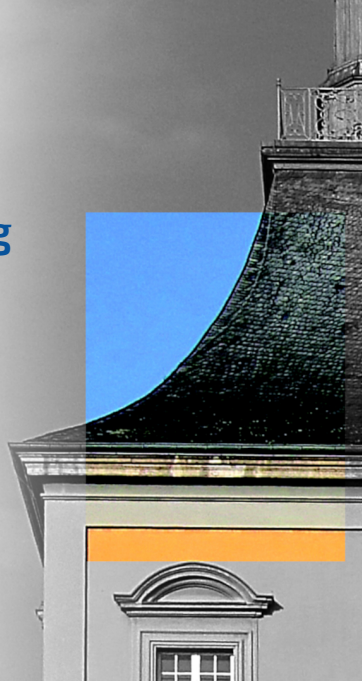
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



Gitversion: 'c0a60ee8268b408f4d4ed2f51908c808b1ea581f'

KURZE WIEDERHOLUNG

Iteratoren

UNIVERSITÄT BONN

Iteratoren in C++

Ein **Iterator** referenziert entweder ein (zum Erstellungszeitpunkt des Iterators) enthaltenes Element Containers oder der Iterator enthält die Information, dass er auf kein Element des Containers zeigt.

In C++ spricht man über die folgenden Iteratorarten und Iteratoroperationen.

Category	output	input	forward	bidirectional	random-access
Read		=*p	=*p	=*p	=*p
Access		->	->	->	-> [ ]
Write	*p=		*p=	*p=	*p=
Iterate	++	++	++	++ --	++ -- + - += -=
Compare		== !=	== !=	== !=	== != < > >= <=

UNIVERSITÄT BONN

Iteratoren in C++ definieren

Ein Forwarditerator muss mindestens folgende Member bereitstellen.

```
class mein_iterator : public std::iterator<
    std::forward_iterator_tag,  // iterator_category
    ElementTyp,                 // value_type
    std::ptrdiff_t,             // difference_type
    ElementTypPointer,          // pointer
    ElementTyp&                 // reference
  >
{
public:
    mein_iterator(/* Zur Initialisierung durch den Container */);
    mein_iterator& operator++();   // ++it
    mein_iterator operator++(int); // it++ [int ist ein Dummyparameter]
    bool operator==(mein_iterator other); // it == anderer_it
    bool operator!=(mein_iterator other); // it != anderer_it
    reference operator*();         // *it
};
```

Iteratoren

Elegant mit Iteratoren arbeiten

Ziel

Programmabschnitte die durch Container iterieren, sollen mit den den Funktionen aus `<algorithm>` umgesetzt werden

Sie lernen eine Auswahl von diesen Funktionen kennen

UNIVERSITÄT BONN

Die Standardalgorithmusbibliothek

Immer wenn wir eine Folge von Elementen erzeugen wollen, für alle Elemente eines Containers etwas produzieren möchten, die Elemente eines Containers umordnen möchten oder ein / alle Elemente eines Containers finden wollen, die eine Bedingung erfüllen, gibt es dazu eine Standardfunktion aus `<algorithm>`.

Diese Funktionen operieren auf einem Bereich `[Anfang, Ende)`, der durch zwei Iteratoren angegeben wird. Dabei beginnen diese Funktionen bei `Anfang` und enden sobald `Ende` erreicht ist (ohne `Ende` zu verarbeiten).

Wir stellen hier beispielhaft eine Auswahl der in `<algorithm>` definierten Funktionen vor.

Iteratoren

Elegant mit Iteratoren arbeiten - Unverändernde Operationen

UNIVERSITÄT BONN

All of / Any of / None of

Mit `std::all_of`, `std::any_of` und `std::none_of` wird zurückgegeben, ob alle, mindestens eins oder keins der Elemente des Bereichs eine Eigenschaft erfüllen.

```
std::set<int> zahlen({ 2, 6, 1, 3, 9, 4, 5 });
auto ist_gerade    = [] (int x) -> bool { return x %  2 == 0; };
auto ist_fuenffach = [] (int x) -> bool { return x %  5 == 0; };
auto ist_zehnfach  = [] (int x) -> bool { return x % 10 == 0; };

bool alle_gerade         = std::all_of (begin(zahlen), end(zahlen), ist_gerade);    // false
bool irgendwer_fuenffach = std::any_of (begin(zahlen), end(zahlen), ist_fuenffach); // true
bool keiner_zehnfach     = std::none_of(begin(zahlen), end(zahlen), ist_zehnfach);  // true
```

UNIVERSITÄT BONN

For each

Um eine Funktion für alle oder die ersten *n* Elemente eines Bereichs auszuführen, verwendet man `std::for_each` oder `std::for_each_n`.

```
std::set<int> zahlen({ 2, 6, 1, 3, 9, 4, 5 });
auto drucke_doppeltes = [] (int x) -> void { std::cout << 2*x << std::endl; };

std::for_each  (begin(zahlen), end(zahlen), drucke_doppeltes);
std::for_each_n(begin(zahlen), 4, drucke_doppeltes);
```

UNIVERSITÄT BONN

Count

Um zu Zählen wir oft ein ausgewähltes Element vorkommt oder wieviele Elemente eine Bedingung erfüllen, verwendet man `std::count` oder `std::count_if`.

```
std::vector<int> zahlen({ 2, 2, 1, 6, 1, 3, 4, 9, 4, 5 });
auto ist_gerade    = [] (int x) -> bool { return x %  2 == 0; };

int anzahl_4      = std::count   (begin(zahlen), end(zahlen), 4);          // 2
int anzahl_gerade = std::count_if(begin(zahlen), end(zahlen), ist_gerade); // 5
```

UNIVERSITÄT BONN

Find first of

Gegeben ein Containerobjekt `c` und eine Kandidatenmenge `m`. Um das erste Vorkommen eines Elements aus `m` in `c` zu bestimmen, verwendet man `std::find_first_of`.

```cpp
std::vector<int> c({ 2, 2, 1, 6, 1, 3, 4, 9, 4, 5 });
std::set<int> m({7, 3, 5});

auto erg = std::find_first_of(c.begin(), c.end(), m.begin(), m.end());
// Das Ergebnis ist ein Iterator, der entweder auf das erste gefundene Element zeigt,
// oder das Ende des Bereichs markiert.
if (erg == c.end()) {
  std::cout << "Kein Kandidat aus m ist in c enthalten" << std::endl;
} else {
  auto position = std::distance(c.begin(), erg);
  std::cout << "Wir haben '" << *erg << "' an Position " << position << " gefunden." << std::endl;
}
```

UNIVERSITÄT BONN

Lower Bound und Min

Gegeben ein Container `c` und ein vergleichbares Element `x`. Um das Minimum oder Maximum des Containers zu finden, nutzt man `std::min` und `std::min_element` oder aber `std::max` und `std::max_element`.

Falls der Container sortiert ist, findet man die kleinste untere oder größte obere Schranke von `x` mit `std::lower_bound` oder `std::upper_bound`.

Es gibt noch mehr Funktionen, die einen Bereich durchlaufen ohne ihn zu verändern

Haben Sie Fragen?

Iteratoren

Elegant mit Iteratoren arbeiten - Verändernde Operationen

UNIVERSITÄT BONN

Generate

Gegeben ein Containerobjekt `c` und ein Funktionsobjekt `f`. Um die Elemente eines Bereichs mit dem Rückgabewert des Funktionsobjekts zu überschreiben, verwendet man `std::generate`.

```
int zahl = 1;   // Wird von der Lambdafunktion fueller gecaptured.
auto fueller = [&zahl] () -> int { auto erg = zahl*zahl; zahl++; return erg; };
auto drucke = [] (int x) -> void { std::cout << x << std::endl; };

std::vector<int> c(7);
std::for_each(begin(c), end(c), drucke);
std::generate(begin(c), end(c), fueller);
std::for_each(begin(c), end(c), drucke);
```

UNIVERSITÄT BONN

Transform

Gegeben ein Containerobjekt `c` und ein Funktionsobjekt `f`. Um die Elemente eines Bereichs mithilfe des Funktionsobjekt zu transformieren und in einen, möglicherweise unterschiedlichen Container zu schreiben, verwendet man `std::transform`.

```
auto verdoppeln = [] (int x) -> int { return 2*x; };
auto drucke = [] (int x) -> void { std::cout << x << std::endl; };

std::set<int> c({1, 66, 2, 3, 4});
std::for_each(  begin(c),  end(c), drucke);
// Parameter: Von, Bis, Ziel, Funktionsobjekt
std::transform(cbegin(c), cend(c), begin(c), verdoppeln);
std::for_each(  begin(c),  end(c), drucke);
```

UNIVERSITÄT BONN

Remove and Erase

Gegeben ein sequentieller Container `c`, wollen wir feste Werte oder Werte die eine Bedingung erfüllen entfernen. Dazu verwenden wir das **C++-Remove-And-Erase-Idiom**. Die nicht-zu-löschenden Elemente werden mit `std::remove` oder `std::remove_if`, unter einhaltung der Reihenfolge, nach vorn gezogen[1]. Die Rückgabe ist ein Iterator der auf das erste, zu löschende Element zeigt. Mithilfe dieses Iterators und `std::erase` kann das Ende des sequentiellen Containers gelöscht werden.

```
std::vector<int> v({0,1,2,2,3,4});
auto rest = std::remove(begin(v), end(v), 2);
std::for_each(begin(v), end(v), drucke); // 0, 1, 3, 4, 3, 4
std::erase(rest.begin(), v.end());       // Entfernt die letzten beiden Elemente
```

Dieses Idiom erlaubt es, das Ende des sequentiellen Containers weiter zu verwenden (falls man `std::erase` nicht aufruft).

[1] Dazu werden Speicher- und Laufzeiteffiziente `move`-Assignments verwendet.

UNIVERSITÄT BONN

Unique and Erase

Gegeben ein sequentieller Container `c`, wollen wir mehrfach vorkommende Werte entfernen. Dazu verwenden wir das **C++-Remove-And-Erase-Idiom**. Die nicht-zu-löschenden Elemente werden mit `std::unique`, unter einhaltung der Reihenfolge, nach vorn gezogen[2]. Die Rückgabe ist ein Iterator der auf das erste, zu löschende Element zeigt. Mithilfe dieses Iterators und `std::erase` kann das Ende des sequentiellen Containers gelöscht werden.

```
std::vector<int> v({0,1,2,2,3,2,4});
auto rest = std::unique(begin(v), end(v));
std::for_each(begin(v), end(v), drucke); // 0, 1, 3, 4, 4, 4
std::erase(rest.begin(), v.end());       // Entfernt die letzten beiden Elemente
```

Dieses Idiom erlaubt es, das Ende des sequentiellen Containers weiter zu verwenden (falls man `std::erase` nicht aufruft).

[2] Dazu werden Speicher- und Laufzeiteffiziente `move`-Assignments verwendet.

UNIVERSITÄT BONN

Shuffle und Sample

Gegeben ein sequentieller Container `c`, wollen wir die Elemente zufällig permutieren, oder eine zufällige Stichprobe (ohne Zurücklegen) ziehen. Dazu verwenden wir die Funktionstemplates `std::shuffle` oder `std::sample`.

```
// Erstellt einen (benötigten) Zufallsgenerator
std::random_device rd;
std::mt19937 g(rd());

std::vector<int> v({0,11,2,3,44,5,66});
std::shuffle(v.begin(), v.end(), g);
std::for_each(begin(v), end(v), drucke);

std::vector<int> stichprobe(3);
std::sample(begin(y), end(y), begin(stichprobe), stichprobe.size(), g);
std::for_each(begin(stichprobe), end(stichprobe), drucke);
```

UNIVERSITÄT BONN

Sort und nth_element

Gegeben ein sequentieller Container `c`, wollen wir die enthaltenen Elemente sortieren. Dazu verwenden wir `std::sort` oder, falls die Reihenfolge von gleichen Elementen erhalten bleiben soll, `std::stable_sort`. Wenn die Elemente den Vergleichoperator `operator<` korrekt implementieren, muss kein Funktionsobjekt zum Vergleich übergeben werden.

```
std::vector<int> v({0,11,2,3,44,5,66});
std::sort(v.begin(), v.end());
std::for_each(begin(v), end(v), drucke);

// Absteigend sortieren
auto vergleicher = [] (int x, int y) -> bool { return x > y; };
std::sort(v.begin(), v.end(), vergleicher);
std::for_each(begin(v), end(v), drucke);
```

Mit `std::nth_element` wird der Container teilsortiert. Anschließend sind die ersten `n` Elemente des teilsortieren Containers, die ersten `n` Elemente einer vollständigen Sortierung des Containers.

Es gibt noch mehr Funktionen, die einen Bereich durchlaufen und ihn dabei verändern

Haben Sie Fragen?

Zusammenfassung

Sie eine Auswahl von Funktionen aus `<algorithm>` kennengelernt

Nutzen Sie diese wenn Sie in Programmabschnitten durch Container iterieren, um Ihren Code nachvollziehbarer und effizienter zu gestalten

Weitere C++-Details

Ziel

Wir haben in fast alle grundlegenden, wesentlichen, atomaren Sprachbausteine von C++ eingeführt

Wir ergänzen fast alle verbleibenden, grundlegenden, wesentlichen, atomaren Sprachbausteine

Weitere C++-Details

Weitere Zugriffsspezifikationen

Ziel

Wir lernen weitere Details zu Memberzugriffsspezifikationen

UNIVERSITÄT BONN

Erweiterter Memberzugriff für Subklassen

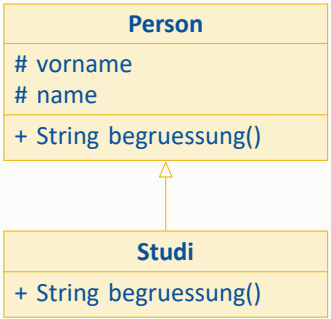
In einer Subklassenbeziehung `B` ◁— `D` ist es fast immer sinnvoll, dass der direkte Zugriff auf private `B`-Member innerhalb von `D` unzulässig ist. So ist sichergestellt, dass Änderungen der `B`-Attribute immer zu einem zulässigen `B`-Zustand führen.

Falls es gute Gründe gibt, kann `B` den Zugriff auf ausgewählte Member all seinen Subklassen erlauben. Der zugehörige Zugriffspezifizierer heißt `protected`. In UML werden solche Member durch die Angabe von `#` gekennzeichnet. In **C++** ergänzt man die `public`- und `private`-Section durch eine `protected`-Section.

UNIVERSITÄT BONN

Beispiel

Vorgabe: (Teil-)Zustände sollen nicht direkt ausgelesen werden können.




```
class Person {
public:
/* ... */
  virtual std::string begruessung() { return "Hallo"; }
protected:
  std::string vorname;
  std::string nachname;
};

class Studi : public Person {
public:
/* ... */                // protected: Legaler Zugriff ---v
  std::string begruessung() { return "Hi, ich bin " + vorname; }
};

int main () {
  Studi s;
  std::cout << s.vorname; // <--- protected: Illegaler Zugriff
}
```

UNIVERSITÄT BONN

Voller Memberzugriff für ausgewählte Freunde

Für Klassen B ist es fast immer sinnvoll, dass der direkte Zugriff auf private Member unzulässig ist.

Falls es gute Gründe gibt, kann B den Zugriff auf alle Member für ausgewählte (Member-)Funktionen erlauben. In **C++** nutzt man dazu die folgende Konstruktion.

```
class B {
/* ... */

friend SIGNATUR_DER_FUNKTION_1; // Dieser Funktion gewährt man vollen Zugriff
friend SIGNATUR_DER_FUNKTION_2; // Dieser Funktion gewährt man vollen Zugriff
/* ... */
};
```

Analog kann man den vollen Zugriff ausgewählten Klassen gestatten.

UNIVERSITÄT BONN

Beispiel

Typische Beispiele sind die Standardausgabe und externe Vergleichoperatoren.

```
class Studi {
private:
  int matrikelnummer;
  std::string vorname;
  /* ... */
// Erlaubt die Konstruktion std::cout << Studiobjekt;
friend std::ostream& operator<<(std::ostream&, const Studi&);
};

// Überlade std::ostream & operator<<(std::ostream &os, const T&);
std::ostream& operator<<(std::ostream &os, const Studi& s) { // Memberzugriff ist erlaubt
  return os << "Name: " << s.name << "; Matrikelnummer: " << s.matrikelnummer;
}

int main() {
  Studi s(/* Konstruktorparameter */);
  std::cout << s << std::endl;
}
```

UNIVERSITÄT BONN

Constness temporär entfernen

Wir definieren Membervariablen als konstant, wenn wir das (unbeabsichtigte) Überschreiben innerhalb einer Memberfunktion verhindern möchten. Damit kommunizieren wir den Leser:innen unseres Codes, dass dieser Wert unveränderlich ist.

Falls es dennoch „nötig“ ist, ein konstante Variable zu ändern, liegt das in den meisten Fällen an einer unvollständigen Designentscheidung und die Anpassung der Modellierung sollte in Betracht gezogen werden.

Falls es anschließend immernoch nötig ist ein konstanten Variable `x` vom Typ `const T` auf den Wert `w` zu setzen, kann die folgende Konstruktion verwendet werden.

```
const_cast<T&>(x) = w // Fasst x vom Typ const T als T& auf und ändert anschließend den Wert
```

Zusammenfassung

Wir haben gelernt, wie wir Zugriffsspezifikationen mit protected, friends und const_cast feiner gestalten

Haben Sie Fragen?

Weitere C++-Details

Weiteres zu Membern

Ziel

Wir lernen weniger oft genutzte Ausdrucksmöglichkeiten kennen, die in besonderen Fällen aber sehr sinnvoll sein können

UNIVERSITÄT BONN

Verschachtelte Klassen

Beim Entwurf von Bibliotheken möchte man sowohl eine öffentliche, möglichst unveränderliche Bibliotheksschnittstelle definieren, als auch nicht öffentliche, ggf. sich verändernde Implementierungsdetails bereitstellen. Beispielsweise möchte man einfach verkettete Listen bereitstellen, ohne sich auf die Gestalt der Knoten festzulegen.

Klassen, die ausschließlich als Implementierungsdetails dienen, werden oft als private Member einer Oberklasse definiert. Man spricht hier von **verschachtelten Klassen**.

Beispiel:

```
class Liste {
/* ... */
private:
  struct Node { /* ... */ }; // Implementierungsdetails
/* ... */
};
```

UNIVERSITÄT BONN

Weiteres Überschreiben verhindern

In seltenen Fällen gibt es eine Memberfunktion die überschrieben wird, aber die anschließend nicht mehr überschreibbar sein soll.

Im typischen Beispiel gibt es eine abstrakte Oberklasse `B`, die eine rein virtuelle Funktion `transfer_data` zum Übertragen von Daten bereit stellt. In einer Subklasse `D` wird `transfer_data` implementiert. Wir gehen nun davon aus, dass der Empfänger genau dieses Kommunikationsschema erwartet und eine Änderung des Schemas unzulässig ist. Also soll die Funktion `D::transfer_data` nicht überschrieben werden.

Mit `final` wird verboten, dass eine virtuelle Funktion überschrieben wird.

```
class B {
public:
  virtual void transfer_data() = 0; // Muss irgendwann überschrieben werden
};
class D : public {
public:
  final void transfer_data(); // Darf nicht weiter überschrieben werden
};
```

UNIVERSITÄT BONN

Destruktoren

Bei der Instanziierung eines Objekts vom Typ `T` wird der Konstruktor `T::T(/*Param*/)` aufgerufen. In C++ ist es außerdem möglich, eine letzte Funktion aufzurufen, wenn das Objekt vernichtet wird. Diese Funktion ist der **Destruktor** `~T::T(/*Param*/)`.

In den allermeisten Anwendungsfällen soll und muss kein Destruktor implementiert werden. Destruktoren werden häufig verwendet, wenn mit Rawpointern gearbeitet wird. Beispielsweise sollten Destruktoren genutzt werden, wenn man Shared Pointern selbstständig und nachvollziehbar implementieren möchte.

Bei der Instanziierung einer abgeleiteten Kasse wird zuerst die Konstruktoren der allgemeineneren Klasse ausgeführt. Bei der Vernichtung der Instanz wird zuerst der Destruktor der spezielleren Klasse ausgeführt (falls der Destruktor nicht virtuell ist).

UNIVERSITÄT BONN

Beispiel

```
class B {
public:
   B() { std::cout << " B::B ausgeführt. " << std::endl; }
  ~B() { std::cout << "~B::B ausgeführt. " << std::endl; }
};

class D : public B {
public:
   D() { std::cout << " D::D ausgeführt. " << std::endl; }
  ~D() { std::cout << "~D::D ausgeführt. " << std::endl; }
};

int main() {
  D d;
}
```

UNIVERSITÄT BONN

Virtuelle Destruktoren

Wir haben bereits gelernt, dass das Arbeiten mit Rawpointern zu ungewünschte Nebeneffekten führt. Das ist zum Beispiel bei polymorphen Typen der Fall. Hier warnt der C++Standard:

```
If the static type of the operand [of the operator *delete*] is different from its dynamic type,
the static type shall be a base class of the operand's dynamic type and the static type shall
have a virtual destructor or the behaviour is undefined.
```

Interessierte Zuhörer:innen finden dazu ein kurzes Beispiel auf der kommenden Folie.

Hinweis: Wenn wir nur mit Referenzen und Smart Pointern arbeiten, treten die angedeuteten Nebeneffekte nicht auf. Dafür sind Smart Pointer ein wenig Laufzeit- und Speicherintensiver.

```cpp
class Base1 {
public:
  virtual ~Base1() {
    std::cout << "Base1" << std::endl;
  }
};

class Derived1 : public Base1 {
public:
  Derived1() {
    x = std::make_shared<int>(1);
  }
  ~Derived1() override {
    std::cout << "Derived1" << std::endl;
  }
private:
  std::shared_ptr<int> x;
};

class Base2 {
public:
  ~Base2() {
    std::cout << "Base2" << std::endl;
  } // Base2 ist polymorph, da g virtuell
  virtual void g() {}
};
```

```cpp
class Derived2 : public Base2 {
public:
  Derived2() {
    x = std::make_shared<int>(1);
  }
  ~Derived2() {
    std::cout << "Derived2" << std::endl;
  }
private:
  std::shared_ptr<int> x;
};

int main () {
  Base1* b1ptr = new Derived1();
  Base2* b2ptr = new Derived2();

  delete b1ptr; // Ruft ~Derived1 auf
  delete b2ptr; // Ruft ~Base2 auf. Deshalb wird
  // Derived2::x nur vom Heap deallokiert. Aber
  // Derived2::x wird nicht durch ~Derived2
  // vernichtet. Also ist der durch x verwaltete
  // Speicher noch vorhanden
```

UNIVERSITÄT BONN

Statische Member

Sehr selten sind globale Variablen sinnvoll. Bei der objektorientierten Modellierung gehören globale Variablen üblicherweise zu einer Klasse.

Member, die unabhängig von einer Instanz existieren sollen, heißen **statisch** oder auch **Klassenmember**. Klassenmember können Membervariablen und Memberfunktionen sein. Das Keyword `static` gibt an, dass ein Member statisch ist.

Statische Memberfunktionen können offenbar nur auf statischen Membervariablen arbeiten, da keine Instanz vorliegen muss.

Beispiel

```
// HEADER
class Zaehler {
public:
   Zaehler() : ticks(0) { instanzen += 1; }
  ~Zaehler()            { instanzen -= 1; }
  int tick() { return ticks++; }
  static int get_anzahl_instanzen() { return instanzen; }
private:
  int ticks;
  static int instanzen;
};

// QUELLE    Die Initialisierung muss in der Quelldatei geschehen!
// Sonst geschieht die Initialisierung bei jedem Einbinden des Headers.
int Zaehler::instanzen = 0;

// DEMO
int main() {
  std::cout << Zaehler::get_anzahl_instanzen() << std::endl; // 0
  Zaehler a,b,c;
  std::cout << Zaehler::get_anzahl_instanzen() << std::endl; // 3
```

Enumerations

Um die Lesbarkeit zu erhöhen, können Enumerations angelegt werden. Jede Enumeration definiert einen neuen Typ sowie die möglichen Ausprägungen dieses Typs.

Beispielsweise führen wir einen neuen Typ ein, um über Farbenwerte zu sprechen.

```
enum Farbe { rot, gruen, blau };
Farbe x = rot;
if (x == blau) {
  std::cout << "Die Farbe ist blau." << std::endl;
}
```

Zusammenfassung

Wir haben weniger oft genutzte Ausdrucksmöglichkeiten kennengelernt, die in besonderen Fällen aber sehr sinnvoll sein können

Weitere C++-Details

Multiple Inheritance

Offene Frage

Kann man von mehreren Klassen erben?

Was ist bei Mehrfachvererbung zu beachten?

UNIVERSITÄT BONN

Multiple Inheritance

Wenn eine Subklasse durch Subklassenbildung aus mehreren Oberklassen entsteht, spricht man von **Multiple Inheritance** oder **Mehrfachvererbung**.

Hierbei gibt es mehrere Möglichkeiten, wie die Attribute der Subklasse aus den Attributen der Oberklasse zusammensetzt werden können. Das ist einer von mehreren Gründen, warum Mehrfachvererbung nur für sehr geübte Programmierer:innen zu nachvollziehbarem Code und Verhalten führen.

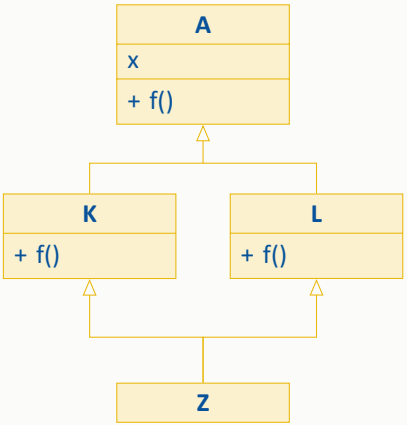
In **Java** ist Multiple Inheritance bei „normalen Klassen“ verboten, bei „Interfaces“ aber erlaubt.

In **C++** und **Python** ist Mehrfachvererbung erlaubt.

UNIVERSITÄT BONN

Das Diamondproblem

Die folgende Mehrfachvererbung wird **Diamondproblem** genannt.



Aus dieser Mehrfachvererbung führt zu folgenden Fragen. Dazu betrachtet man ein Z-Objekt `obj`.

- Welche Funktion wird durch `obj.f()` aufgerufen? `K::f` oder `L::f`?
- Wie sieht der Attributblock von `obj` aus? Sowohl `K` hat einen `A`-Anteil als auch `L`. Kommt der `A`-Anteil im `K`-Anteil und im `L`-Anteil jeweils einmal vor? Falls ja, besteht `obj` aus zwei `A`-Anteilen. Wie wird `obj` dann als ein `A`-Objekt aufgefasst? Falls nein, wie wird dann sichergestellt, dass die Veränderung des `A`-Anteils in `K` zu einem gültigen Zustand von `L` führt (und umgekehrt)?

UNIVERSITÄT BONN

Multiple Inheritance verwenden

Bei Multiple Inheritance muss vollständig erklärt werden, wie die Attribute zusammensetzt werden und welche Memberfunktionen ausgewählt gebunden werden. Die daraus resultierenden Nebeneffekte führen in der Praxis immer wieder zu unerwartetem Verhalten.

Man schließt diese Nebeneffekte aus, falls man nur dann von mehreren Klassen erbt, wenn diese keine Attribute enthalten und ausschließlich rein virtuelle Memberfunktionen bereitstellen. Dieser Ansatz wird in **Java** praktisch erzwungen sowie in **C++** und **Python** empfohlen.

Zusammenfassung

Wird von mehreren Klassen gerebt, spricht man von Multiple Inheritance

Falls die Oberklassen Attribute und nicht rein virtuelle Funktionen enthalten, führt das zu vielen Fragen, Nebeneffekten und oft schlechtem Code

Haben Sie Fragen?

Entwurfsprinzipien

Offne Frage

Welche objektorientierten Modellierungsansätze verwendet man bei Softwareprojekten mittlerer Größe?

UNIVERSITÄT BONN

Motivation

In den vergangenen Vorlesungen haben wir (fast) alle wesentlichen, atomaren Bestandteile der objektorientierten Modellierung kennen gelernt. Nun lernen wir **Modellierungsansätze** und **Entwurfsprinzipien** kennen, die diese atomaren Bestandteile kombinieren.

Die hier vorgestellten Entwurfsprinzipien finden in Projekten anwendung die mindestens mittlerer Größe ausweisen.

UNIVERSITÄT BONN

Übersicht

- Abstrakte Klassen
- SOLID und insbesondere das Dependency Inversion Principle
- Objekt- und Typbeziehungen
- Ausblick

Entwurfsprinzipien

Abstrakte Klassen und Interfaces

Ziel

Wir lernen abstrakte Klassen und Interfaces kennen

Diese werden bei der Modellierung von objektübergreifenden Interaktionsschnittstellen verwendet

UNIVERSITÄT BONN

Abstrakte Klassen und Interfaces

Wiederholung: Eine rein virtuelle Memberfunktion besitzt keine Implementierung. Wenn eine rein virtuelle Funktion überschrieben wird, ist sie weiterhin virtuell, aber nicht mehr rein virtuell.

Eine **abstrakte Klasse** ist eine Klasse, die über mindestens eine rein virtuelle Memberfunktion verfügt. Abstrakte Klassen können nicht instanziiert werden.

Ein **Interface** ist eine abstrakte Klasse, die ausschließlich über rein virtuelle Memberfunktionen[3] und keine Membervariablen verfügt.

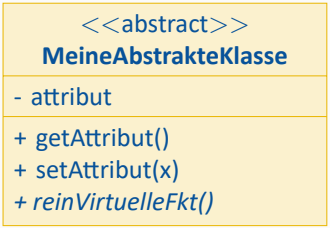
Also sind Interfaces abstrakter als gewöhliche abstrakte Klassen und diese sind abstrakter als gewöhnliche Klassen.

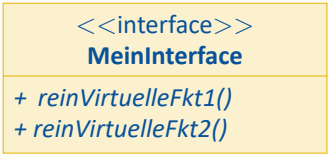
[3] Mit der Ausnahme, dass der Konstruktor nie virtuell ist.

UNIVERSITÄT BONN

Abstrakte Klassen und Interfaces in UML

In UML werden rein virtuelle Funktionen durch kursive Namen oder den Zusatz {abstract} gekennzeichnet. Zur besseren Lesbarkeit darf der Namen einer abstrakte Klassen oder eines Interfaces durch die Beschreibung «abstract» bzw. «interface» ergänzt werden.

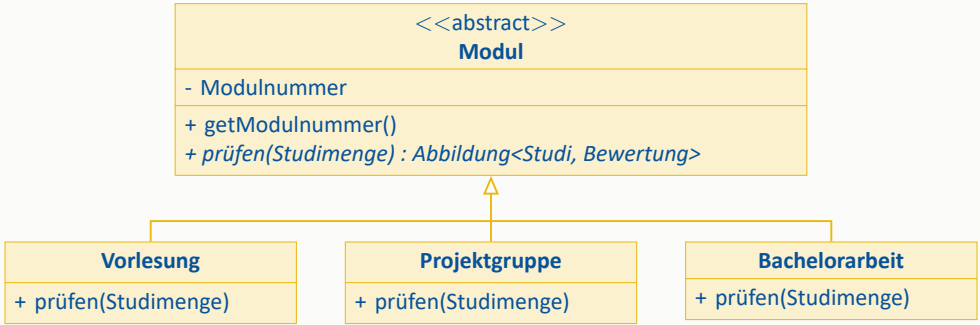




UNIVERSITÄT BONN

Abstrakte Klassen am Beispiel

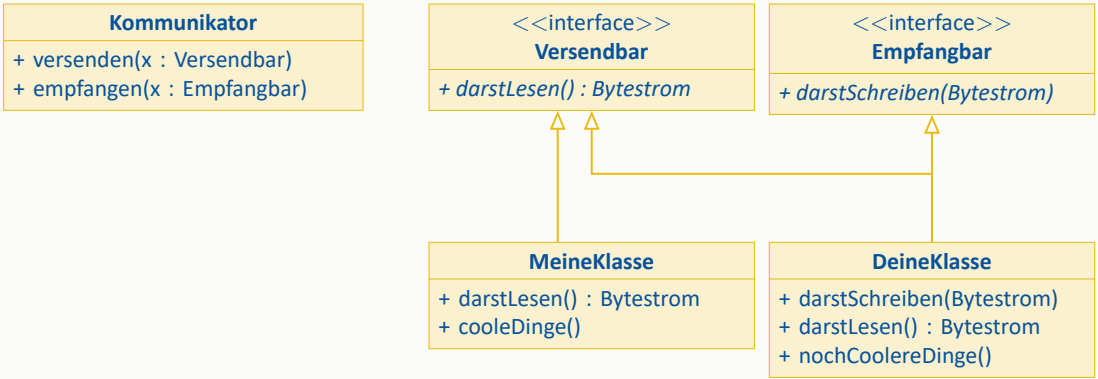
In unserem Studiengang hat jede abschlussrelevante Veranstaltung eine feste Modulnummer. Pro Modul wird regelmäßig entschieden, ob eine Menge von Studierenden das Modul bestanden haben oder nicht. Wie diese Prüfung genau stattfindet, hängt von dem Modul ab. Diesen Sachverhalt lässt sich wie folgt formulieren.



UNIVERSITÄT BONN

Interfaces am Beispiel

Wir wollen eine Klasse `Kommunikator` entwerfen, die beliebige Objekte über das Netzwerk versenden oder empfangen kann. Dazu müssen die Objekte versendbar bzw. empfangbar sein. Wir wählen den folgenden Modellierungsansatz.



Abstrakte Klassen und Interfaces in objektorientierten Sprachen

In **C++** und **Python** werden Interfaces nicht gesondert ausgezeichnet. Die oben besprochenen, wesentlichen Nebeneffekte von Multiple Inheritance müssen durch die vollständig durchdachte Modellierung ausgeschlossen werden.

In **Java** unterscheidet sprachlich zwischen abstrakte Klassen und Interfaces. Gewöhnliche Klassen dürfen von höchstens einer (abstrakten) Klassen erben, aber (gleichzeitig) von mehreren Interfaces. So werden die oben besprochenen, wesentlichen Nebeneffekte von Multiple Inheritance verhindert.

Zusammenfassung

Wir haben abstrakte Klassen und Interfaces kennengelernt

Diese werden bei der Modellierung von objektübergreifenden Interaktionsschnittstellen verwendet

Haben Sie Fragen?