Lektion 16

Mehrfachvererbung

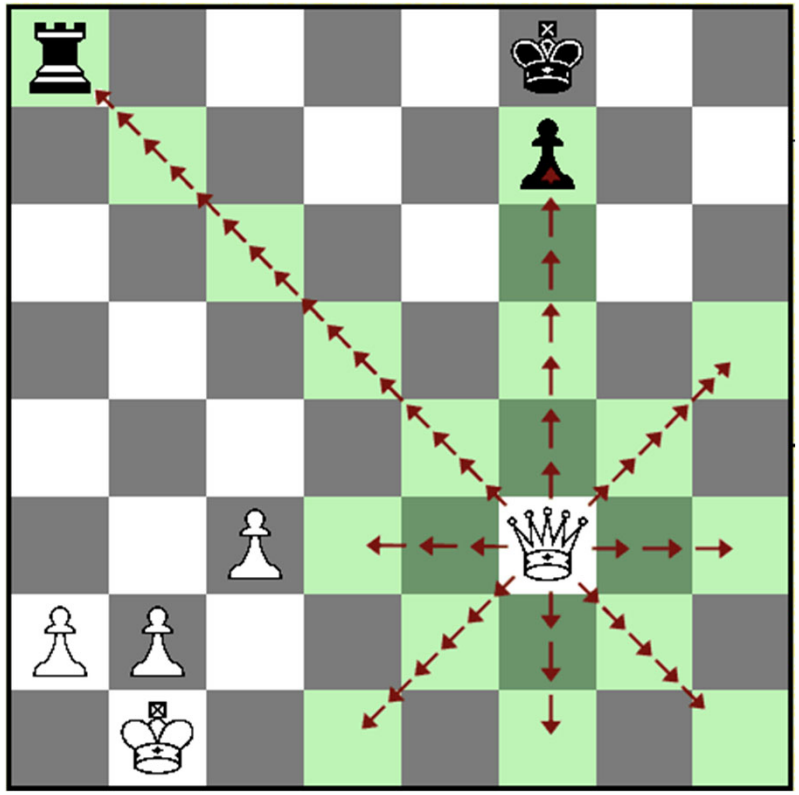
Interfaces

Komposition

Interfaces

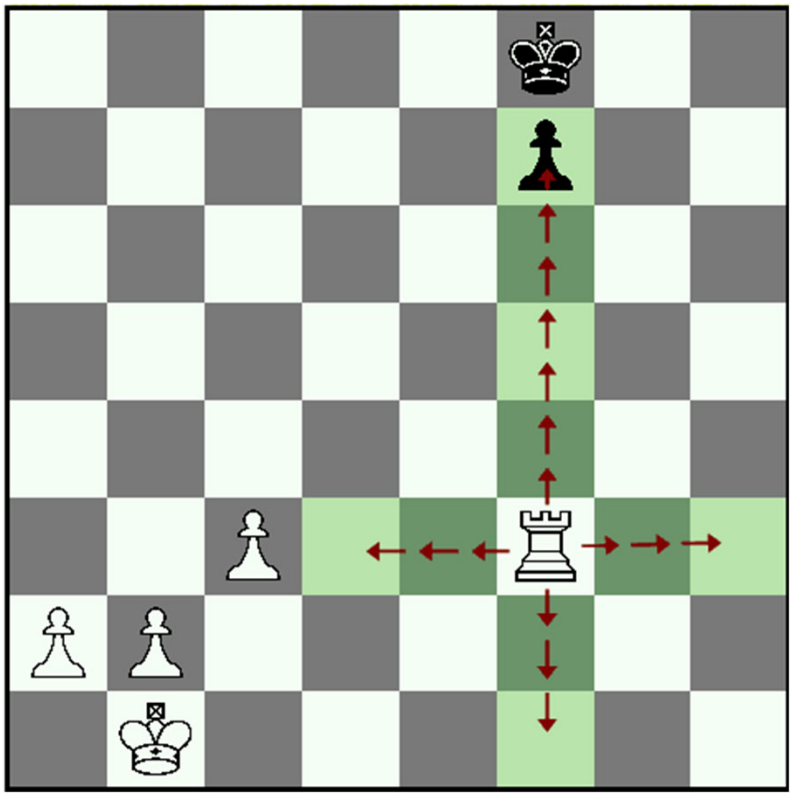
Manchmal ist es wünschenswert, das Verhalten von mehreren Klassen zu erben…

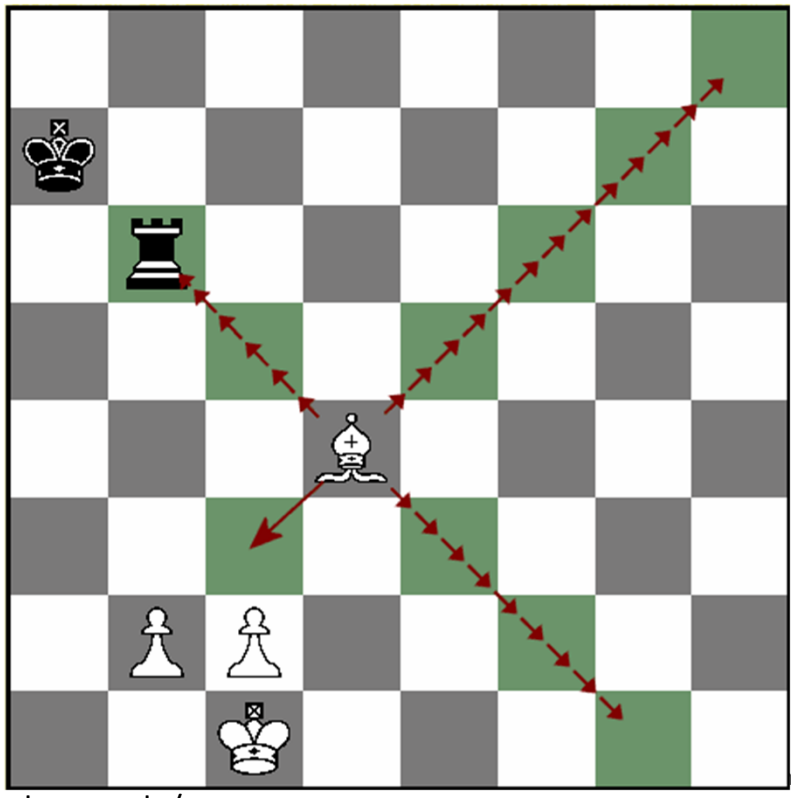
Nehmen wir an, wir entwickeln ein Schachspiel und wollen anzeigen, auf welche Felder sich eine Dame bewegen kann.



Quelle: http://wiki-schacharena.de/

Nehmen wir weiterhin an, wir haben dieses Verhalten bereits für einen Turm und einen Läufer implementiert.





Heinzl

Quelle: http://wiki-schacharena.de/

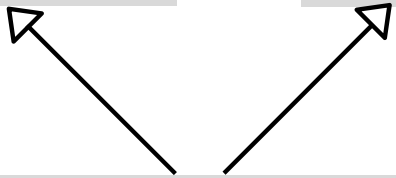
Das Verhalten der Dame setzt sich zusammen aus dem Verhalten des Läufers und dem Verhalten des Turms.

Die Dame könnte also das Verhalten des Läufers **und** des Turms erben.

Ist eine Dame ein Läufer?
Ist eine Dame ein Turm?

```
public class Laeufer
{
  public Brett gibErlaubteFelder()
  {
    …
  }
}
```

```
public class Turm
{
  public Brett gibErlaubteFelder()
  {
    …
  }
}
```



```
public class Dame extends Laeufer, Turm
{
  public Brett gibErlaubteFelder()
  {
    …
  }
}
```

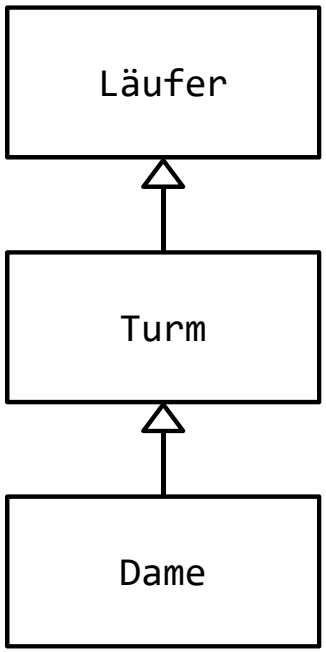
Die Methode heißt in beiden Oberklassen gleich. Welche wird vererbt?

Mehrfachvererbung bringt einige Probleme mit sich:

- Methoden können die gleiche Signatur haben
- Methoden können die gleiche Signatur haben, aber verschiedene Rückgabetypen
- Attribute können mehrfach vorkommen
- Attribute können mehrfach vorkommen und unterschiedliche Datentypen haben.
- …

Daher hat man sich entschieden in Java keine Mehrfachvererbung zu erlauben.

Was ist, wenn die Dame dennoch Läufer und Turm sein soll?



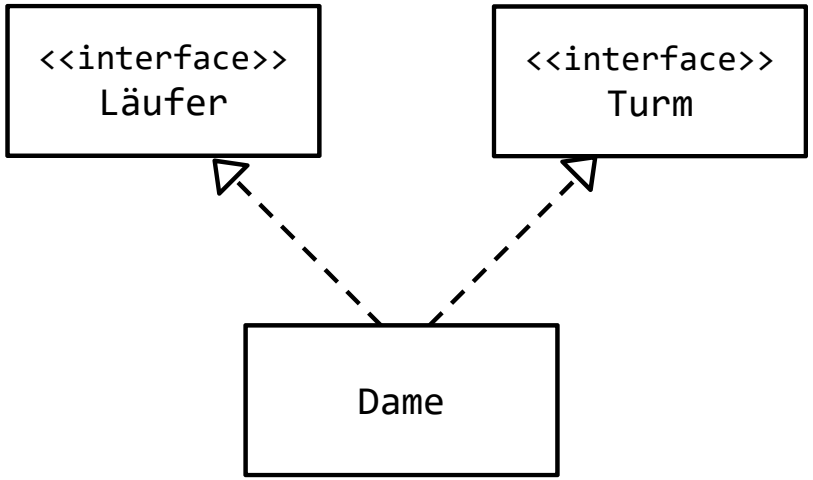
Dame ist **ein** Turm.
Dame ist **ein** Läufer.

Aber:
Turm ist **kein** Läufer
➢ keine gute Lösung

Weil sich ein solches Problem nicht gut abbilden lässt, wurden Interfaces (Schnittstellen) eingeführt.

Eine Schnittstelle spezifiziert nur,
WAS eine Klasse mind. leisten muss.

WIE die Implementierung erfolgt, gibt eine Schnittstelle nicht an.



Dame ist **ein** Turm.
Dame ist **ein** Läufer.

Die Dame muss jetzt mind. das können, **WAS** ein Läufer und ein Turm kann.

WIE die Dame das umsetzt, ist ihre Sache.

Wie deklariert man ein Interface?

```
public interface Laeufer
{
  public Brett gibErlaubteFelder();
}
```

```
public interface Turm
{
  public Brett gibErlaubteFelder();
}
```

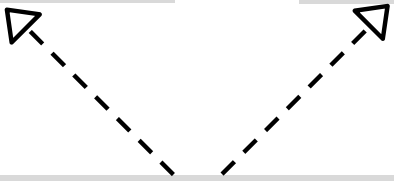
Anstelle des Schlüsselwortes **class** wird das Schlüsselwort **interface** zur Definition einer Schnittstelle verwendet.

Da eine Schnittstelle nur spezifiziert, **WAS** eine Klasse mind. können soll und nicht **WIE** sie es umsetzt,
enthalten die Methoden im Interface **keinen Methodenrumpf**.

Alle Methoden im Interface sind automatisch **abstrakt**.

```
public interface Laeufer
{
  public Brett gibErlaubteFelder();
}
```

```
public interface Turm
{
  public Brett gibErlaubteFelder();
}
```



```
public class Dame implements Laeufer, Turm
{
  public Brett gibErlaubteFelder()
  {
    …
  }
}
```

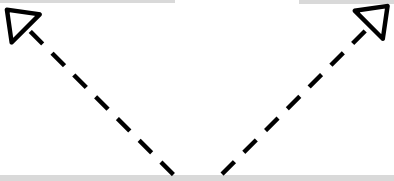
Durch das Schlüsselwort **implements** wird angegeben, dass eine Klasse die Spezifikationsvorgaben eines Interfaces erfüllt.

Dazu muss die Klasse die Methoden des Interfaces implementieren (wie bei einer abstrakten Klasse).

Man sagt dann: Die Klasse **implementiert** das Interface.

```
public interface Laeufer
{
  public Brett gibErlaubteFelder();
}
```

```
public interface Turm
{
  public Brett gibErlaubteFelder();
}
```



```
public class Dame implements Laeufer, Turm
{
  public Brett gibErlaubteFelder()
  {
    …
  }
}
```

Die Klasse Dame implementiert die Interfaces Laeufer und Turm.

Daher muss die Klasse Dame die Methode gibErlaubteFelder() implementieren.

Jetzt wird die Methode gibErlaubteFelder() immer noch zweimal „geerbt“.
Ist das ein Problem?

Man könnte in Versuchung kommen im Interface
dem Läufer eine Position auf dem Brett zuzuweisen.

```
public interface Laeufer
{
  int x = 1;
  int y = 3;
  public Brett gibErlaubteFelder();
}
```

```
public interface Laeufer
{
  public static final int x = 1;
  public static final int y = 3;
  public Brett gibErlaubteFelder();
}
```

In einem Interface können Konstanten deklariert werden.
Jede Deklaration hat automatisch die Modifier static und final.

Die Deklaration ist also nicht sinnvoll, da die Position nicht verändert werden kann und nicht pro Objekt zur Verfügung steht.

- I.d.R. benutzt man keine Attribute in einem Interface.

Zusammenfassend kann man sagen:

Mehrfachvererbung auf Spezifikationsebene
ist in Java möglich.

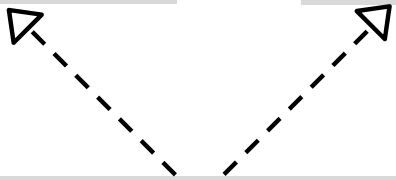
Mehrfachvererbung auf Implementierungsebene
ist in Java nicht möglich.

Beleuchten wir das Beispiel noch etwas näher:

Bisher haben wir die Interfaces Laeufer und Turm und eine angefangene Implementierung für die Dame.

```
public interface Laeufer
{
  public Brett gibErlaubteFelder();
}
```

```
public interface Turm
{
  public Brett gibErlaubteFelder();
}
```



```
public class Dame implements Laeufer, Turm
{
  public Brett gibErlaubteFelder()
  {
    //alle waagrechten, senkrechten und diagonalen Felder
  }
}
```

Jede Figur auf dem Brett benötigt zusätzlich ihre Position, damit man die erlaubten Felder für eine Bewegung ermitteln kann.

Schauen wir uns das für den Läufer an.

```
public interface Laeufer
{
  public Brett gibErlaubteFelder();
}
```

Bisher gibt es nur das Laeufer-Interface.

```
public class LaeuferImpl implements Laeufer {
  int x;
  int y;

  //getter und setter
  …
  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

Eine mögliche Implementierung.

Für den Turm ist es ähnlich:

```
public interface Turm
{
  public Brett gibErlaubteFelder();
}
```

```
public class TurmImpl implements Turm {
  int x;
  int y;

  //getter und setter
  …
  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

Eine mögliche Implementierung.

```
public class TurmImpl implements Turm {
  int x;
  int y;

  //getter und setter
  …
  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

```
public class LaeuferImpl implements Laeufer {
  int x;
  int y;

  //getter und setter
  …
  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

Doppelter Code.

Den können wir in eine gemeinsame Oberklasse auslagern.

```
public class Figur {
  int x;
  int y;

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

```
public class LaeuferImpl extends Figur
  implements Laeufer {

  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

```
public class TurmImpl extends Figur
  implements Turm {

  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

```
public class Figur {
  int x;
  int y;

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

```
public class LaeuferImpl extends Figur
  implements Laeufer {

  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

```
public class TurmImpl extends Figur
  implements Turm {

  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

Wenn jede Figur auf dem Schachbrett erlaubte Felder ausgeben können soll, sollte die Methode in die gemeinsame Oberklasse ausgelagert werden.

```
public abstract class Figur {
  int x;
  int y;

  public abstract Brett gibErlaubteFelder();

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

```
public class LaeuferImpl extends Figur
  implements Laeufer {

  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

```
public class TurmImpl extends Figur
  implements Turm {

  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

Die Interfaces Turm und Laeufer müssen die Methode gibErlaubteFelder weiterhin beinhalten, da ansonsten folgender Zugriff nicht möglich ist:

```
Turm t = new TurmImpl();
t.gibErlaubteFelder();
```

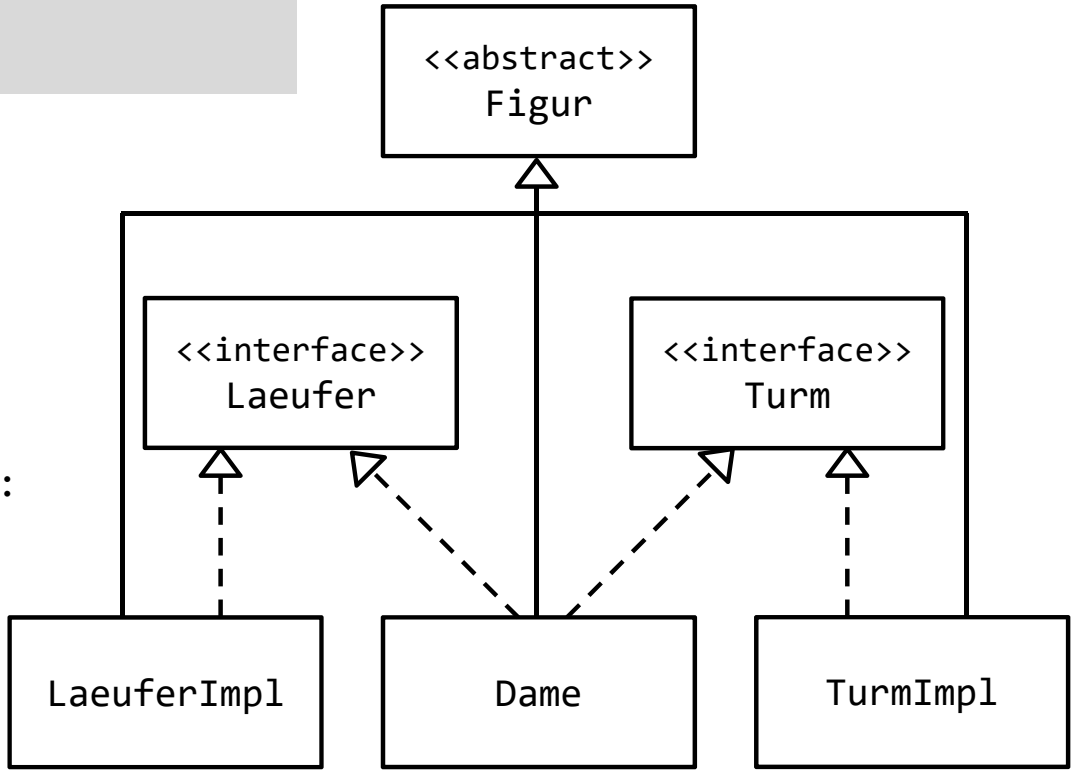
```
public class Dame extends Figur implements Laeufer, Turm
{
  @Override
  public Brett gibErlaubteFelder()
  {
    //alle waagrechten, senkrechten und diagonalen Felder
  }
}
```

Dame hat jetzt mehrere Typen gleichzeitig:

Dame
Figur
Object
Laeufer
Turm

Daher sind folgende Instanziierungen möglich:

```
Dame d = new Dame();
Turm t = new Dame();
Laeufer l = new Dame();
Figur f = new Dame();
Object o = new Dame();
```



Jetzt ist in den Interfaces Turm und Laeufer die Methode gibErlaubteFelder() spezifiziert, sowie in der abstrakten Klasse Figur.

Es wäre gut, wenn man diese Methoden-Spezifikation aus der abstrakten Klasse erben könnte.

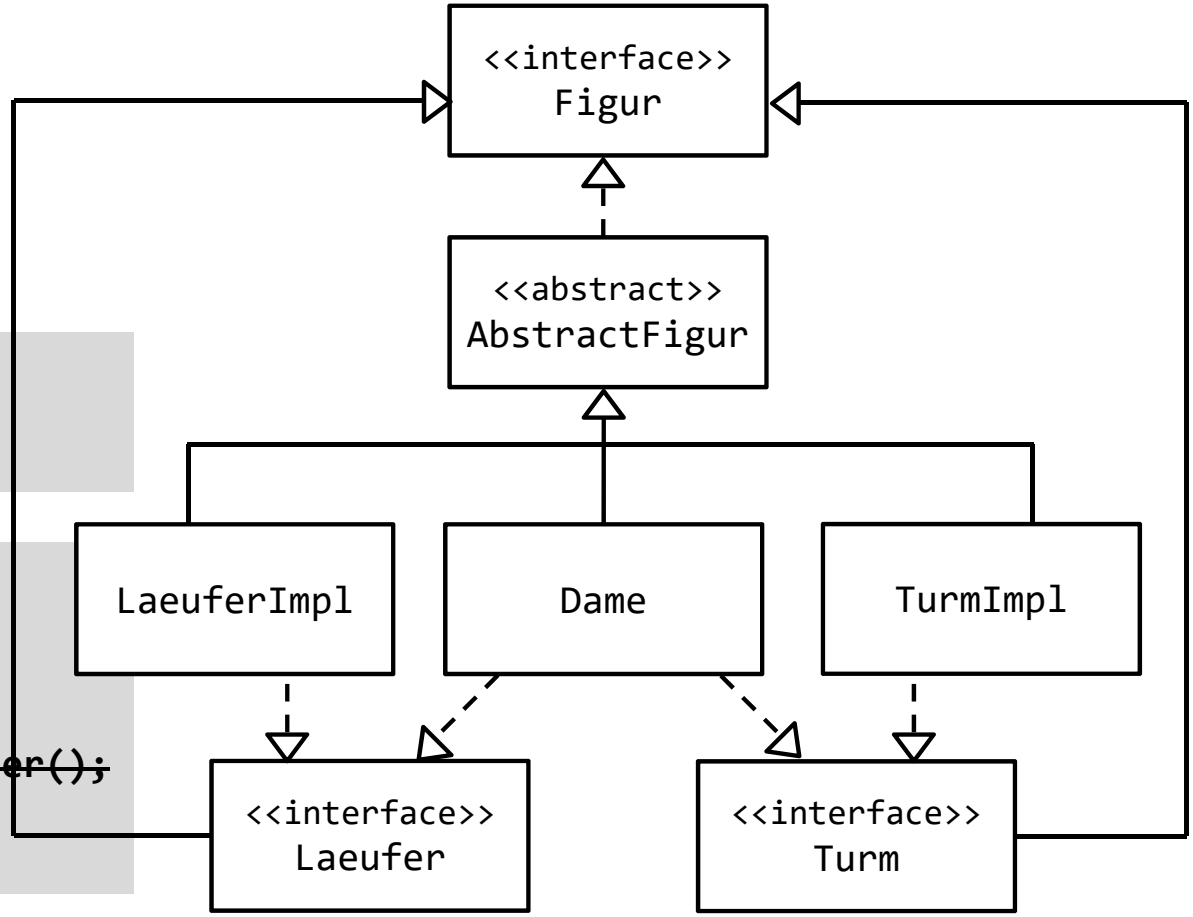
Daher führt man noch ein Interface ein, von dem die anderen Interfaces **erben**.

Die abstrakte Klasse wird in **AbstractFigur** umbenannt.

Ein neues Interface Figur wird eingeführt.

```
public interface Figur {
  public Brett gibErlaubteFelder();
}
```

```
public abstract class AbstractFigur
  implements Figur {
  int x;
  int y;
  public abstract Brett gibErlaubteFelder();
   …
}
```



Interfaces werden benutzt, um den Typ festzulegen,
abstrakte Klassen um Implementierungsvorgaben zu machen.
Der Name der abstrakten Klasse setzt sich dann zusammen aus Abstract*Interfacename.*

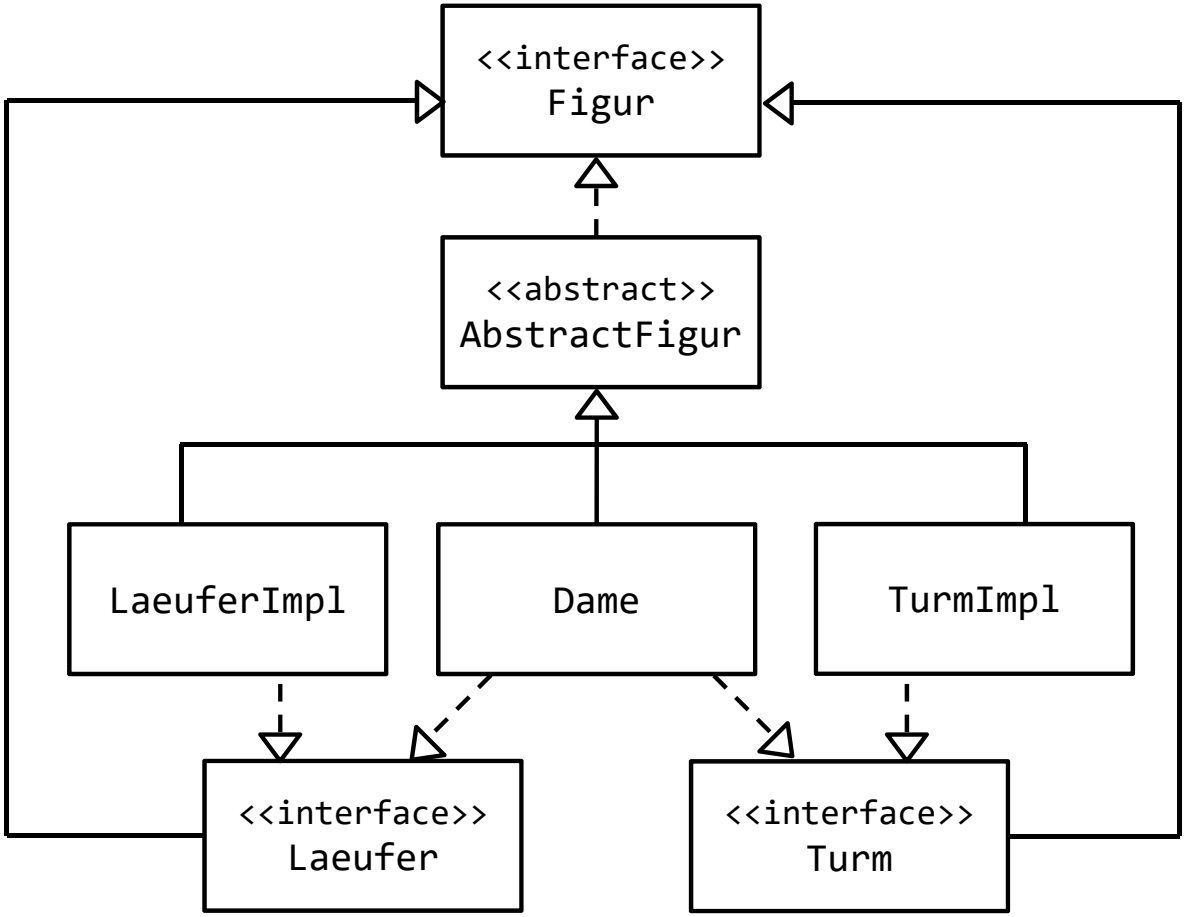
(siehe auch: Joshua Bloch: Effective Java, 3rd Edition, S.99ff, Addison-Wesley Professional, 2017)

Die Interfaces Turm und Laeufer erben von Figur und brauchen die Methode gibErlaubteFelder() nicht mehr selbst deklarieren.

```
public interface Figur {
  public Brett gibErlaubteFelder();
}
```

```
public interface Laeufer extends Figur
{
}
```

```
public interface Turm extends Figur
{
}
```



Interfaces können die Methodenspezifikation von anderen Interfaces durch das Schlüsselwort **extends** erben.

Die Vererbungshierarchie ist jetzt sauber spezifiziert.

Wie kann die Dame jetzt die Funktionalität des Läufers und des Turms nutzen, ohne den Code (teilweise) duplizieren zu müssen?

Wir haben immer noch das Problem, dass die Dame nicht von beiden Klassen das Verhalten erben kann.

Komposition

Wir könnten das Problem mit einem anderen Ansatz angehen:

Wir könnten sagen, dass eine Dame aus mehreren Teilen besteht, nämlich

- aus einem Turm
- aus einem Läufer

```
public abstract class AbstractFigur
  implements Figur {
  int x;
  int y;

  public AbstractFigur(int x, int y) {
    if (x >= 1 && x <= 8) this.x = x;
    if (y >= 1 && y <= 8) this.y = y;
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

```
public class LaeuferImpl extends AbstractFigur
  implements Laeufer {

  public LaeuferImpl(int x, int y) {
    super(x, y);
  }
  public Brett gibErlaubteFelder() {
    //alle diagonalen Felder
  }
}
```

```
public class TurmImpl extends AbstractFigur
  implements Turm {

  public TurmImpl(int x, int y) {
    super(x, y);
  }

  public Brett gibErlaubteFelder() {
    //alle waagrechten und senkrechten Felder
  }
}
```

Zunächst ergänzen wir einen Konstruktor in `TurmImpl`, `LaeuferImpl` und `AbstractFigur`.

```
public class Dame extends AbstractFigur
  implements Laeufer, Turm {

  Laeufer lauefer;
  Turm turm;

  public Dame(int x, int y)
  {
    super(x, y);
    lauefer = new LaeuferImpl(x, y);
    turm = new TurmImpl(x, y);
  }

  @Override
  public Brett gibErlaubteFelder()
  {
    Brett brettLaeufer = laeufer.gibErlaubteFelder();
    Brett brettTurm = turm.gibErlaubteFelder();
    Brett kombiniertesBrett = brettTurm.kombiniere(brettLaeufer);
    return kombiniertesBrett;
  }
  …
}
```

Die Dame verfügt jetzt über einen Läufer und einen Turm auf gleicher Position.

Die Dame lässt sich je ein Brett mit erlaubten Feldern vom Läufer und vom Turm zurückgeben und kombiniert diese zu einem Brett.

```
public class Dame extends AbstractFigur
  implements Laeufer, Turm {

  Laeufer lauefer;
  Turm turm;

  …

  @Override
  public void setX(int x)
  {
    laeufer.setX(x);
    turm.setX(x);
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  @Override
  public void setY(int y)
  {
    laeufer.setY(y);
    turm.setY(y);
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

Wollen wir die Position der Dame ändern, müssen wir auch gleichzeitig die Position des Läufers und des Turms ändern!

Da es sie Methoden setX und setY nicht im Interface Figur gibt, müssen wir diese noch ergänzen!

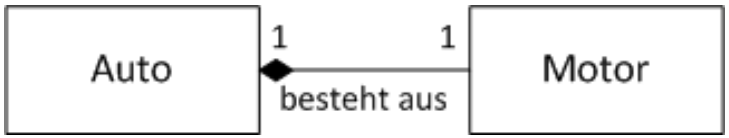
```
public interface Figur {
  public Brett gibErlaubteFelder();
  public void setX(int x);
  public void setY(int y);
}
```

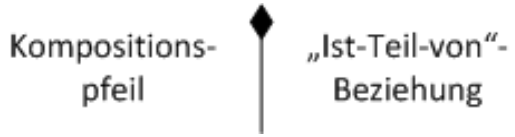
Wir haben den Code durch eine **Komposition** anstelle von Vererbung wiederverwendet.

Die Dame hat ihre Fähigkeit, die erlaubten Felder zu bestimmen, aus den Fähigkeiten des Turm und Läufers zusammengesetzt/komponiert.

Beziehungen zwischen Klassen: Komposition

- Eine **Komposition** stellt eine „ist-Teil-von“-Beziehung dar.
- Beispiel: Ein Motor ist Teil eines Autos. Der Motor ist an den Lebenszyklus des Autos gebunden. Hört das Auto auf zu existieren, gibt es den Motor auch nicht mehr.





mögliche Umsetzung!

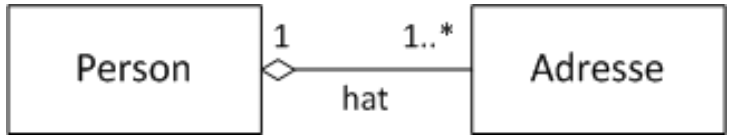
```
public class Auto
{
  Motor motor;

  public Auto() {
    motor = new Motor();
  }
  …
}
```

```
public class Motor
{
  …
}
```

Beziehungen zwischen Klassen: Aggregation

- Eine schwächere Form der Komposition ist die Aggregation. Eine **Aggregation** stellt eine „hat-ein“-Beziehung dar.
- Beispiel: Eine Person hat eine oder mehrere Adressen. Hört die Person auf zu existieren, gibt es die Adressen immer noch.



Leserichtung beachten!

mögliche Umsetzung!

```
public class Person
{
  Adresse[] adressen = new Adresse[10];

  public Person(Adresse adresse) {
    adressen[0] = adresse;
  }
  …
}
```

```
public class Adresse
{
  …
}
```

weitere Adressen könnten bspw. über Methodenaufrufe hinzugefügt werden

Schauen wir uns noch ein Problem an, das man mit Komposition lösen kann:

Unser Rechteck-Quader-Problem

```
public class Rechteck {

  double laenge;
  double breite;

  public Rechteck(double laenge,
    double breite) {
    this.laenge = laenge;
    this.breite = breite;
  }

  public double berechneFlaeche() {
    return laenge*breite;
  }
}
```

```
public class Quader extends Rechteck {

  double tiefe;


  public Quader(double laenge, double breite,
    double tiefe) {
    super(laenge, breite);
    this.tiefe = tiefe;
  }

  public double berechneVolumen() {
    return berechneFlaeche()*tiefe;
  }
}
```

Wir hatten folgende Probleme:

- Auf dem Quader kann die Methode `berechneFlaeche` aufgerufen werden.
- Der Quader ist **kein** Rechteck! (keine ist-ein Beziehung).

```
public class Rechteck {

  double laenge;
  double breite;

  public Rechteck(double laenge,
    double breite) {
    this.laenge = laenge;
    this.breite = breite;
  }

  public double berechneFlaeche() {
    return laenge*breite;
  }

  public double berechneUmfang() {
    return 2*laenge + 2*breite;
  }
}
```

```
public class Quader extends Rechteck {

  double tiefe;


  public Quader(double laenge, double breite,
    double tiefe) {
    super(laenge, breite);
    this.tiefe = tiefe;
  }

  public double berechneVolumen() {
    return berechneFlaeche()*tiefe;
  }
}
```

Noch ein weiteres Problem:

- Wenn wir das Rechteck um eine Methode ergänzen, ergänzt sich auch die „Schnittstelle“ des Quaders um diese Methode.

➢ Die Kapselung, die durch eine Klasse geboten wird, wird verletzt.

```
public class Rechteck {

  double laenge;
  double breite;

  public Rechteck(double laenge,
    double breite) {
    this.laenge = laenge;
    this.breite = breite;
  }

  public double berechneFlaeche() {
    return laenge*breite;
  }

  public double berechneUmfang() {
    return 2*laenge + 2*breite;
  }
}
```

```
public class Quader ~~extends Rechteck~~ {

  double tiefe;
  Rechteck rechteck;

  public Quader(double laenge, double breite,
    double tiefe) {
    rechteck = new Rechteck(laenge, breite);
    this.tiefe = tiefe;
  }

  public double berechneVolumen()
  {
    return rechteck.berechneFlaeche()*tiefe;
  }
}
```

- Quader hat als Schnittstelle nach außen nur die Methode berechneVolumen().
- Kapselung bleibt intakt.
- Die Schnittstelle von Rechteck kann problemlos erweitert werden.
- Wir modellieren keine falsche ist-ein-Beziehung mehr.

Zusammenfassend kann man sagen:

- Häufig ist die Wiederverwendung von Code durch Vererbung (Implementierungsvererbung) nicht optimal.
- Vererbung nur einsetzen, wenn eine ist-ein(e)-Beziehung vorliegt!
- Wiederverwendung durch Komposition ist oft besser
  (siehe auch: Joshua Bloch: Effective Java, 3rd Edition, S.87ff, Addison-Wesley Professional, 2017)
- Zur reinen Typisierung (od. Spezifikationsvererbung) sind oft Interfaces besser geeignet („Dame ist ein Läufer“).
- Um von einem Interface minimale Implementierungsvorgaben zu machen, verwendet man eine abstrakte Klasse.

```
public abstract class AbstractFigur
  implements Figur {
  int x;
  int y;

  public AbstractFigur(int x, int y) {
    setX(x);
    setY(y);
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 1 && x <= 8)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 1 && y <= 8)
      this.y = y;
  }
}
```

Ein paar kleine Nachbetrachtungen:

Im Konstruktor sollte man keine **überschreibbaren** Methoden aufrufen.

- (siehe auch: Joshua Bloch: Effective Java, 3rd Edition, S.95, Addison-Wesley Professional, 2017)

Im Konstruktor sollte man keine **überschreibbaren** Methoden aufrufen.

```
public class Angestellter {
  String name;
  String vorname;
  int gehalt;

  public Angestellter(String name, String vorname,
    int gehalt) {
    this.name = name;
    this.vorname = vorname;
    setGehalt(gehalt);
  }

  public void setGehalt(int gehalt) {
    if(gehalt > 0) this.gehalt = gehalt;
  }
}
```

```
public class Consultant extends Angestellter {
  int GEHALTSUNTERGRENZE = 55000;
  int GEHALTSOBERGRENZE = 80000;

  public Consultant(String name, String vorname,
    int gehalt) {
    super(name, vorname, gehalt);
  }

  @Override
  public void setGehalt(int gehalt) {
    if (gehalt >= GEHALTSUNTERGRENZE
      && gehalt <= GEHALTSOBERGRENZE)
      this.gehalt = gehalt;
  }
}
```

Im Konstruktor sollte man keine **überschreibbaren** Methoden aufrufen.

Abhilfe: Wenn die Methode setGehalt() im Konstruktor unbedingt benutzt werden soll, kann durch den Modifier **final** unterbunden werden, dass diese überschrieben wird.

```
public class Angestellter {
  String name;
  String vorname;
  int gehalt;

  public Angestellter(String name, String vorname,
    int gehalt) {
    this.name = name;
    this.vorname = vorname;
    setGehalt(gehalt);
  }

  public final void setGehalt(int gehalt) {
    if(gehalt > 0) this.gehalt = gehalt;
  }
}
```

```
public class Consultant extends Angestellter {
  int GEHALTSUNTERGRENZE = 55000;
  int GEHALTSOBERGRENZE = 80000;

  public Consultant(String name, String vorname,
    int gehalt) {
    super(name, vorname, gehalt);
    if (gehalt >= GEHALTSUNTERGRENZE
      && gehalt <= GEHALTSOBERGRENZE)
      this.gehalt = gehalt;
  }

  @Override
  public void setGehalt(int gehalt) {
    if (gehalt >= GEHALTSUNTERGRENZE
      && gehalt <= GEHALTSOBERGRENZE)
      this.gehalt = gehalt;
  }
```

Man kann auch unterbinden, dass Klassen überhaupt durch Vererbung erweitert werden.

Bspw. stellt Google Java Klassen zur Verfügung, die Werbung in einer Android-Anwendung passend zum aktuellen Userprofil anzeigen.

Die Werbeanzeige wird abgerechnet.

Google möchte sich dagegen wehren, dass die Abrufe oder die Anzeige der Werbung, etc. manipuliert werden.

Durch den **final** Modifier kann man anderen Klassen verbieten, von der eigenen Klasse zu erben.

```
final public class AdfetcherActivity
{
  …
}
```

```
public class MyClass extends AdfetcherActivity
{

}
```