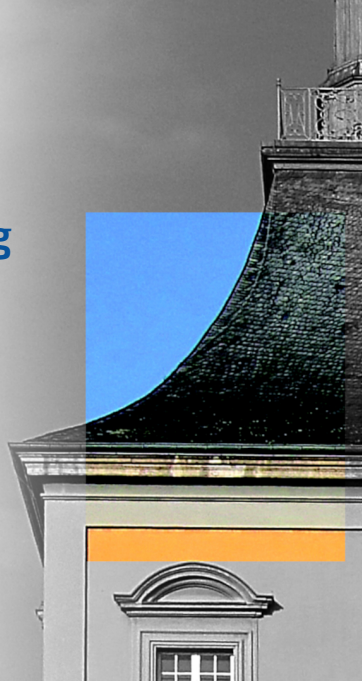
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

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



Probeklausur

Mittwoch 25.01.2023 während der Vorlesung

KURZE WIEDERHOLUNG

Subtypenbeziehungen

UNIVERSITÄT BONN

Ist-Ein-Faustregel und Liskovsches Substitutionsprinzip

(Subtypenbeziehungen als Faustregel)

Ein Objekt X vom Typ S ist ein **Subtyp** von einem Typ T, falls es Sinn macht zu sagen:

X ist auch ein T

(Liskovsches Substitutionsprinzip (vereinfacht))

Falls der Typ S ein Subtyp vom Typ T ist, dann können Objekte vom Typ T immer durch Objekte vom Typ S ersetzt werden, ohne die gewünschten Eigenschaften eines beliebigen Programms zu verändern.

UNIVERSITÄT BONN

Subtypenbeziehungen in UML

In UML wird die Subtypenbeziehung durch den Pfeil —▷beschrieben. Dabei zeigt der Pfeil vom spezielleren Subtyp zum allgemeineren Obertyp.

Beim Modellieren soll das Liskovsches Substitutionsprinzip eingehalten werden. Das muss durch die Modellierer:innen selbst sichergestellt werden.

Subtypenbeziehungen

Subtypenbeziehung durch Typerweiterung

UNIVERSITÄT BONN

Typerweiterung

Die **Typerweiterung** stellt die einfachste Form der Subtypenbeziehung dar. Hierbei wird ein gegebener Typ T durch das **Hinzufügen von Attributen** und das **Erweitern der Interaktionsschnittestelle** zu einem Subtyp S erweitert.

Subtypenbildung durch ausschließliche Typerweiterung ist untypisch. In Realweltanwendungen werden Subtypenbeziehung durch **Typerweiterung** und **Typkonkretisierung** gebildet.

Einschub: Qualified Identifier in C++

Um über Typen und Objekte zu sprechen, stellen wir uns folgende

Zwischenfrage

Wie werden Typen und Objekte eindeutig(er) in C++ benannt?

UNIVERSITÄT BONN

Typen und Objekte eindeutiger benennen

Um einen Typ oder ein Objekt eindeutiger zu benennen, verwenden wir **Qualified Identifier**, auch **einschränkende Bezeichner** genannt.

Durch qualified Identifier wird **relativ** oder **absolut** angegeben, wo ein Typ oder Objekt definiert ist (gleich mehr).

In C++ ist ein qualified Identifier eine Folge von Namen, die durch `::` getrennt wird. Jeder Name links von `::` bezeichnet dabei einen Namespace oder eine Klasse.

Zum Beispiel bezeichnet `DerivedClass::h(int)` die Memberfunktion `h` in der Klasse `DerivedClass`.

UNIVERSITÄT BONN

Absolute Identifier

Absolute Identifier beginnen mit `::...` und geben den vollen Namenspfad, beginnend bei globalen Namespace, an.

Beispiel:

```
#include <iostream>
void hallo() {
  // Im globalen Namespace ist der Namespace std definiert.
  // Der absolute Identifier von std ist also ::std.
  // Im Namespace std ist das Objekt cout definiert.
  // Also ist der absolute Identifier ::std::cout.
  ::std::cout << "Hallo";
}
```

UNIVERSITÄT BONN

Relative Identifier und Namensauflösung

Relative Identifier beginnen nicht mit `::...` und geben einen Namen an, der wie folgt aufgelöst wird.

Bei der **Auflösung** eines relativen Identifiers, wird bei dem linken Namen begonnen. Dieser wird lokal aufgelöst, das heißt, es wird zunächst im aktuellen Namespace oder in der aktuellen Klasse nach diesem Namen gesucht. Wird der Name dort nicht gefunden, wird der Name in den umgebenden Namespaces oder Klassen gesucht.

Wird der Name so nicht gefunden, schlägt die Auflösung fehl. Anderenfalls wird an der gefundenen Stelle (Namespace oder Klasse) der nächste Name gesucht. Dies geschieht sukzessive, bis der Name vollständig aufgelöst ist oder die Auflösung fehlschlägt.

UNIVERSITÄT BONN

Beispiel

```
#include <iostream>
void hallo() {
  std::cout << "Hallo"; // Absoluter Identifier ::std::cout
}

namespace mein_namespace {
  namespace noch_einer {
    class A {
    public:
      class Z {
        ...
      };
      ...
    };
  }

  noch_einer::A     // Findet A im lokal bekannten Namespace noch_einer
  // Der absolute Identifiert ist also ::mein_namespace::noch_einer::A
  noch_einer::A::Z  // Findet Z in der lokal bekannten Klasse noch_einer::A
  // Der absolute Identifiert ist also ::mein_namespace::noch_einer::A::Z
}
```

Zusammenfassung

Mit qualified Identifiern werden Typen und Objekte eindeutig(er) benannt

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung in objektorientierten Programmiersprachen

Offene Frage

Wie und in welchem Umfang drücken wir in objektorientierten Programmiersprachen aus, dass zwei Typen miteinander verwandt sind?

UNIVERSITÄT BONN

Subtypenbeziehung in objektorientierten Programmiersprachen I

In objektorientierten Programmiersprachen wird die Subtypenbeziehung fast ausschließlich durch **öffentliche Subklassenbildung** realisiert.

Wenn `D` (Derived) eine öffentliche Subklasse von `B` (Base) ist, dann ist `D` ein Subtyp von `B` und das Liskovsche Substitutionsprinzip ist erfüllt.

UNIVERSITÄT BONN

Subtypenbeziehung in objektorientierten Programmiersprachen II

In objektorientierten Programmiersprachen wird die Subtypenbeziehung fast ausschließlich durch **öffentliche Subklassenbildung** realisiert[1].

Mit wenigen Ausnahmen gilt außerdem: Wenn `D` ein Subtyp von `B` ist, dann ist `D` eine öffentliche Subklasse von `B`.

- In **C++**, **Java** und **Python** stehen primitive / grundlegende Datentypen in einer Subtypenbeziehung, aber in keiner öffentlichen Subklassenbeziehung.
- In **Java** sind `Interfaces` keine Klassen. Allerdings stehen Interfaces und zugehörige, implementierende Klassen in einer Subtypenbeziehung, wenn auch nicht in einer öffentlichen Subklassenbeziehung.
- In **Python** stehen Tuple zwar in einigen Subtypenbeziehung, aber in keiner öffentlichen Subklassenbeziehung.

[1] Es gibt außerdem nicht-öffentliche Subklassenbeziehung. Diese haben ganz spezielle Anwendungsgebiete, sind aber typischerweise nicht mit dem Liskovschen Substitutionsprinzip verträglich.

UNIVERSITÄT BONN

Öffentliche Subklassenbildung in Java und Python

Um in **Java** auszudrücken, dass die Klasse `DerivedClass` eine öffentliche Subklasse von `BaseClass` ist, verwendet man die folgende Konstruktion.

```
public class BaseClass {
  ...
}

public class DerivedClass extends BaseClass {
  ...
}
```

Um in **Python** verwendet man die folgende Konstruktion.

```
class BaseClass:
  ...

class DerivedClass (BaseClass):
  ...
```

UNIVERSITÄT BONN

Öffentliche Subklassenbildung in C++

Um in **C++** auszudrücken, dass die Klasse `DerivedClass` eine öffentliche Subklasse von `BaseClass` ist, verwendet man die folgende Konstruktion.

```
class BaseClass {
  ...
};

class DerivedClass : public BaseClass {
  ...
};
```

Zusammenfassung

In objektorientierten Programmiersprachen wird die Verwandtschaft von zwei Objekten, durch öffentliche Subklassenbildung ausgedrückt

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typerweiterung in C++

Offene Frage

Wie wird die Subtypenbeziehung durch Typerweiterung in C++ umgesetzt?

UNIVERSITÄT BONN

Öffentliche Subklassenbildung und Typerweiterung

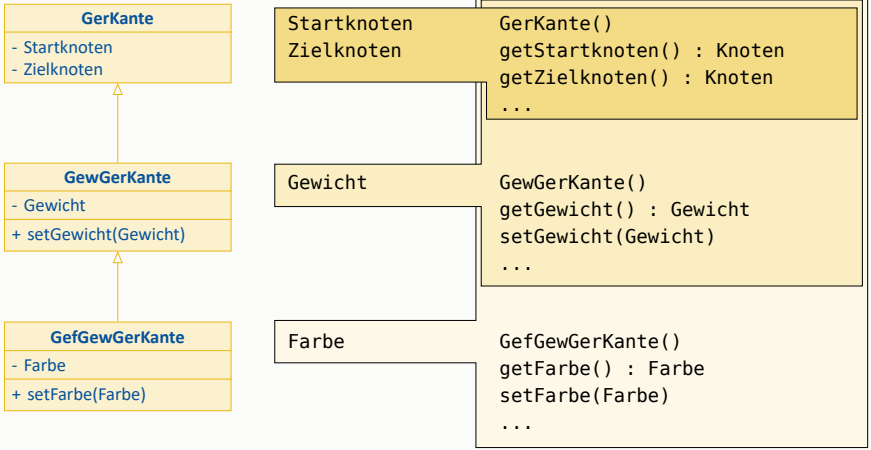
Wir konzentrieren uns zunächst auf **Subtypbildung durch Typerweiterung**.

- Hierbei werden alle **hinzugefügten** Membervariablen und Memberfunktionen genannt.
- Die öffentlichen Membervariablen und Memberfunktionen der Oberklasse stehen automatisch zur Verfügung. Man sagt, dass die öffentlichen Membervariablen und Memberfunktionen **vererbt** wurden.
- Die privaten Membervariablen und Memberfunktionen der Oberklasse sind Teil der Unterklasse, aber Sie dürfen in den hinzugefügten Memberfunktionen der Unterklasse nicht direkt verwendet werden.

UNIVERSITÄT BONN

Typerweiterung in UML

Auf der rechten Seite wird dargestellt, über welche Attribute und Methoden die jeweilige Klasse direkt zugreifen kann / zur Verfügung stellt.



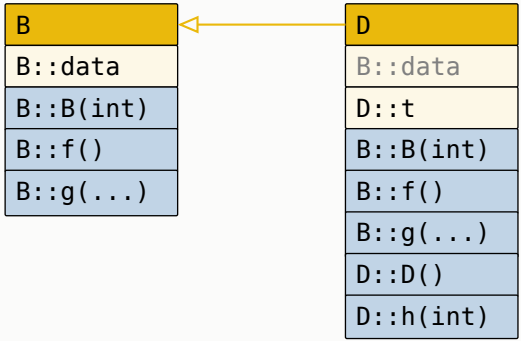
UNIVERSITÄT BONN

Typerweiterung in C++

```
class B { // Base Class
public:
  B(int x);
  void f();
  void g(std::string s);
private:
  int data;
};

// Derived Class
class D : public B {
public:
  DerivedClass();
  void h(int u);
private:
  std::string t;
}
```

Der Attributblock und die verfügbaren Funktionen sind visuell skizziert. Auf graue Attribute haben die Memberfunktionen der Form `D::...` keinen Zugriff.



UNIVERSITÄT BONN

Zugriff auf Memberfunktionen der Oberklasse

```
class B { // Base Class
public:
  B(int x);
  void f();
  void g(std::string s);
private:
  int data;
};

// Derived Class
class D : public B {
public:
  DerivedClass();
  void h(int u);
private:
  std::string t;
}
```

Durch diese öffentlichen Subklassenbildung verfügt D über die Memberfunktionen der Oberklasse B.

Der Zugriff auf die vererbten Memberfunktionen geschieht ohne oder mit qualified Identifiern.

```
// Implementierung der Memberfunktion void h(int u)
// der Klasse DerivedClass.
void DerivedClass::h(int u) {
  // Zugriff auf ::B::f mit einfachem Identifier.
  f();
  // Zugriff auf ::B::f mit relativem Identifier.
  B::f();
  // Zugriff auf ::B::f mit vollständigem Identifier.
  ::B::f();
}
```

UNIVERSITÄT BONN

Objektinstanziierung

Wird eine abgeleitete Klasse instanziiert, dann wird zuerst der Konstruktor der Oberklasse ausgeführt und anschließend der Konstruktor der abgeleiteten Klasse.

```
class BaseClass {
public:
  BaseClass()      {std::cout << "BaseClass()" << std::endl;}
  BaseClass(int x) {std::cout << "BaseClass(" << x << ")" << std::endl;}
};

class DerivedClass : public BaseClass {
public:
  DerivedClass()      : BaseClass(-100) {std::cout << "DerivedClass()" << std::endl;}
  DerivedClass(int x) : BaseClass(x)    {std::cout << "DerivedClass(" << x << ")" << std::endl;}
};

int main() {
  DerivedClass x;     // BaseClass(-100) [NEUE ZEILE] DerivedClass()
  DerivedClass y(42); // BaseClass(42)   [NEUE ZEILE] DerivedClass(42)
}
```

Zusammenfassung

In C++ erschaffen wir Subtypenbeziehung typischerweise durch öffentliche Subklassenbildung

In C++ wird die reine Typerweiterung durch die Angabe der zusätzlichen Membervariablen und Memberfunktionen realisiert

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung und Typerweiterung im Speicher

Offene Frage

Wir wird öffentliche Subklassenbildung mit reiner Typerweiterung unterhalb von C++ realisiert?

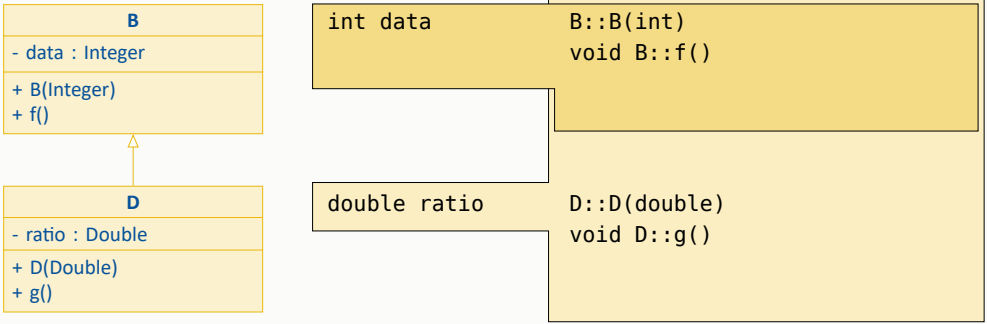
Disclaimer

Wir beantworten die Frage erst visuell und anschließend textuell

UNIVERSITÄT BONN

Grundlegendes Beispiel in UML

Auf der rechten Seite wird dargestellt, über welche Attribute und Methoden die jeweilige Klasse direkt zugreifen kann / zur Verfügung stellt.



UNIVERSITÄT BONN

Grundlegendes Beispiel in C++

Wir diskutieren, wie das folgende Beispiel im Speicher realisiert wird.

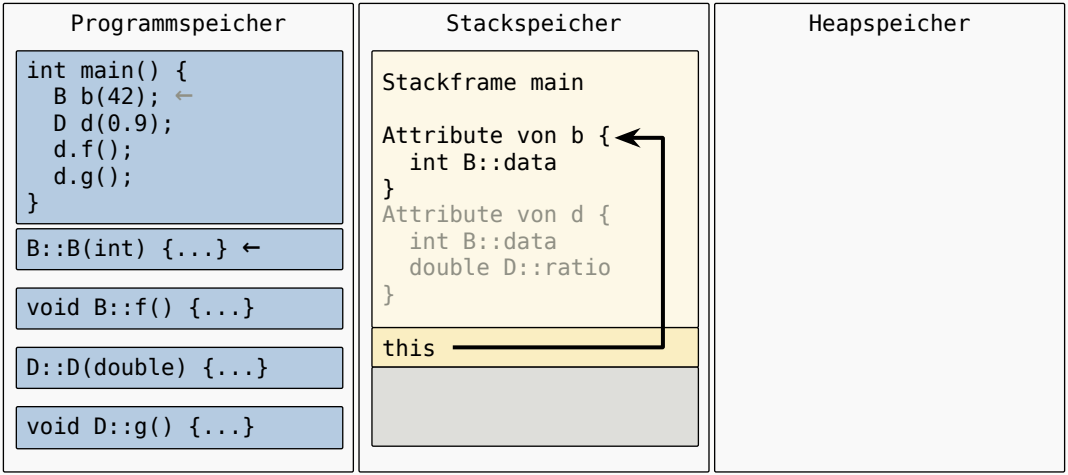
```
class B {
public:
  B(int t) : data(t) {std::cout << "B(" << t << ")" << std::endl;}
  void f() { ... }
private:
  int data;
};

class D : public B {
public:
  D(double x) : B(13), ratio(x) {std::cout << "D(" << x << ")" << std::endl;}
  void g() { ... }
private:
  double ratio;
};

int main() {
  B b(42);  // B(42)
  D d(0.9); // B(13) [NEUE ZEILE] D(0.9)
}
```

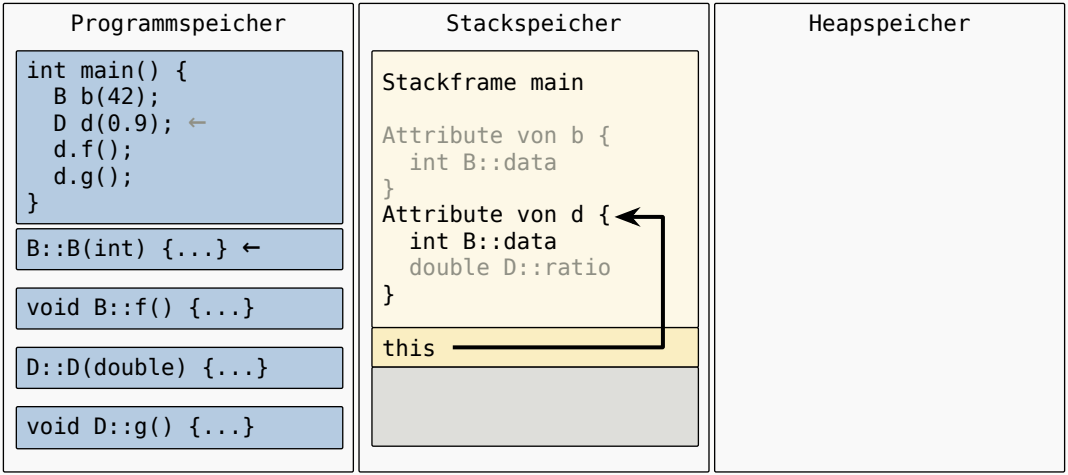
UNIVERSITÄT BONN

Grundlegendes Beispiel im Speicher



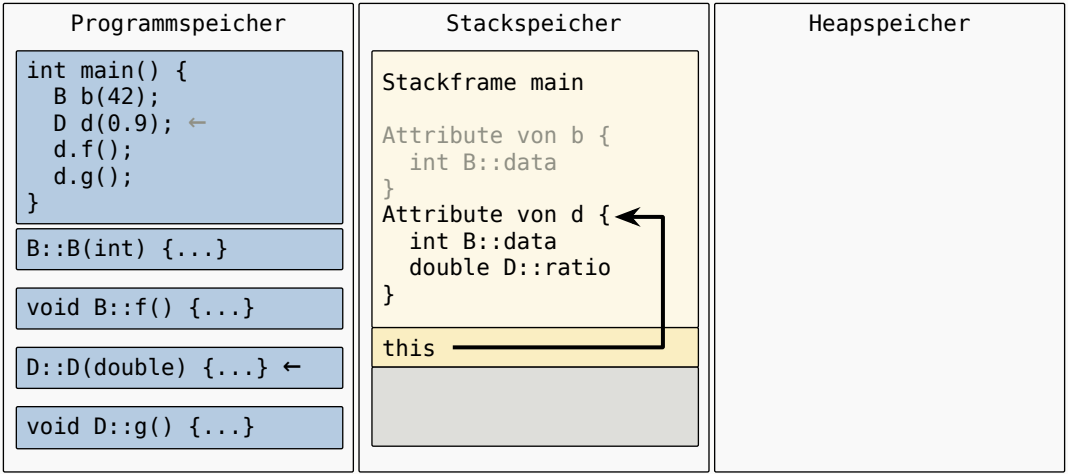
UNIVERSITÄT BONN

Grundlegendes Beispiel im Speicher



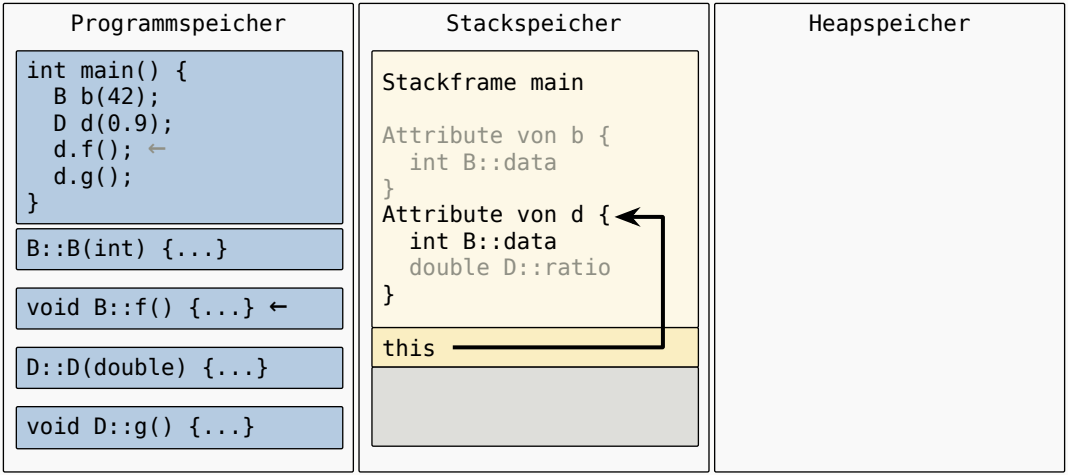
UNIVERSITÄT BONN

Grundlegendes Beispiel im Speicher



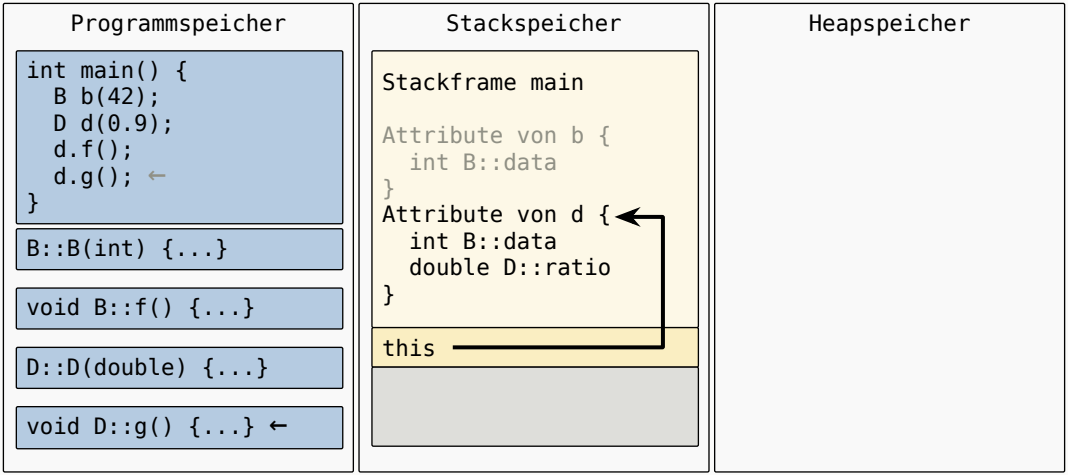
UNIVERSITÄT BONN

Grundlegendes Beispiel im Speicher



UNIVERSITÄT BONN

Grundlegendes Beispiel im Speicher



UNIVERSITÄT BONN

Textuelle Zusammenfassung Typerweiterung im Speicher I

Erinnerung: Jedes Objekt verfügt über (einmal im Programmspeicher abgelegte) Memberfunktionen und besteht aus (pro Objekt existierenden, auf dem Stack oder Heap abgelegten) Attributen.

Bei der **reinen Typerweiterung** durch öffentliche Subklassenbildung, stehen der Subklasse alle öffentlichen Memberfunktionen der Oberklasse zur Verfügung und werden durch neue Memberfunktionen ergänzt. Im (Stack- oder Heap-)Speicher wird der Attributblock der Oberklasse durch die Attribute der Subklasse ergänzt. Allerdings verbietet der Compiler den neuen Memberfunktion den Zugriff auf die privaten Attribute der Oberklasse.

Bemerkung: Da solche Subklassen technisch auch Memberfunktionen der Oberklasse aufrufen, stehen diesen Memberfunktionen die Attribute der Oberklasse zur Verfügung.

Bemerkung: Durch dieses Verhalten ist das Liskovsche Substitutionsprinzip erfüllt.

UNIVERSITÄT BONN

Textuelle Zusammenfassung Typerweiterung im Speicher II

Beim Instanziieren von Objekten wird zuerst der Konstruktor der Oberklasse aufgerufen und anschließend der Konstruktor der Subklasse. Dadurch ist sichergestellt, dass die Oberklasse einen zulässigen Zustand enthält, bevor die Subklasse mit der Oberklasse interagieren kann.

Zusammenfassung

Wir haben gelernt wie die Subklassenbildung mit reiner Typerweiterung im Speicher realisiert wird

Haben Sie Fragen?

Subtypenbeziehungen

Namehiding

UNIVERSITÄT BONN

Probleme im Studium

Was sind nicht-fachliche Hindernisse, die im Studium auftauchen können?

- Unterrepräsentiert oder alleine sein
- Sich unwohl oder nicht willkommen fühlen
- Leistungsdruck, Konkurrenzgehabe
- (Psychische) Gesundheit, familiäre Situation
- Unterbewusste Vorurteile, Microaggressionen
- Diskriminierende Kommentare oder Witze, Belästigung
- Geschlechtsbasierte Gewalt, Sexismus
- ...

UNIVERSITÄT BONN

Beispiele

- „Mir ist nicht eingefallen, dass Sie auch eine Frau sein könnten.“
- Jemand löst ungefragt Aufgaben für mich.
- Nach dem ersten Tutorium: „Mach du mal die Theorieaufgaben, die Praxisaufgaben können Studentinnen eh nicht so gut.“
- „Das ist eine einfache Frage, da kann ich mal eine Frau drannehmen.“
- „Die hat die Stelle doch nur bekommen, weil es eine Frauenquote gibt.“
- Ein Student kommt mir wiederholt im Seminarraum zu nahe, obwohl ich ihm sage, er soll etwas zurück gehen.
- „Wenn ich Frauen belästigen würden, wären Studentinnen die so ne Tasche tragen wie du als erstes dran.“

UNIVERSITÄT BONN

Definitionen

- **Gewalt** = Handlung oder Verhalten, das anderen schadet.
- **Geschlechtsbasierte Gewalt** = Gewalt gegen eine Person aufgrund ihres Geschlechts, oder Gewalt, die Personen eines Geschlechts überproportional häufig betrifft. Es gibt viele verschiedene Formen, z.B. Physisch, Sexuell, Psychologisch, Finanziell oder Digital. [2]

[2] Source:Lipinsky, A., Schredl, C., Baumann, H., Humbert, A., Tanwar, J. (2022). Gender-based violence and its consequences in European Academia, Summary results from the UniSAFE survey. Report, November 2022. UniSAFE project no.101006261.

UNIVERSITÄT BONN

Zahlen und Fakten

Studie mit über 40.000 Teilnehmenden aus 15 europäischen Ländern.

- 62% der Befragten haben mindestens eine Form von geschlechtsbasierter Gewalt an der Universität erfahren.
- 66% der weiblichen, 56% der männlichen und 74% der nicht-binären Befragten sind betroffen.
- Nur 13% der Vorfälle werden gemeldet.
- 63% der Betroffenen werden unzufrieden mit Ihrem Studiumsverlauf, 50% erzielen schlechtere Leistungen oder ziehen sich sozial zurück.

UNIVERSITÄT BONN

Was können wir tun?

Wie kann ich anderen helfen?

- Sei dir des Problems bewusst, informiere dich und andere.
- Reflektiere dein eigenes Verhalten
- Einschreiten, Verhalten kritisieren, Stellung beziehen
- Freund:innen unterstützen, zuhören, Zeit geben
- Eine angenehmere und persönliche Atmosphäre schaffen, miteinander reden.

Was kann ich tun, wenn es mir passiert?

- Regiere direkt in der Situation, oder auch danach.
- Denk daran, du bist nicht alleine!
- Rede mit anderen darüber, hol dir Hilfe bei der Fachschaft, Gleichstellungs-AG oder der zentralen Gleichstellungsbeauftragten.

Ziel

Sie lernen welche Nebeneffekte es hat, wenn Sie in einer Subklassenbeziehung Membernamen mehrfach einführen

UNIVERSITÄT BONN

Namehiding

Wenn in einer Klasse `S` der Name einer Membervariablen oder Memberfunktion eingeführt wird, dieser Name aber bereits in einer Oberklasse `T` von `S` eingeführt wurde, spricht man von **Namehiding**, auch **Namensverdeckung genannt**.

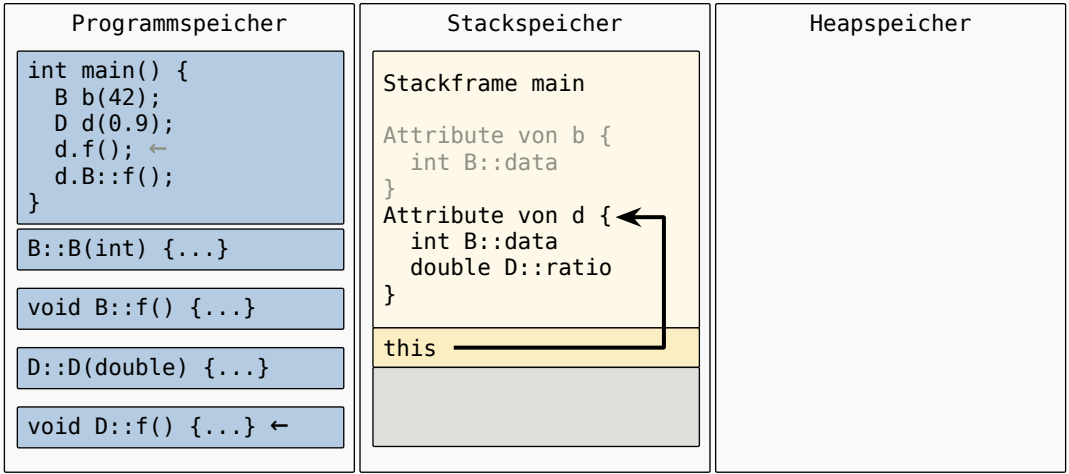
In `S` wird der Name `N` zu `S::N` aufgelöst und verdeckt somit den Namen `T::N`. Um in einer Memberfunktion von `S` auf das Member `T::N` zuzugreifen, muss der einschränkende Name `T::N` verwendete werden.

Durch Namehiding können sowohl Membervariablen als auch Memberfunktionen verdeckt werden.

UNIVERSITÄT BONN

Beispiel

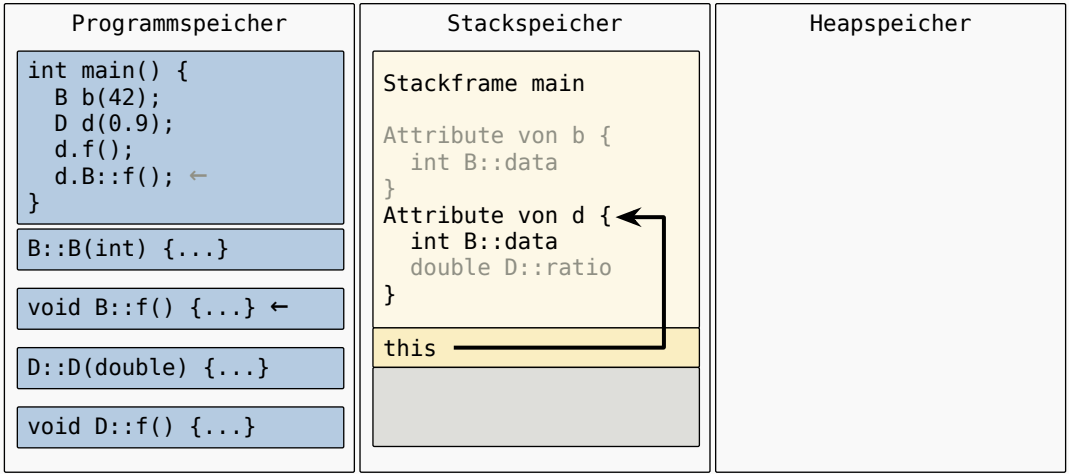
Die Klasse `D` erbt von `B` und der Name `B::f` wird durch den Namen `D::f` verdeckt.



UNIVERSITÄT BONN

Beispiel

Die Klasse `D` erbt von `B` und der Name `B::f` wird durch den Namen `D::f` verdeckt.



UNIVERSITÄT BONN

Memberfunktionen verdecken und überladen

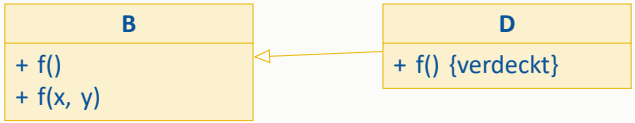
Wenn Memberfunktionen überladen und überdeckt werden, kommt es zu (möglicherweise unerwarteten Neben-)effekten.

Beim Funktionsaufruf bestimmt der Compiler, welche Funktion verwendet werden soll. Gegeben ein Objekt `obj` von Typ `D`, dann bestimmt der Compiler **zuerst** den vollen Namen von `obj.f(...)` und **anschließend**, welche der überladenen Varianten verwendet wird.

UNIVERSITÄT BONN

Beispiel

Im folgenden Beispiel wird `B::f` überladen und verdeckt.



Angenommen wir haben ein Objekt `obj` vom Typ `D`.

- Beim Kompilieren des Befehls `obj.f()` wird zuerst der volle Name als `D::f` bestimmt. Anschließend wird festgestellt, dass `D::f()` in `D` definiert ist.
- Beim Kompilieren des Befehls `obj.f(x,y)` wird zuerst der volle Name als `D::f` bestimmt. Anschließend wird festgestellt, dass `D::f(x,y)` in `D` nicht definiert ist, denn nur `B::f()`, `B::f(x,y)` und `D::f()` sind definiert.

UNIVERSITÄT BONN

Namehiding und guter Stil

Namehiding versteckt Membernamen von Oberklassen. Fast immer führt Namehiding zu schlecht nachvollziehbarem und schlecht wartbarem Code. Es gibt nur sehr wenige sinnvolle Anwendungsfälle für Namehiding.

Zusammenfassung

Namehiding versteckt Membernamen von Oberklassen und ist fast immer ein klares Zeichen für schlechten Stil

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typkonkretisierung

Offene Frage

Warum reicht die reine Typerweiterung nicht aus um Objekte zu modellieren?

Wie löst die Typkonkretisierung die auftretenden Probleme?

UNIVERSITÄT BONN

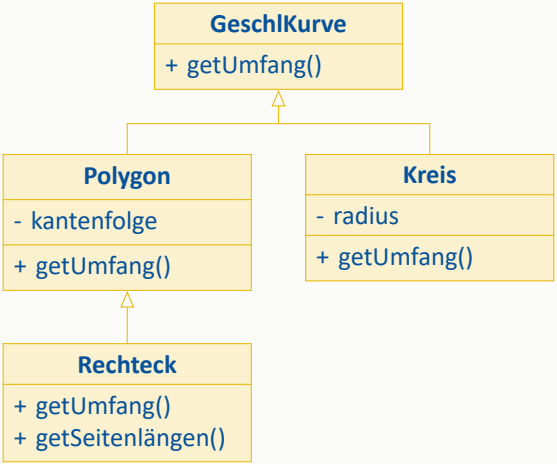
Motivation

Bei der reinen Typerweiterung sind wir von einem gegebenen Typ ausgegangen und haben diesen um Attribute und Interatkionsmöglichkeiten erweitert.

Das Modellieren von Subtypen verläuft sehr oft auch in die andere Richtung. Gegeben „ähnliche“ Typen `D_1`, . . ., `D_N` wollen wir einen gemeinsamen, möglichst konkreten Obertyp `B` beschreiben. Wir erklären beispielhaft, warum die reine Typerweiterung hierbei oft ungeeignet ist.

UNIVERSITÄT BONN

Motivierendes Beispiel



Die Umfangsberechnung von Polygonen und Kreisen geschieht unterschiedlich. Hier kann keine allgemeine, direkte Umfangsberechnungsfunktion angegeben werden ohne die Modellierung von Polygonen oder Kreisen zu ändern.

Dennoch ist es offensichtlich, dass wir einen möglichst konkreten Obertyp haben möchten. Dieser soll eine Memberfunktionen `getUmfang` besitzen, die dann von den Subtypen **konkretisiert** wird.

UNIVERSITÄT BONN

Probleme der reinen Typerweiterung

Falls wir geschlossene Kurven, Polygone, Kreise und Rechtecke mit reiner Typerweiterung umsetzen, dann wird folgender, naheliegender Code nicht das gewünschte Ergebnis liefern.

```
void drucke_umfang(const GeschlKurve& K) {
  std::cout << "Der Umfang ist" << K.getUmfang() << "." << std::endl;
}
```

Wenn wir in diese Funktion ein Objekt von Typ `Polygon` einsetzen, wird es als Objektreferenz vom Typ `GeschlKurve` behandelt und somit wird die Memberfunktionen `GeschlKurve::getUmfang` aufgerufen[3]. Wir möchten aber, dass die „konkretisierte“ Memberfunktionen `Polygon::getUmfang` aufgerufen wird.

[3] Wie wir sehen werden liegt das daran, dass wir keine polymorphen Objekte verwenden