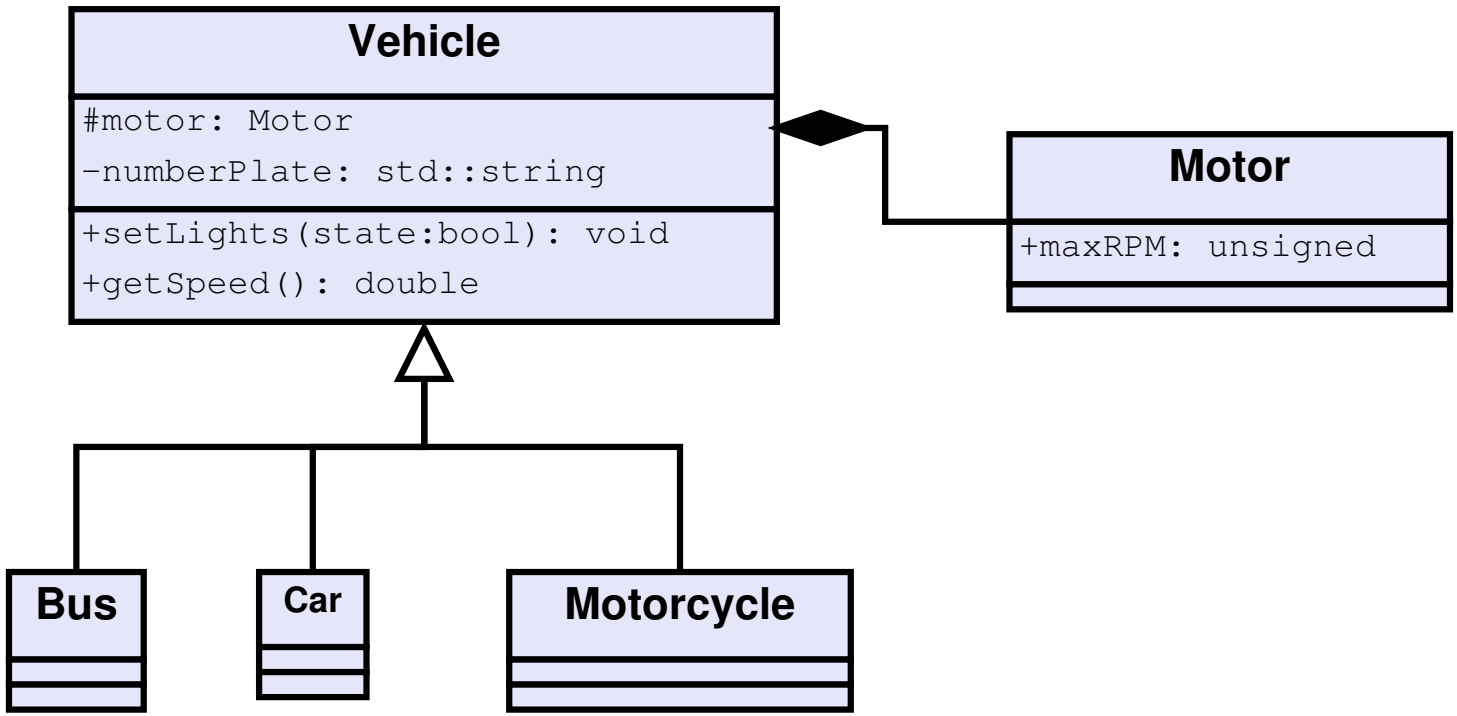
UML Diagramme — kurzer Einblick

- UML steht für Unified Modeling Language.
- UML definiert u.a. sogenannte Klassendiagramme.
- Durch Klassendiagramme lassen sich die Beziehungen zwischen Klassen visualisieren.



UML Diagramme — kurzer Einblick

- Eine Klassen besteht aus
  - Klassennamen, Liste der Attribute, Liste der Funktionen
- Arten von Beziehungen:
  - Vererbung
  - Aggregation, Komposition
- Zugriffsbeschränkung:
  - public (+), protected (#), private (-)



Kapitel

Exceptions

Exceptions: Motivation

- Exceptions dienen zur Fehlerbehandlung.
- Oft wird ein Fehler durch bestimmte Rückgabewerte einer Funktion signalisiert.
- Fehlercodes müssen von aufrufender Funktion weiter „nach oben“ geleitet werden.
- Duplizierter und langweiliger Code, der keine eigentliche Funktionalität umsetzt.
- Folgerungen:
  - Oft wird auf Fehlerbehandlung verzichtet. Schlechter Stil!
  - Programm wird beendet. Schlechter Stil!
  - Globale Variablen für Fehlercodes. Katastrophaler Stil!
- Idee: Man „fängt“ Fehler bewusst ab und behandelt diese an geeigneter Stelle.

Exceptions: Motivation

```
1  int readFile(string filename, string &content)
2  {
3    if( /*file does not exist*/ )
4      return -1;
5    if( /*no permissions to read file*/ )
6      return -2;
7    /*Read file byte by byte.*/
8  }
9  int readJPGFromFile(string filename, Image &image)
10 {
11   string content;
12   int ret = readFile(filename, content);
13   if( ret < 0 )
14     return ret;
15   /*Interpret the JPG file format.*/
16 }
17 int displayFile(string filename)
18 {
19   Image image;
20   int ret = readJPGFromFile(filename, &image);
21   if( ret < 0 )
22     cout << "Dear user, an err..." << endl;
23 }
```

Exceptions: Einfaches Beispiel

```
1  void funcA()
2  {
3    if( /*is there a problem?*/ )
4    {
5      throw string("Houston, we have a problem.");
6      // funcA is not further executed after we throw
7      // this exception!
8    }
9  }
10 void funcB()
11 {
12   try
13   {
14     //Call funcA, or call function that calls a
15     //function that... calls funcA.
16   }
17   // If an exception of type std::string is thrown within
18   // try-Block --> execute this catch-block.
19   catch(string& e)
20   {
21     cout << "Some error occurred: " << e << endl;
22   }
23   //Go on here, if no exception thrown or exception handled.
24 }
```

Exceptions: Ablauf

1. Angenommen in der Funktion `funcA` führen wir ein Statement `throw string("hello");` aus.
2. Dann springen wir aus dieser Funktion heraus und wir befinden uns dort, wo diese Funktion aufgerufen wurde.
3. Sind wir in einem `try`-Block, der einen `catch`-Block für die Exception vom Typ `string` besitzt, so führen wir den Code im `catch`-Block aus.
4. Ansonsten: zurück zu Schritt 2.

Exception: Mehr Details

- Exceptions können von beliebigem Typ sein.
  - `throw 10; //int`
  - `throw "hello"; //const char*`
  - `throw FileUnknownException(filename);`
- Manche Exceptions bringt C++ (STL) mit:
  - `std::exception //common base class`
  - `std::bad_alloc //if 'new' fails`
  - `std::ios_base::failure //iostream`
- Implementiere eigene Exception-Klassen (gewöhnliche Klasse)
  - um bei den `catch`-Blöcken unterscheiden zu können,
  - um Zusatzinformationen mitzugeben.

- `catch(...)` fängt Exceptions von jedem Typ.
- Ist `B` eine Unterklasse von `A`, so fängt `catch(A& e)` auch Exceptions vom Typ `B`.

- In einem `catch`-Block kann man erneut eine Exception werfen.
- `throw;` in einem `catch`-Block wirft die aktuelle Exception weiter.

Exceptions: Beispiel

```
1  void funcA()
2  {
3    int* array = new int[10000000];
4    if( /*Condition*/ )
5      throw "you can throw strings";
6    if( /*Condition*/ )
7      throw MyFileException("filename");
8  }
9  void funcB()
10 {
11   try {
12     //Call funcA, or call function that calls a
13     //function that... calls funcA.
14   }
15   catch(std::bad_alloc& e) {
16     cout << "Dear user, no more memory!" << endl;
17   }
18   catch(string& e) {
19     cout << "Some error: " << e << endl;
20   }
21   catch(MyFileException& e) {
22     cout << "File Error: " << e.getFilename() << endl;
23   }
24 }
```

Exceptions: Beispiel

```
1  string readFile(string filename)
2  {
3    if( /*file does not exist*/ )
4      throw FileNotExisitingException(filename);
5    if( /*no permissions to read file*/ )
6      throw FileWrongPermissionsException(filename);
7    /*Read file byte by byte.*/
8  }
9  Image readJPGFromFile(string filename)
10 {
11   string content = readFile(filename);
12   /*Interpret the JPG file format.*/
13 }
14 void displayFile(string filename)
15 {
16   try
17   {
18     Image image = readJPGFromFile(filename);
19   }
20   catch(FileNotExisitingException& e)
21   {
22     cout << "Dear user, an err..." << endl;
23   }
24 }
```

Exception handling versus assertions

Verwende Exceptions für Fehler, auf die **zur Laufzeit reagiert** werden kann:

- Beispiele:
  - Datei kann nicht geöffnet werden, falsches Dateiformat, etc.
  - Fehler in der Interaktion mit dem Anwender, etwa falsche Eingaben.
  - Speicher kann nicht reserviert werden.
- Verwende Exceptions nicht um gewöhnliche Programmlogik zu implementieren!
  - Die Abhandlung von Exceptions sollte die *Ausnahme* sein und nicht der Regelfall.
  - Jene Funktion, die eine Exception wirft, soll nicht davon abhängen, dass eine Exception an einer bestimmten Stelle behandelt wird!

Verwende assert um die **Logik des Programms** zu überprüfen:

- Asserts weisen auf Programmierfehler hin und dienen zur Fehlersuche und zur Überprüfung der Programmlogik.
  - Verwende Asserts nicht um auf falsche Eingaben zu reagieren!
- Asserts werden in der Regel nur im Debug-build aktiviert und im Release-build häufig deaktiviert.
  - `gcc -DNDEBUG main.cpp`