Vererbung, Polymorphie

Paul Raffer

2021-04-11

Inhaltsverzeichnis

Copyright © Paul Raffer 2021

1 Datenkapselung

(siehe Ausarbeitung von N. M.)

1.1 Sichtbarkeit von Eigenschaften und Methoden

Zugriffsmodifikator	UML	Eigene Klasse	Abgeleitete Klasse	Außerhalb
öffentlich	+	sichtbar	sichtbar	sichtbar
geschützt	#	sichtbar	sichtbar	nicht sichtbar
privat	-	sichtbar	nicht sichtbar	nicht sichtbar

1.1.1 Sichtbarkeit 'öffentlich'

Auf öffentliche Eigenschaften und Methoden kann von überall zugegriffen werden.

1.1.2 Sichtbarkeit 'geschützt'

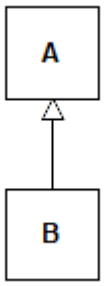
Auf geschützte Eigenschaften und Methoden kann nur innerhalb der eigenen Klasse und von allen davon abgeleiteten Klassen zugegriffen werden. (siehe 2 Vererbung)

1.1.3 Sichtbarkeit 'privat'

Auf private Eigenschaften und Methoden kann nur innerhalb der eigenen Klasse zugegriffen werden.

2 Vererbung

Vererbung (englisch inheritance) ist ein Konzept der objektorientierten Programmierung, das meist in Kombination mit Polymorphie eingesetzt wird (siehe 3.1.2 Vererbungspolymorphie), aber auch eigenständig sinnvoll verwendet werden kann. Wenn z.B. Klasse B von Klasse A erbt, werden alle Eigenschaften und Methoden der Basisklasse A (auch Ober-, Super- oder Elternklasse) in die abgeleitete Klasse B (auch Unter-, Sub- oder Kindklasse) übernommen. Dies hat den Vorteil, dass Eigenschaften und Methoden aus einer bestehenden Basisklasse wiederverwendet werden können, und somit doppelter Code und Schreibarbeit vermieden werden. In UML (Unified Modeling Language) wird Vererbung durch einen Pfeil von einer abgeleiteten Klasse zu dessen Basisklasse dargestellt.



2.1 Datenkapselung im Rahmen der Vererbung

Wenn eine Klasse von einer anderen erbt, kann sich die Sichtbarkeit der vererbten Eigenschafen und Methoden ändern.

Copyright © Paul Raffer 2021

Sichtbarkeit in ...			
Basisklasse	**abgeleiteter Klasse (erbt ... von Basisklasse)**		
	öffentlich ("ist-ein")	**geschützt** ("ist-implementiert-mit")	**privat** ("ist-implementiert-mit")
öffentlich →	öffentlich	geschützt	privat
geschützt →	geschützt	geschützt	privat
privat →	nicht vererbt	nicht vererbt	nicht vererbt

Es gibt 3 Arten der Vererbung: die öffentliche, die geschützte und die private Vererbung.

Bei der öffentlichen Vererbung bleiben geschützte Eigenschafen und Methoden der Basisklasse auch in der abgeleiteten Klasse geschützt. Öffentliche Eigenschafen und Methoden bleiben öffentlich. Öffentliche Vererbung sollte immer eine "ist-ein"-Beziehung darstellen. Jedoch sollte nicht jede "ist-ein"-Beziehung durch Vererbung dargestellt werden (siehe 3.1.2.2 Kreis-Ellipse-Problem). Diese Art der Vererbung wir am häufigsten verwendet.

Bei der geschützten Vererbung sind alle öffenlichen und geschützten Eigenschafen und Methoden der Basisklasse in der abgeleiteten Klasse geschützt. Geschützte Vererbung sollte immer eine "ist implementiert mit"-Beziehung darstellen

Bei der privaten Vererbung sind alle öffentlichen und privaten Eigenschafen und Methoden der Basisklasse in der abgeleiteten Klasse privat. Private Vererbung solle wie geschützte Vererbung immer eine "ist implementiert mit"-Beziehung darstellen. Eine gute Alternative zur geschützten und privaten Vererbung bietet das Layering (auch Komposition genannt). Beim Layering wird statt von einer Klasse zu erben, eine Eigenschaft der Klasse hinzugefügt. Layering stellt eine "hat-ein"- und somit auch eine "ist-implementiert-mit"-Beziehung dar. Layering sollte wenn möglich gegenüber geschützter oder privater Vererbung bevorzugt werden.

Private Eingenschafen und Methoden der Basisklasse werden nie in die abgeleitete Klasse vererbt.

Beispiel:
Eine C++ Klasse Point2d, deren Objekte Punkte auf einer Ebene darstellen, existiert bereits.

```
struct Point2d {
    int x;
    int y;
};
```

Nun sollen auch Punkte im Raum erstellt werden können. Dafür wird die Klasse Point3d geschrieben, welche unterschiedlich programmiert werden kann.

Lösung ohne Vererbung:
Natürlich könnte die Klasse von Grund auf neu geschrieben werden.

```
struct Point3d {
    int x;
    int y;
    int z;
};
```

Diese Lösung ist nicht optimal, da die Eingenschaften x und y aus der Klasse Point2d nicht wiederverwendet werden.

Lösung mit öffentlicher Vererbung:
Mit öffentlicher Vererbung können die Eigenschaften 'x' und 'y' aus der Klasse 'Point2d' für die Klasse 'Point3d' wiederverwendet werden.

```
struct Point3d : Point2d {
    int z;
};
```

Obwohl hier öffenliche Vererbung verwendet wurde, ist es wahrscheinlich nicht sinnvoll diese Klassen vielgestaltig zu verwenden, da die Aussage "Ein Punkt im Raum (Point3d) ist ein Punkt auf einer Ebene (Point2d)." nicht stimmt.

Copyright © Paul Raffer 2021

```
Point2d* point = new Point3d; // Moeglich, aber nicht erwuenscht!
```

Diese Zeile würde zwar vom Compiler übersetzt werden, macht aber wenig Sinn. Leider gibt es keine Möglichkeit eine solche Verwendung zu verhindern. Deshalb ist auch die Lösung mit öffentlicher Vererbung nicht optimal.

Lösung mit privater Vererbung:

```
struct Point3d : private Point2d {
    int z;
    // Eigenschaften aus Point2d wieder oeffenlich machen:
    public: using Point2d::x;
    public: using Point2d::y;
};
```

Wenn man von der Klasse 'Point2d' privat erbt, hat man keines der Probleme der anderen beiden Lösungen, allerdings muss man die Eigenschaften 'x' und 'y' in der abgeleiteten Klasse wieder öffentlich machen, da diese bei der privaten Vererbung privat werden. In diesem Fall hat diese Lösung keinen Vorteil im Vergleich zur ersten, da der Code sogar länger ist. Wenn aber 'x' und 'y' Methoden statt Eigenschaften sind, lohnt es sich sehr wohl private Vererbung zu verwenden, da diese dann nicht doppelt implementiert werden müssen.

2.2 Schnittstellenvererbung

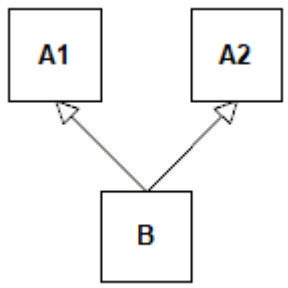
Bei der Schnittstellenvererbung wird nur die Signatur, aber keine Standardimplementierung, einer Methode der Basisklasse in die abgeleitete Klasse übernommen. Deshalb MÜSSEN alle Methoden, die mittels Schnittstellenvererbung vererbt werden, in der abgeleiteten Klasse implementiert werden. In Java ist das Typensystem zweigeteilt, in normale Klassen und Interfaces, die von Klassen implementiert werden können. In C++ können Interfaces mit abstrakten Klassen, die nur rein virtuelle Methoden enthalten, nachgebildet werden.

2.3 Implementierungsvererbung

Bei der Implementierungsvererbung wird im Gegensatz zur Schnittstellenvererbung nicht nur die Signatur, sondern auch die Implementierung einer Methode in die abgeleitete Klasse übernommen. So kann beispielsweise von der Basisklasse ein Standardverhalten vorgeschlagen werden, welches aber, falls Notwendig, von der abgeleiteten Klasse überschrieben werden kann.

2.4 Mehrfachvererbung

Wenn eine Unterklasse von mehr als einer Oberklasse erbt spricht man von Mehrfachvererbung. Mehrfachvererbung wird verwendet, um Features mehrerer Klassen in einer Klasse zu vereinen. Mehrfachinterfacevererbung ist in den meisten Fällen problemlos möglich. Mehrfachimplementierungsvererbung jedoch lassen viele Sprachen nicht zu, da diese oft zu fehleranfälligem und unübersichtlichem Code führt. Ein weiteres Problem der Mehrfachimplementierungsvererbung ist, dass sich Implementierungen aus unterschiedlichen Basisklassen, widersprechen können.

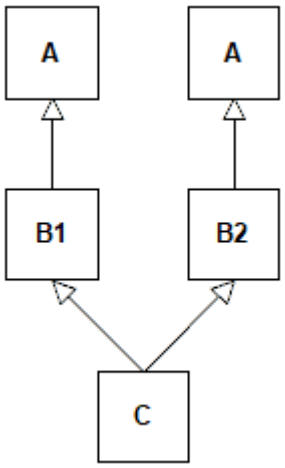


2.4.1 Diamond-Problem

Das Diamond-Problem ist eines der Probleme, zu denen Mehrfachvererbung führen kann. Dieses tritt auf, wenn eine abgeleitete Klasse C über mehr als einen Pfad von derselben Basisklasse A erbt. Dies ist problematisch,

Copyright © Paul Raffer 2021

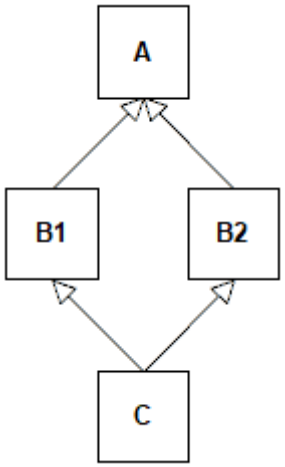
da die Eigenschaften und Methoden aus Klasse A mehrfach an Klasse C vererbt werden, und deshalb in der abgeleiteten Klasse A öfter existieren.



In Java kann das Diamond-Problem nicht auftreten, da es keine Mehrfachimplementierungsvererbung gibt.

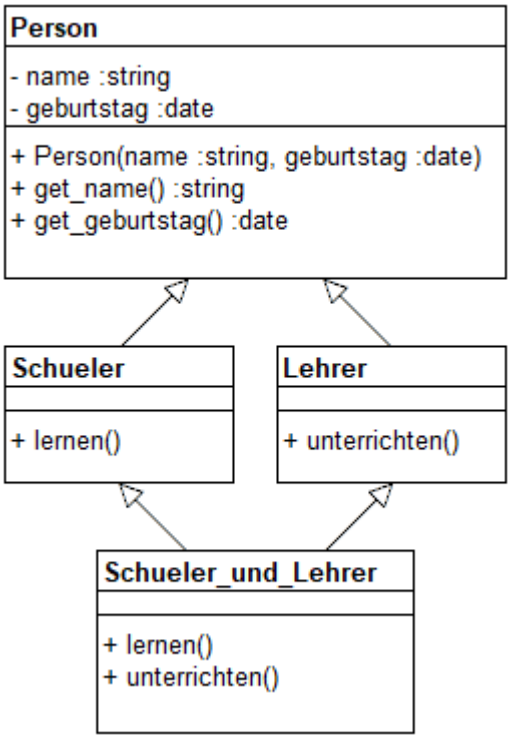
2.4.1.1 Virtuelle Vererbung

In C++ kann das Diamond-Problem mithilfe virtueller Vererbung gelöst werden. Hierbei wird von allen Klassen, die über mehrere Pfade von einer abgeleiteten Klasse beerbt werden, virtuell geerbt. Somit teilen sich dessen abgeleitete Klassen eine gemeinsame Instanz von dieser Klasse.



Beispiel:
Es könnte z.B. eine Klasse 'Person' geben, von der die Klassen 'Schueler' und 'Lehrer' erben. Die Klasse 'Lehrer' fügt die Methode 'unterrichten' hinzu und die Klasse 'Schueler' die Methode 'lernen'. Es gibt aber auch Schüler, die anderen Schülern Nachhilfe geben. Solche Schüler sind also Schüler und Lehrer zugleich. Deshalb wird eine neue Klasse 'Schueler_und_Lehrer' programmiert, die von der Klasse 'Schueler' und von der Klasse 'Lehrer' erbt. Objekte dieser Klasse können sowohl unterrichten, als auch lernen.

Copyright © Paul Raffer 2021



2.5 Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, von der keine Objekte erzeugt werden können. Das ist natürlich nur sinnvoll, wenn von so einer Klasse geerbt wird, damit von einer Unterklasse ein Objekt erzeugt werden kann. In Java werden abstrakte Klassen mit dem abstract-Schlüsselwort gekennzeichnet. In C++ ist eine Klasse, die eine rein virtuelle Methode (virtual method() = 0) enthält, automatisch abstrakt.

Java:

```
abstract class Abstrakte_klasse {
public:
    void virtuelle_methode();
};
```

C++:

```
class Abstrakte_klasse {
public:
    virtual void rein_virtuelle_methode() = 0;
};
```

Folgender Code wäre nicht zulässig, da kein Objekt einer abstrakten Klasse erzeugt werden darf.

```
Abstrakte_klasse* = new Abstrakte_klasse{}; // Error! :)
```

Es muss zuerst von der abstrakten Klasse geerbt werden, um von abgeleiteten Klassen Objekte zu erzeugen. Die rein virtuelle Methode muss von der abgeleiteten Klasse implementiert werden, sonst ist auch die abgeleitete Klasse abstrakt.

```
class Abgeleitete_klasse : public Abstrakte_klasse {
public:
    virtual void rein_virtuelle_methode() override { /*...*/ }
    // ...
};
```

Copyright © Paul Raffer 2021

Nun kann ohne Probleme ein Objekt der abgeleiteten Klasse erzeugt werden.

```
Abstrakte_klasse* = new Abgeleitete_klasse{}; // Funktioniert! :)
```

2.6 Endgültige Klassen

Endgültige Klassen sind Klassen, von denen nicht geerbt werden kann.

```
final class Endgueltige_klasse {
    // ...
};
```

3 Polymorphie

Mithilfe von Polymorphie kann das gleiche Interface für Objekte verschiedener Typen bereitgestellt werden. Ein Bezeichner kann dabei Objekte unterschiedlicher Datentypen annehmen. Polymorphie wird auch Polymorphismus oder Vielgestaltigkeit genannt. Das Gegenteil von Polymorphie ist Monomorphie. Die meisten Codebeispiele in diesem Kapitel sind in C++ geschrieben, da C++ alle der hier genannten Formen der Polymorphie unterstützt.
Es gibt unterschiedliche Arten der Polymorphie, die man unterschiedlich einteilen kann. Eine mögliche Einteilung könnte beispielsweise so aussehen:

	universell unendlich viele Typen eine Implementierung	**ad-hoc** endliche Anzahl an Typen unterschiedliche Implementierungen
dynamisch Laufzeit langsamer	Inklusionspolymorphie/ Vererbungspolymorphie	
statisch Kompilezeit schneller	parametrische Polymorphie	Überladung, Coercion

Polymorphie kann in statische und dynamische Polymorphie eingeteilt werden. Bei der statischen Polymorphie steht schon zur Kompilezeit fest, welche Funktion bzw. Methode zur Laufzeit aufgerufen werden wird. Bei der dynamischen hingegen wird dies erst zu Laufzeit entschieden. Welche Methode aufgerufen wird, ist abhängig vom Typen des Objekts. Diese Einteilung stimmt für die meisten Programmiersprachen, es gibt aber auch Ausnahmen. In dynamischen Programmiersprachen müssen beispielsweise alle Arten der Polymorphie dynamisch sein, da das Programm nicht kompiliert wird. Weiters kann auch zwischen universeller- und Ad-hoc-Polymorphie unterschieden werden.

3.1 Universelle Polymorphie

Mithilfe universeller Polymorphie kann das gleiche Interface für unendlich viele Typen mit nur einer einzigen Implementierung bereitgestellt werden. Diese Typen muss es noch gar nicht geben, sondern können auch erst in der Zukunft definiert werden. Darum wird universelle Polymorphie auch oft “echte” Vielgestaltigkeit genannt.

3.1.1 Inklusionspolymorphie

Inklusionspolymorphie (englisch subtyping) liegt dann vor, wenn das Liskovsche Substitutionsprinzip erfüllt ist.

3.1.1.1 Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip ist erfüllt, wenn jedes Objekt von Typ A (bei Vererbung die Basisklasse) problemlos durch ein Objekt von Typ B (bei Vererbung die abgeleitete Klasse) ersetzt werden kann, ohne dass sich dabei das Verhalten ändert. Ein Objekt von Typ B muss jedoch nicht durch ein Objekt von Typ A ersetzt werden können. Das bedeutet, dass die Schnittstelle von Typ A eine Teilmenge der Schnittstelle von Typ B sein muss.

Copyright © Paul Raffer 2021

3.1.2 Vererbungspolymorphie

In objektorientierten Programmiersprachen wird Inklusionspolymorphie meist durch Vererbung ausgedrückt. Trotzdem sind Inklusionspolymorphie und Vererbungspolymorphie (englisch subclassing) nicht dasselbe, da sich die Vererbungspolymorphie nicht an das Liskovsche Substitutionsprinzip der halten muss. Es kann beispielsweise eine Methode, welche in einer in einer Basisklasse existiert, in der abgeleiteten Klasse entfernt werden. Es sollte aber auch bei Vererbungspolymorphie darauf geachtet werden, dass das Liskovsche Substitutionsprinzip eingehalten wird, auch wenn dies nicht erforderlich ist.

3.1.2.1 Virtuelle Methoden

Eine virtuelle Methode, ist eine Methode bei der zur Kompilezeit noch nicht feststeht, welcher Code ausgeführt wird, wenn sie aufgerufen wird.

Beispiel – Fahrzeuge:

```
class Fahrzeug {
    // ...
    public: virtual void rechts_abbiegen() = 0;
    // ...
};

class Auto {
    // ...
    public: virtual void rechts_abbiegen()
    {
        // Implementierung fuer Autos
    }
    // ...
}

class Fahrrad {
    // ...
    public: virtual void rechts_abbiegen()
    {
        // Implementierung fuer Fahrraeder
    }
    // ...
};

auto main() -> int
{
    Fahrzeug* fahrzeug = new Auto;
    fahrzeug.rechts_abbiegen();
}
```

Eine Alternative zu Vererbungspolymorphie wäre mehrfache Auswahl (z.B. mit dem switch-Statement). Nachteil dabei ist, das der Code für ein Objekt nicht gesammelt an einem Platz ist, sondern über das ganze Programm verteilt. So müsste man immer den ganzen Code durchsuchen, wenn man z.B. einen Fahrzeugtypen hinzufügen möchte. Wenn man allerdings Vererbungspolymorphie verwendet, muss man lediglich eine weiter Unterklasse schreiben.

```
enum Fahrzeugtyp {
    Auto,
    Fahrrad,
};

struct Fahrzeug {
    Fahrzeugtyp typ;
    // ...
```

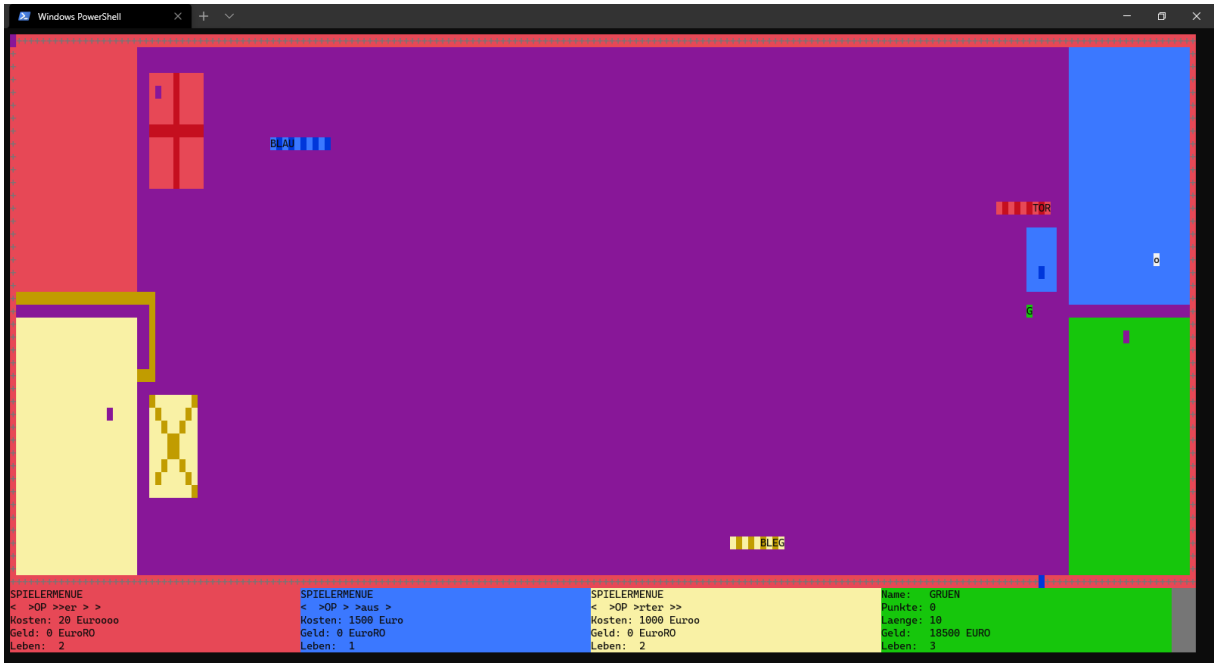
Copyright © Paul Raffer 2021

```
};

void rechts_abbiegen(Fahrzeug* fahrzeug)
{
    switch (fahrzeug->typ) {
        case Auto:
            // Implementierung fuer Autos
            break;
        case Fahrrad:
            // Implementierung fuer Fahrraeder
            break;
        default:
            // Fehler: Ungueltiger Typ!
            break;
    }
}

auto main() -> int
{
    Fahrzeug fahrzeug{Auto};
    rechts_abbiegen(&fahrzeug);
}
```

Beispiel – Snake:



Als ich in der 1. Klasse mein Multiplayer-Snake-Spiel mit ein paar Sonderregeln programmierte, kannte ich Klassen noch nicht. Man kann in dem Spiel unterschiedliche Arten von Gebäuden bauen (Kanonen, Mauern, Geldlager, Krankenhäuser, ...) und unterschiedliche Arten von Punkten fressen (normale Punkte, Geld, Leben, ...). Dies wäre der Ideale Anwendungsfall für Vererbungspolymorphie gewesen. Da ich diese aber noch nicht kannte, musste ich mit mehrfacher Auswahl bzw. doppeltem Code arbeiten. Dies führte zu sehr unübersichtlichem, redundanten und schwer erweiterbarem Code. Jedes Mal, wenn man einen Gebäudetyp hinzufügen möchte, muss man den Code an unterschiedlichen Stellen im gesamten File bearbeiten.
Viel Code wurde doppelt für unterschiedliche Gebäude geschrieben:

```
// ...
```

Copyright © Paul Raffer 2021

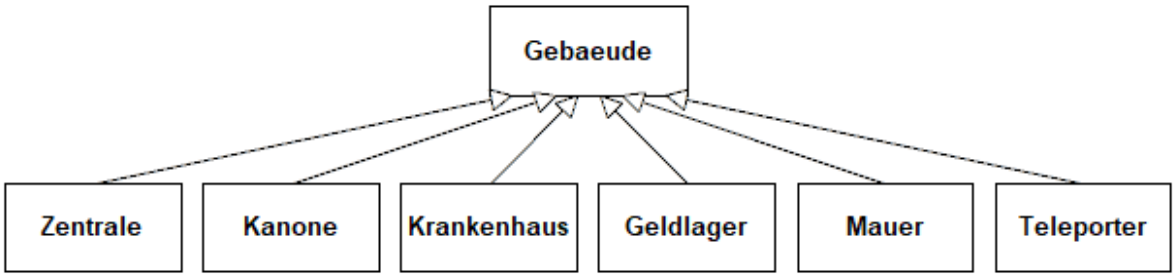
```
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.zentrale, s, v);
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.kanone, s, v);
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.krankenhaus, s, v);
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.geldlager, s, v);
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.mauer, s, v);
gebaeude_verschieben(spielfeld, spieler, spieler.at(s).gebaeude.teleporter, s, v);

// ...

gebaeude_gameover(spielfeld, spieler, spieler.at(sp).gebaeude.kanone, s, sp);
gebaeude_gameover(spielfeld, spieler, spieler.at(sp).gebaeude.krankenhaus, s, sp);
gebaeude_gameover(spielfeld, spieler, spieler.at(sp).gebaeude.geldlager, s, sp);
gebaeude_gameover(spielfeld, spieler, spieler.at(sp).gebaeude.mauer, s, sp);
gebaeude_gameover(spielfeld, spieler, spieler.at(sp).gebaeude.teleporter, s, sp);

// ...
```

Codeteile wie diese findet man über den gesamten Code verteilt. Abgesehen davon, dass diese Funktionsnamen nicht sehr aussagekräftig sind, wird das Programm durch den doppelten Code auch länger als notwendig. Das gesamte Programm ist 1900 Zeilen lang. Wenn man es mit Vererbungspolymorphie neu programmieren würde, könnte man sich mindestens die Hälfte des Codes sparen. Ein weiteres Problem ist, dass leicht auf ein Gebäude vergessen werden kann. Die führt zu mehr Fehlern, die im schlimmsten Fall gar nicht auffallen, sondern sich erst später bemerkbar machen.
Mit Vererbungspolymorphie könnte eine Vererbungshierarchie beispielsweise so aussehen:



Die Funktionsaufrufe würden sich dadurch vereinfachen, da zur Laufzeit automatisch die richtige virtuelle Methode, abhängig vom Gebäudetyp, ausgewählt wird.

```
// ...

for (auto & g : spieler.at(s).gebaeude) {
    // Welchen Typ g hat, und somit auch welche Methode 'verschieben'
    // aufgerufen wird, wird erst zur Laufzeit bestimmt.
    g.verschieben(spielfeld, spieler, s, v);
}

// ...

for (auto & g : spieler.at(sp).gebaeude) {
    g.gameover(spielfeld, spieler, s, sp);
}

// ...
```

Der Vollständige Code ist auf GitHub zu finden: `https://github.com/PaulRaffer/Snake`

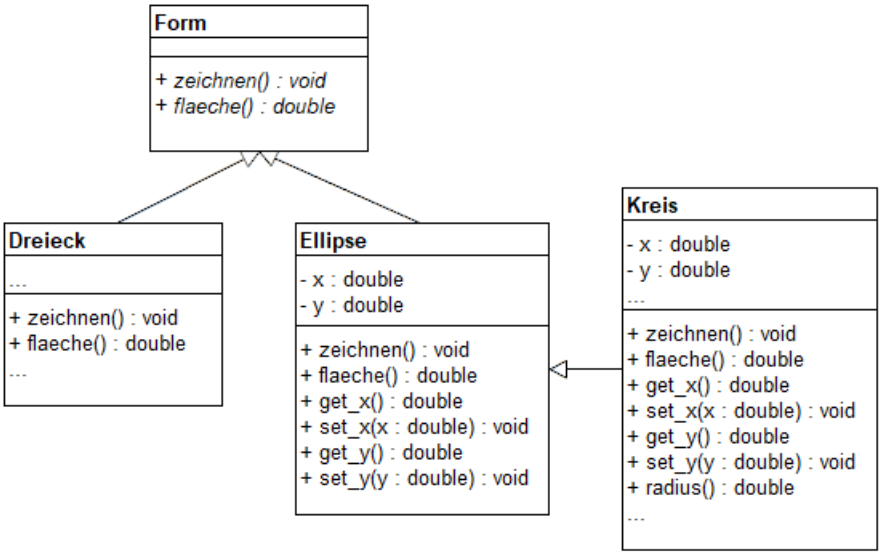
3.1.2.2 Kreis-Ellipse-Problem

Das Kreis-Ellipse-Problem (bzw. Quadrat-Rechteck-Problem) ist ein Problem in der objektorientierten Programmierung.

Copyright © Paul Raffer 2021

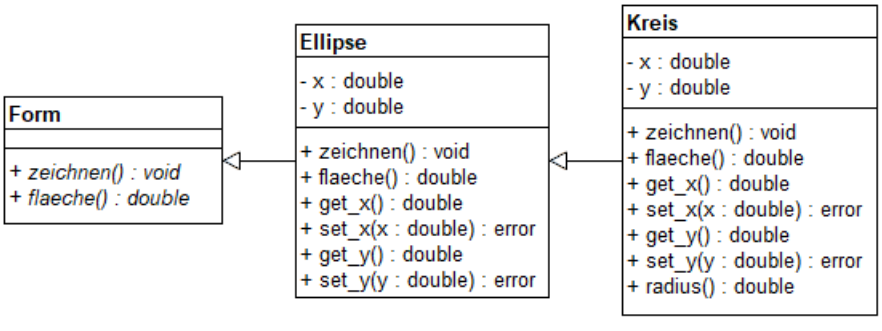
Problem:
Für eine Grafiksoftware existiert bereits die (abstrakte) Basisklasse 'Form', welche die Methoden 'zeichnen' und 'flaeche' enthält. Einige konkrete Klassen, wie z.B. Dreieck, haben bereits von 'Form' geerbt. Nun sollen auch Klassen für Ellipsen und Kreise geschrieben werden, doch das ist komplizierter als es auf den ersten Blick scheint.

Die intuitive Lösung wäre, dass die Klasse 'Kreis' von der Klasse 'Ellipse' erbt, welche wiederum von der Klasse 'Form' erbt, da jeder Kreis auch eine Ellipse ist und somit eine "ist-ein"-Beziehung vorliegt. Zusätzlich zu den beiden Methoden 'zeichnen' und 'flaeche' implementiert die Klasse 'Ellipse' auch noch die Setter 'set_x' und 'set_y' zum skalieren der Ellipse. Diese beiden Methoden werden auch an die Klasse 'Kreis' weitergegeben, obwohl man bei einem Kreis nicht beide Dimensionen unabhängig voneinander skalieren kann. Wenn ein skalieren der einen Dimension ein automatisches skalieren der anderen Dimension bewirken würde, wäre das Liskovsches Substitutionsprinzip nicht erfüllt, da sich die Setter 'set_x' und 'set_y' der Klasse 'Kreis' nicht gleich verhalten würden, wie die der Klasse 'Ellipse'!



Lösungsvorschläge:

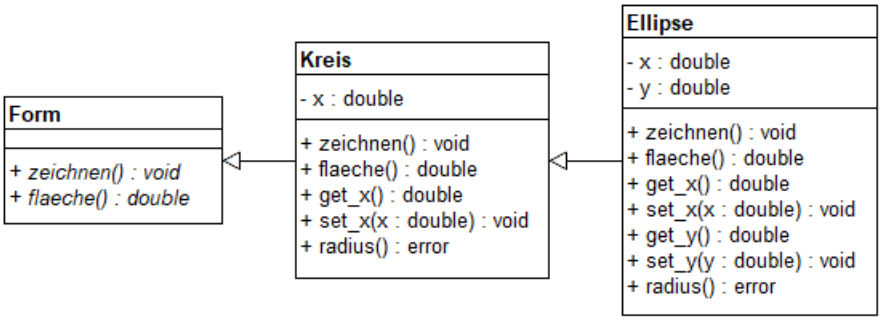
- **Fehler bei Größenänderung:** Bereits die Basisklasse 'Ellipse' legt fest, dass die Methoden zum ändern der Größe scheitern können. Dies wird z.B. durch das zurückgeben eines Fehlercodes signalisiert.



  Der Nachteil dieser Lösung ist, dass das Problem bereits auf Ebene der Basisklasse gelöst werden muss, und der Entwickler der Basisklasse auch alle abgeleiteten Klassen kennen muss.

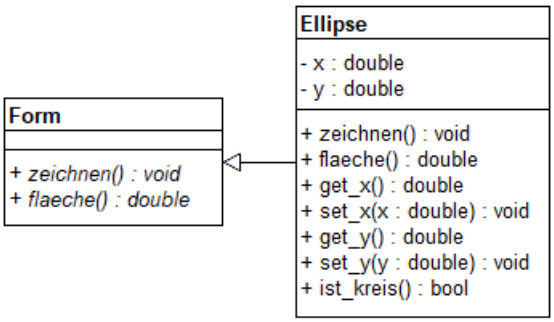
- **Ellipse erbt von Kreis:** Bei dieser Lösung enthält die Klasse 'Kreis' einen Setter und die entsprechende Eigenschaft. Die Klasse 'Ellipse' erbt bin der Klasse 'Kreis' und ergänzt den zweiten Setter und die zweite Eigenschaft für die zweite Dimension.

Copyright © Paul Raffer 2021



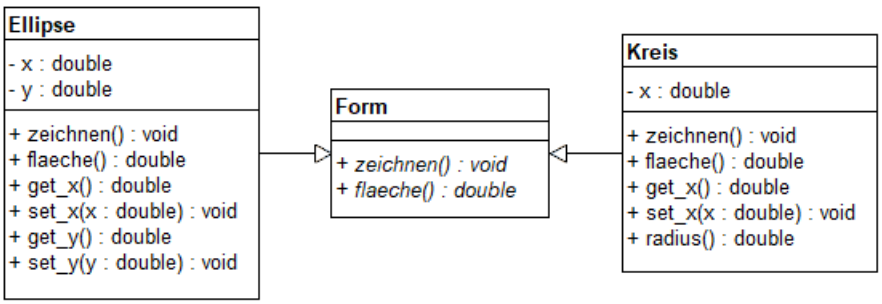
Das Problem hierbei ist, dass die Aussage "Jede Ellipse ist ein Kreis." nicht richtig ist, und keine "ist-ein"-Beziehung darstellt. Es wird das gesamte Interface der Klasse 'Kreis' in die Klasse 'Ellipse' vererbt, auch Eigenschaften und Methoden, die nur ein Kreis haben sollte, wie z.B. 'radius'. Diese müssten dann entweder weggelassen werden oder ebenfalls einen Fehlercode zurückgeben, falls es sich um einen Ellipse handelt.

- **Keine Klasse Kreis:** Eine weitere mögliche Lösung ist, dass es gar keine Klasse gibt, sondern die Klasse Ellipse eine Methode 'ist_kreis' bereitstellt.



Der Nachteil davon ist, dass der Kreis kein eigenes Interface (z.B. 'radius') bereitstellen kann.

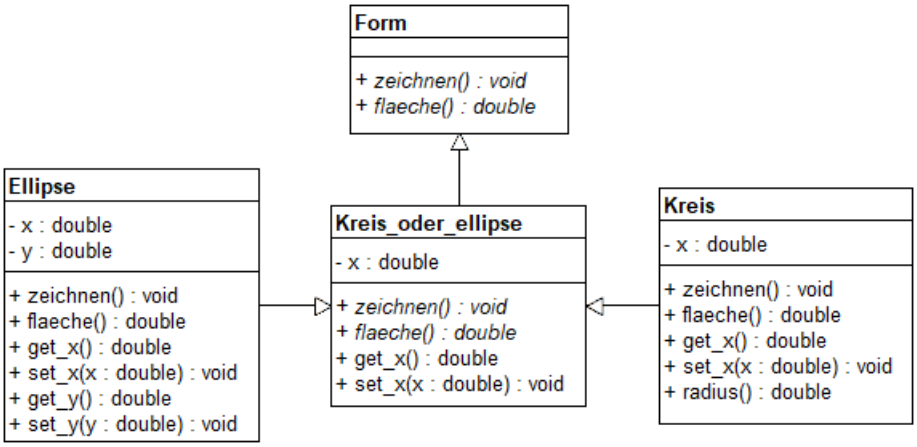
- **Keine Vererbungsbeziehung zwischen Ellipse und Kreis:** Es könnten auch beide Klassen direkt von der gemeinsamen Basisklasse 'Form' erben.



Bei diese Lösung müssten allerdings die Gemeinsamkeiten der beiden Klassen doppelt implementiert werden.

- **Einführen neuer Basisklasse:** Man könnte auch eine weitere Klasse ('Ellipse_oder_kreis') einführen, welche in der Vererbungshierarchie zwischen der Basisklasse 'Form' und den beiden konkreten Formen 'Kreis' und 'Ellipse' steht. Diese Klasse könnte dann den Code enthalten, den die Klassen 'Kreis' und 'Ellipse' gemeinsam haben.

Copyright © Paul Raffer 2021



Der Nachteil diese Lösung ist, dass dadurch die Tiefe der Vererbungshierarchie erhöht werden würde, was den Code unflexibler machen würde.

Keiner der oben genannten Lösungsvorschläge ist ideal. Jeder hat seine Vor- und Nachteile.
Daraus folgt, das nicht jede "ist-ein"-Beziehung durch öffentliche Vererbung dargestellt werden sollte, auch wenn öffentliche Vererbung meistens "ist-ein" bedeuten sollte!

3.1.3 Parametrische Polymorphie

Parametrische Polymorphie ist vor allem in der funktionalen Programmierung weit verbreitet. Diese Art der Polymorphie gehört zur statischen Polymorphie, da bereits zur Kompilezeit feststeht, welche Funktion aufgerufen werden wird. TODO

3.1.3.1 Einfacher parametrischen Polymorphismus

3.1.3.2 Beschränkter parametrischen Polymorphismus

Im Gegensatz zum einfachen parametrischen Polymorphismus ist der beschränkte parametrische Polymorphismus typensicher.

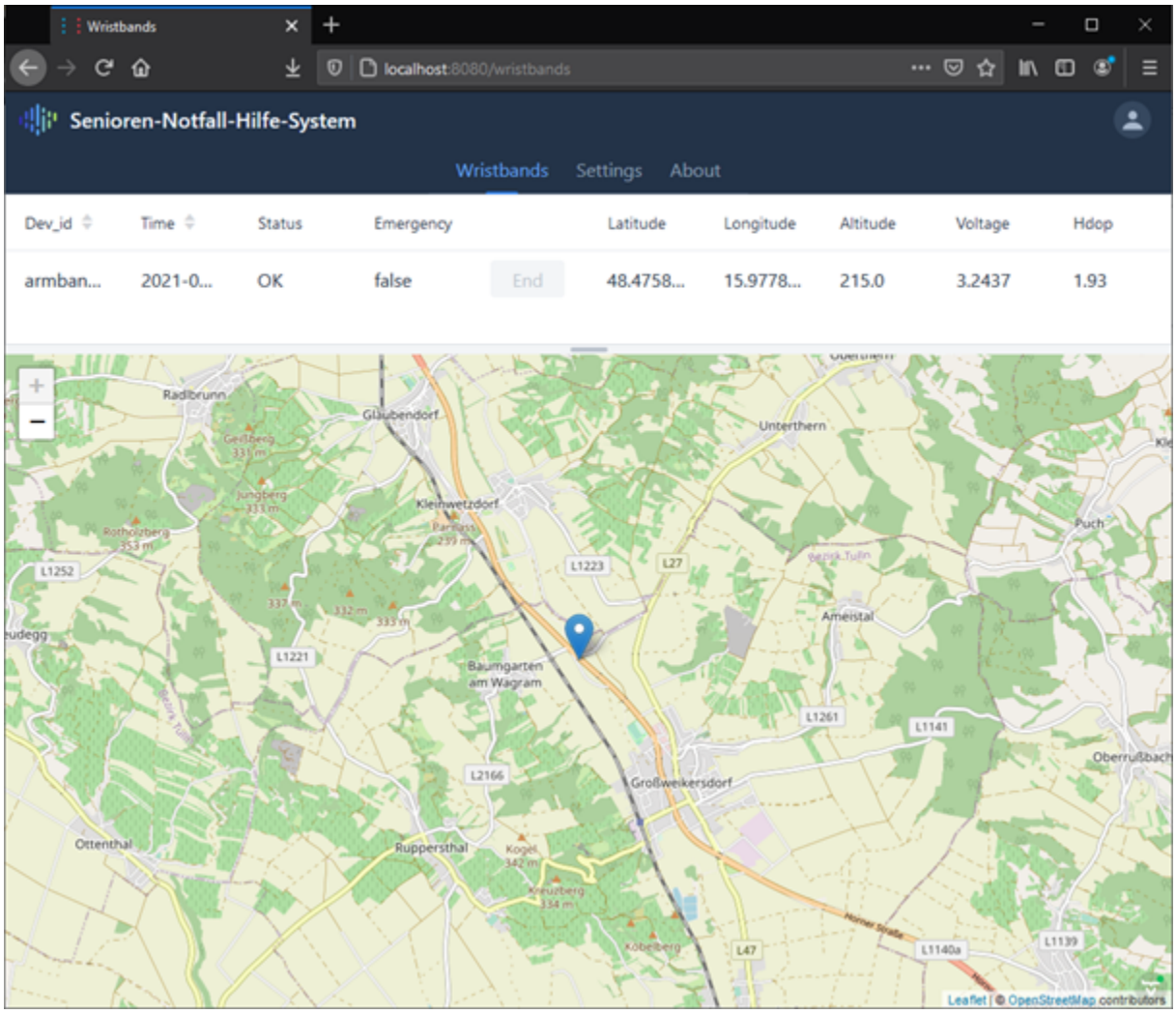
3.1.3.3 CRTP

Das CRTP (Curiously recurring template pattern) kombiniert parametrische Polymorphie mit Vererbung. Hierbei erbt eine Klasse von einer generischen Klasse, an die die abgeleitete Klasse selbst als Typ-Parameter übergeben wird.

```
template <typename T>
class Basisklasse {
    //...
};

class Abgeleitete_klasse : public Basisklasse<Abgeleitete_klasse> {
    //...
};
```

3.1.3.3.1 Beispiel – Diplomarbeit:



Im Rahmen der Diplomarbeit wurde ein Senioren-Notfall-Hilfe-System entwickelt. Ein Armband nimmt Messdaten auf und sendet diese mittels LoRaWAN an ein Gateway. Dieses leitet die Daten an einen Server weiter, welcher diese darstellt. Die Position des Armbands wird auf einer Karte und zusätzlich in einer Liste (falls mehrere Armbänder registriert sind) angezeigt. Natürlich müssen beide Ansichten regelmäßig aktualisiert werden, da sich die Position oder der Status des Armbandes ändern können. Dazu wurde eine generische Basisklasse (UpdateView) geschrieben, welche die Logik zum aktualisieren der Daten enthält. Jede zweite Sekunde wird die Ansicht in einem Hintergrundthread aktualisiert. Dies geschieht durch das Aufrufen der Funktion 'updateFunction'. Die Funktion wird von der Unterklasse als Lambda an eine Setter-Funktion höherer Ordnung (setUpdateFunction) übergeben. Da es in Java keine freien Funktionen gibt muss ein Interface (UpdateFunction) mit einer einzigen Methode (update) erstellt werden.

```
// Derive from this Class using the
// Curiously recurring template pattern!
// https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
// e.g. public class MyView extends UpdateView<MyView> { ... }
public class UpdateView<View> extends VerticalLayout {
    private UpdateThread updateThread;       // Background thread, which updates the view
    private UpdateFunction updateFunction;   // Function, which updates the view
    private long pauseMillis = 2000;         // Update every 2 s

    public interface UpdateFunction<View> { // Lambda interface
        void update(View view);             // Update object of subclass
    }
    protected void setUpdateFunction(UpdateFunction<View> f)
    {
```

```
        updateFunction = f;
    }
    protected void setUpdatePause(long millis)
    {
        pauseMillis = millis;
    }

    @Override
    protected void onAttach(AttachEvent e)
    {
        updateThread = new UpdateThread(e.getUI(), this); // this wird an Thread Ãijberge
        updateThread.start(); // Start new thread
    }

    @Override
    protected void onDetach(DetachEvent e)
    {
        updateThread.interrupt();  // Stop thread
        updateThread = null;
    }

    private class UpdateThread extends Thread {
        private final UI ui;
        private final UpdateView view;

        public UpdateThread(
                final UI ui, final UpdateView view)
        {
            this.ui = ui;
            this.view = view;
        }

        @Override
        public void run()
        {
            try {
                while (true) {
                    Thread.sleep(view.pauseMillis); // Wait
                    ui.access(() -> { // Lock session

                        // 'view' wird von Typ 'UpdateView' (diese Klasse == Basisklasse
                        // zu 'View' (Typ-Parameter == abgeleitete Klasse) umgewandelt!
                        // Dies ist ok, da wir das CRTP verwenden!
                        updateFunction.update(view);

                    });
                }
            }
            catch (InterruptedException e) {
                // We don't care if sleep was interrupted.
            }
        }
    }
}
```

Um eine Ansicht zu erstellen, die automatisch mit neuen Daten aktualisiert wird, muss von der Klasse 'UpdateView' geerbt werden. Dabei wird das CRTP angewandt. Daraus folgt, dass ein Objekt der Klasse MyView, wobei MyView von UpdateView¡MyView¿ aberbt, auch ein Objekt des Typen UpdateView¡MyView¿ sein muss. Umgekehrt ist dies streng genommen eigentlich nicht der Fall, da aber das CRTP verwendet wird und die Funk-

Copyright © Paul Raffer 2021

tion, welche die Ansicht aktualisiert in MyView definiert wird und deshalb alle Eigenschafen und Methoden der Unterklasse (MyView selbst) kennt, ist auch eine Typumwandlung von UpdateView nach MyView problemlos (abgesehen von ein paar Warnungen) möglich.

```
public class MapView extends UpdateView<MapView> {
    private final LeafletMap map;

    public MapView()
    {
        map = new LeafletMap();

        //...

        map.addMarkersAndZoom( // Add all known markers to the map
                MQTTService.getInstance().getAll());

        setUpdateFunction(view -> {
            view.map.removeAllMarkers();
            MQTTService.getInstance().getAll().forEach(view.map::addMarker);
        });
    }
}

public class ListView extends UpdateView<ListView> {

    private final Grid<TTNUplinkMessage<Wristband>> grid =
            new Grid<>((Class<TTNUplinkMessage<Wristband>>)(Class)
                    TTNUplinkMessage.class);

    public ListView()
    {
        //...

        setUpdateFunction(view ->
                grid.getDataProvider().refreshAll());
    }
}
```

3.2 Ad-hoc-Polymorphie

Mit Ad-hoc-Polymorphie kann die gleiche Schnittstelle, im Gegensatz zu universellen Polymorphie, nur für eine begrenzte Anzahl an bestimmten Typen bereitgestellt werden. Für jeden Typ gibt es eine eigene Implementierung

3.2.1 Coercion

Coercion ist eine implizite Typumwandlung vom Compiler. Diese Art der Polymorphie ist statisch, da schon zur Kompilezeit feststeht, welche Funktion bzw. Methode aufgerufen wird. Implizite Typumwandlungen können entweder zwischen zwei Variablen von eingebauten Typen, zwischen Objekten von selbst definierten Typen oder zwischen einer Variable eines eingebauten Typen und einem Objekt stattfinden.

3.2.1.1 Coercion von Variablen eingebauter Typen

```
auto main() -> int
{
    // Implizite Umwandlung von int{5} zu double{5.0}
    // Output: 8.2
    std::cout << 5 + 3.2;
}
```

Copyright © Paul Raffer 2021

Ohne Coercion würde diese Zeile einen Fehler verursachen! Diese Art der Coercion ist die einzige Art der Polymorphie die von der Programmiersprache C unterstützt wird!

3.2.1.2 Coercion von Objekten

Konvertierungskonstruktor (C++):
In C++ ist ein Konvertierungskonstruktor ein nicht expliziter Konstruktor der einen einzigen Parameter akzeptiert. Solch ein Konstruktor wandelt Objekte bzw. Variablen des Parametertyps in Objekte des Klassentyps um. Der Konstruktor darf nicht explizit sein, da sonst nur explizite Typumwandlungen möglich sind.

```
class Bruch {
    int zaehler_;
    int nenner_;
    // ...
};

// Konvertierungskonstruktor (int to Bruch)
Bruch::Bruch(int zaehler = 0, int nenner = 1)
    : zaehler_{zaehler}, nenner_{nenner} {}

void print(Bruch const & bruch)
{
    std::cout << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}

auto main() -> int
{
    // implizite Cast von int{42} zu Bruch{42, 1}
    // Output: (42/1)
    print(42);
}
```

Konvertierungsoperatoren (C++):
In C++ werden Konvertierungsoperatoren verwendet, um ein Objekt in ein Objekt oder eine Variable eines anderen Typen umzuwandeln.

```
Bruch::operator double() // Konvertierungsoperator (Bruch zu double)
{
    return double{this->zaehler_} / double{this->nenner_};
}

auto main() -> int
{
    // Implizite Umwandlung von Bruch{1, 4} zu double{0.25}
    // Output: 4.5
    std::cout << Bruch{1, 4} + 4.25;
}
```

3.2.2 Überladung

Bei der Funktionsüberladung können verschiedene Funktionen gleich heißen und sich nur durch die Typen oder die Anzahl ihrer Parameter unterscheiden. Funktionen, die das gleiche Verhalten haben, sollten auch den gleichen Namen haben.

Beispiel – Ausgabe: Es sollen zwei Funktionen zur Ausgabe von einem C-String (`char []`) bzw. einer ganzen Zahl (`int`) geschrieben werden. Da die Funktionen beide das gleiche tun, sollten sie auch den gleichen Namen haben.

Copyright © Paul Raffer 2021

```c
void print_str(char const * str)
{
    printf("%s", str);
}

void print_i(int i)
{
    printf("%d", i);
}

print_str("Hallo!\n");
print_i(42);
```

C-Lösung:

```c
void print(char const * str)
{
    printf("%s", str);
}

void print(int i)
{
    printf("%d", i);
}

print("Hallo!\n");
print(42);
```

C++-Lösung:

Beispiel – Reset:

```c
void reset(int& i)
{
    i = 0;
}

void reset(std::vector<int>& vec)
{
    for (int& i : vec) {
        reset(i);
    }
}
```

3.2.2.1 Operatorüberladung

Manche Programmiersprachen (darunter C++) ermöglichen das Überladen von Operatoren für eigene Typen. Die Operatorüberladung ist ein Spezialfall der Funktionsüberladung, da Operatoren auch nichts anderes als Funktionen mit einem oder zwei Parametern sind.
Es können beispielsweise die arithmetischen Operatoren für eine eigene Bruchklasse definiert werden.

3.2.3 C++ Template-Spezialisierung

C++ Templates gehören eigentlich zur Kategorie der parametrischen Polymorphie, jedoch kann eine generische Funktion in C++ auch für bestimmte neu implementiert werden implementiert werden. Dies ist auch eine Art der Ad-hoc-Polymorphie.

Copyright © Paul Raffer 2021

3.3 Kombination verschiedener Arten der Polymorphie

Verschiedene Arten der Polymorphie können meist sehr gut in einem Projekt kombiniert werden.

Beispiel – Bruchklasse:

In meiner Bruchklassen (`https://github.com/PaulRaffer/raffer_cpplib-fraction/blob/cpp14/fraction.hpp`) habe ich z.B. die drei wichtigsten Arten der statischen Polymorphie (parametrische Polymorphie, Coercion und Überladung) verwendet.
Parametrische Polymorphie wird verwendet, um beliebige Zähler bzw. Nenner beliebiger Typen zuzulassen. So können beispielsweise Doppelbrüche (fraction <fraction<int>, fraction <int>>) oder Brüche mit komplexen Zahlen als Zähler bzw. Nenner (fraction <complex<double>, complex<double>>) erstellt werden.
Des weiteren hat das Klassen-Template einen Konvertierungskonstruktor, der Objekte des Zähler-Typen implizit in Brüche umwandeln kann.
Mit Brüchen kann auch gerechnet werden, da die arithmetischen Operatoren überladen wurden.

Copyright © Paul Raffer 2021

4 Quellen

- FSST-Mitschrift der 2. und 3. Klasse
- https://www.youtube.com/watch?v=7EmboKQH8lM
- https://www.ics.uci.edu/~jajones/INF102-S18/readings/05_stratchey_1967.pdf
- https://invidious.snopyta.org/watch?v=7PfNo-FMIf0
- https://invidious.snopyta.org/watch?v=uTxRF5ag27A
- https://soundcloud.com/lambda-cast
- https://de.wikipedia.org/wiki/Polymorphie_(Programmierung)
- https://lec.inf.ethz.ch/ifmp/2019/slides/lecture14.handout.pdf
- https://courses.cs.washington.edu/courses/cse331/11wi/lectures/lect06-subtyping.pdf
- https://de.wikipedia.org/wiki/Vererbung_(Programmierung)
- https://wiki.haskell.org/Polymorphism
- http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf
- C++ Templates: The Complete Guide
- C++: Das umfassende Handbuch
- Grundkurs C++
- C++: Die Sprache der Objekte
- Exceptional C++: 47 engineering puzzles, programming problems, and solutions
- Effektiv C++. 50 Specific Ways to Improve Your Programs and Designs
- https://invidious.snopyta.org/watch?v=HddFGPTAmtU
- https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html
- https://de.wikipedia.org/wiki/Diamond-Problem
- https://stackoverflow.com/questions/66983156/is-coercion-static-or-dynamic-polymorphism

Copyright © Paul Raffer 2021