Bessere Struktur durch Schnittstellenvererbung

Buch, Kapitel 12; Weitere Techniken zur Abstraktion ab 12.6

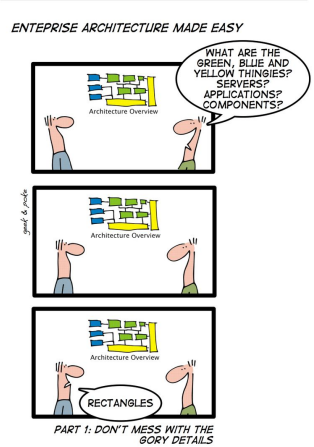
J. Kleimann, H.-W. Sehring, D. Versick, F. Zimmermann

Studiengang Wirtschaftsinformatik

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Vorlesungsinhalt

1. Lerninhalte
2. Ein Beispiel
3. Schnittstellenvererbung (Subtyping)
4. Erweiterung vs. Modifikation
5. Polymorphie
6. Zusammenfassung
7. Comparable



Lernziele

In dieser Lektion werden Sie

- Schnittstellenvererbung (Subtyping),
- Polymorphismus
- und polymorphe Variablen

kennenlernen.

Varianten von Geschäftsobjekten

- In IT Systemen entsteht häufig der Bedarf nach unterschiedlichen Varianten von ähnlich zu behandelnden Geschäftsobjekten.
- Unterschiede können im Status der Objekte oder im Verhalten der Objekte liegen.
- In nicht objektorientierten Programmiersprachen entstehen durch die Varianten in der Regel Wartungsprobleme durch Code-Duplikate.

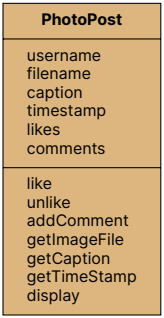
Das Socialnetwork-Beispiel

Sie arbeiten in einem Unternehmen eines gewissen Pfennig Salztal an einer völlig neuartigen Anwendung, die

- einen Nachrichten-Feed bietet
- verschiedene Varianten von Posts speichert: Text- und Photomeldungen
  - `MessagePost`: mehrzeilige Textnachrichten
  - `PhotoPost`: Foto und Überschrift
- Operationen auf den Meldungen anbietet:
  - z.B.: suchen, anzeigen und entfernen

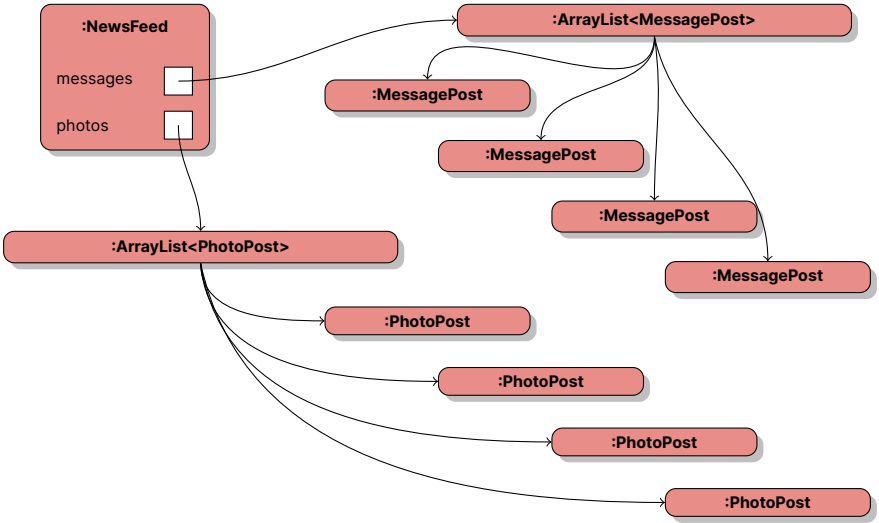
Netzwerkobjekte

Netzwerkklassen

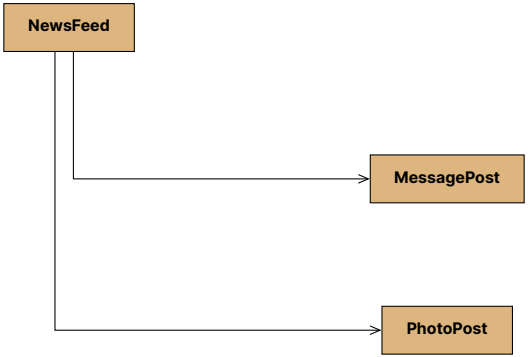


- Obere Hälfte: Felder
- Untere Hälfte: Methoden

Netzwerkobjektmodell



Klassendiagramm



Übung 1

Erkunden Sie die Social-Network Applikation:

1. Öffnen Sie das Projekt *network-v1*. Sie finden das Projekt im Moodle.
2. Erzeugen Sie ein `MessagePost` und ein `PhotoPost` Objekt in BlueJ.
3. Liken Sie das `MessagePost` Objekt.
4. Kommentieren Sie das `PhotoPost` Objekt.
5. Erzeugen Sie ein `NewsFeed` Objekt.
6. Fügen Sie die beiden Objekte dem `Newsfeed` Objekt hinzu.
7. Zeigen Sie den `Newsfeed` mit Hilfe von `show` an.

MessagePost Quellcode (ein Ausschnitt)

```
public class MessagePost {
  private String username;
  private String message;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;

  public MessagePost(String author,
      String text) {
   username = author;
   message = text;
   timestamp = System.
       currentTimeMillis();
   likes = 0;
   comments = new ArrayList<>();
  }

  public void addComment(String text)
      ...
  public void like() ...
  public void display()...
  ...
}
```

PhotoPost SourceCode (ein Ausschnitt)

```
public class PhotoPost {
  private String username;
  private String filename;
  private String caption;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;

  public PhotoPost(String author,
      String filename,
                   String caption) {
    username = author;
    this.filename = filename;
    this.caption = caption;
    timestamp = System.
        currentTimeMillis();
    likes = 0;
    comments = new ArrayList<>();
  }

  public void addComment(String text)
      ...
  public void like() ...
  public void display() ...
  ...
}
```

NewsFeed

```
public class NewsFeed {
  private ArrayList<MessagePost> messages;
  private ArrayList<PhotoPost> photos;
  public void addMessagePost(MessagePost msg){  messages.add(msg); }
  public void addPhotoPost(PhotoPost ph){ photos.add(ph); }
  public void show() {
    for (MessagePost message : messages) {
       message.display();
       System.out.println();  // empty line between posts
    }
    for (PhotoPost photo : photos) {
         photo.display();
         System.out.println();  // empty line between posts
    }
  }
  ...
}
```

Kritik am Netzwerk

- Die Klassen enthalten duplizierten Quellcode:
  1. Die `MessagePost` und `PhotoPost` sind ähnlich (große Teile sind identisch)
  2. Wir haben ebenfalls in der Klasse `NewsFeed` in der `show` Methode duplizierten Code. Hier wiederholt sich die Strategie, wie mit den Posts umgegangen wird.
- Dies macht Wartung schwieriger und arbeitsaufwändiger
- Erhöht die Gefahr von Fehlern durch nicht korrekte Wartung

Übung 2

1. Finden Sie Gemeinsamkeiten und Unterschiede in den Klassen.
2. Schätzen Sie die Auswirkungen der Codeduplizierung auf Basis der betroffenen Lines of Code (LOC) ab.
3. Schätzen Sie den Aufwand ab, der entstehen würde, wenn Sie eine neue Variante `EventPost` einführen. In welchen Klassen entsteht der Aufwand?
4. Beziehen Sie in Ihre Überlegungen zu Punkt 2 und 3 bitte Häufigkeitsbetrachtungen mit ein. Gehen Sie davon aus, dass die Klassen `MessagePost` und `PhotoPost` eine zentrale Ressource in Ihrem Unternehmen beschreiben, von denen es ca 10 unterschiedliche Varianten gibt. Die Klasse `Newsfeed` dagegen beschreibt einen Umgang mit dieser zentralen Ressource, die in der Regel sehr viele „Geschwister“ hat. In der Ausbaustufe gehen wir von 100 solchen Diensten aus.

Auflösung Übung 2

- Auf Basis der LOC scheint die Duplikation in den Klassen `PhotoPost` und `MessagePost` problematischer, da in `Newsfeed` ca 10 LOC verdoppelt werden, in den `Post`-Klassen je nach zählweise zwischen 30 und 50 Zeilen den Konstruktoren und in den Methoden `like`, `unlike`, `addComment`, … . Die Duplikation in den `Post`-Klassen ist scheinbar(?) wichtiger.
- Wenn sie eine zusätzlich Variante hinzufügen, sind die neue Klasse und die Klasse `Newsfeed` betroffen. Also zwei Klassen.
- Wenn man die Häufigkeitsbetrachtung macht, werden es vielleicht zehn unterschiedliche `Post`-Klassen werden, aber mehrere hunderte von `show`-Methoden, die die zentrale Ressource `Post` verarbeiten: $10 \cdot 50 = 500$ *LOC* aus den `Post`-Klassen stehen $10 \cdot 100 = 1000$ *LOC* gegenüber.
- Unter Einbeziehung der Häufigkeitsbetrachtung werden bei einer Einführung einer neuen Klasse 101 Klassen betroffen sein.

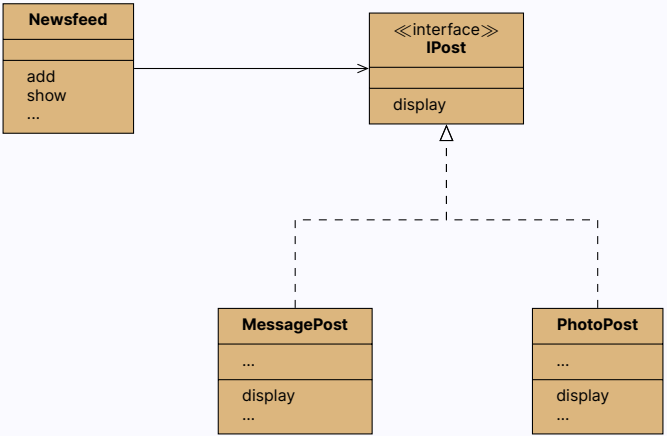
Was soll man aus Übung 2 lernen?

- Für den Anfänger scheint es so zu sein, dass der Hauptaufwand in den `Post`-Klassen versteckt ist. Das ist jedoch ein Trugschluss.
- Die Praxis zeigt: **Die Vermeidung von Code-Duplikaten in der `Newsfeed` Klasse ist viel bedeutender als die Vermeidung innerhalb der `Post`- Klassen.**

Vermeidung von Code-Duplikaten in `Newsfeed`

- Statt die `Post`-Klassen in `Newsfeed` zu verwenden definiert man eine Schnittstelle (**Interface**) für die Postklassen.
- Java erlaubt es, `Newsfeed` nur unter Nutzung der Interfaces zu formulieren. Code-Duplikate in `Newsfeed` werden dadurch vermeiden.
- Die Postklassen sind „**Implementation**en“ der Schnittstelle.
- An allen Stellen, an denen die Schnittstelle genutzt wird, dürfen Implementationen der Schnittstelle stehen. Eine Schnittstelle darf durch eine beliebige Implementation ersetzt werden. Dies Konzept nennt man **Polymorphie**.
- Diesen Mechanismus nennt man Schnittstellenvererbung (**Subtyping**) .

Verwendung von Schnittstellenvererbung



Das Interface IPost

Interface

```
public interface IPost {
  public void display();
}
```

- Interface-Definitionen unterscheiden sich von Klassendefinitionen durch das Schlüsselwort interface.
- Interfaces dürfen keine Exemplarvariablen[1] haben. Sie repräsentieren **keinen** Status.
- Interfaces beschreiben nur die Signatur von Methoden nicht deren Implementation[2]. Sie definieren somit auch **kein** Verhalten.
- Interfaces haben **keine** Konstruktoren.

[1] und auch keine Klassenvariablen, es sei denn sie sind final

[2] Seit Java 8 sind default Implementationen möglich.

NewsFeed **nutzt Schnittstelle**

```
public class NewsFeed {
  private ArrayList< IPost > posts;
  public addPost( IPost  post){
    posts.add(post);
  }
  public void show() {
    for ( IPost  post : posts) {
      post.display();
      System.out.println();
// empty line between posts
    }
  }...}
```

- Interfaces können als Typ an allen Stellen angegeben werden, an denen auch eine Klasse angegeben werden kann:
  - Als Typ einer Variablen: `IPost post` (zweimal!).
  - Als Typparameter in einer Collection: `ArrayList<IPost>`.
- Methoden des Interfaces können auf Variablen des des Typs der Schnittstelle angewendet werden: `post.display()`

Schnittstellenvererbung in Java

```
public class PhotoPost implements IPost {
...
 public void display(){
  ...
  System.out.println(filename);
 };
}
```

```
public class MessagePost implements IPost {
...
 public void display(){
  ...
  System.out.println(message);
 };
}
```

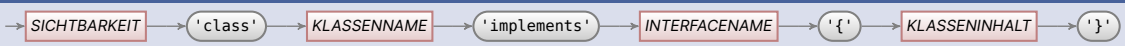
- Die Schnittstelle `IPost` **definiert** die Methode `display()` **implementiert** sie aber nicht.
- Die Klassen `PhotoPost` und `MessagePost` müssen die Methode `display()` aus `IPost` implementieren.
- Die Klassen `PhotoPost` und `MessagePost` implementieren die Methode `display()` aus `IPost` unterschiedlich.

Interfacedefinition und -implementation (Syntax)

Interfacedefinition

- Die Sichtbarkeit einer Schnittstelle ist in der Regel **`public`**.
- Die Sichtbarkeit darin deklarierter Methoden muss nicht explizit mit **`public`** angegeben werden. Häufig wird dies auch weggelassen.

Interface-Implementation



- Der Klasseninhalt muss Implementationen für alle Methoden der Schnittstelle enthalten.

Übung 3 siehe Moodle Test Foliensatz 9

- Setzen sie die Schnittstellenvererbung in dem Projekt network-v1 ein.
- Wiederholen Sie die Übung 1, um sich zu überzeugen, dass das neue System sich genau wie das alte verhält.

Übung 4

- Wie in Übung 2 schätzen Sie den Aufwand ab, der entstehen würde, wenn Sie eine neue Variante `EventPost` einführen. In welchen Klassen entsteht der Aufwand? Wieviel sind es, wenn Sie die Häufigkeitsbetrachtung mit einbeziehen?

Erweiterung vs. Modifikation

- Erweiterungen im objektorientierten Sinn verändern die bestehende Code-Basis nicht.
- Code-Modifikationen sind keine Erweiterungen. Erweiterungen sind keine Code-Modifikationen.
- Da Erweiterungen keine Modifikationen sind, wird der bestehende Code weiter funktionieren.
- Umsetzung der alten Programmiererweisheit: Never change a running system.

Erweiterungspunkte

- Interfaces stellen Erweiterungspunkte dar.
- Sie ermöglichen Software an gezielt gewählten Punkten zu erweitern.
- Diese Erweiterungspunkte sind eine axis of change. D.h. mit den Interfaces können gezielt neue Anforderungen umgesetzt werden.
- Die Wahl der Erweiterungspunkte ist entscheidend für einen objektorientierten Entwurf:
  - Interfaces sollten möglichst schlank sein, da sie von vielen Klassen verwendet werden und deshalb nur schwer änderbar sind.
  - Erweiterungspunkte erzeugen Programmieraufwand. Zu viele Erweiterungspunkte sind nicht effektiv. Keine unnötigen Erweiterungspunkte vorsehen.

Polymorphe Variablen

- Variablen die einen Interfacetyp haben in Java sind **polymorph**.
- Sie können Exemplare aller implementierenden Klassen enthalten.
- Implementierende Klassen sind zuweisungskompatibel zu Interfaces.

```
IPost p = new MessagePost();
p = new PhotoPost();
NewsFeed n = new Newsfeed()
n.addPost(new MessagePost()); //Parameter vom Typ IPost!
n.addPost(new PhotoPost());
```

- Die Zuweisungskompatibilität ist nicht bidirektional:

```
IPost p = new MessagePost();
MessagePost m = p; // compile-time error!⇒Narrowing Reference Conversion
new ArrayList<MessagePost>().add(p); // compile-time error!
```

Polymorphe Collections

- Ein Typparameter begrenzt den Grad der Polymorphie

```
ArrayList<IPost> list = new ArrayList<>();
list.add(new MessagePost());
list.add(new PhotoPost());
list.add("post"); //compile time error
```

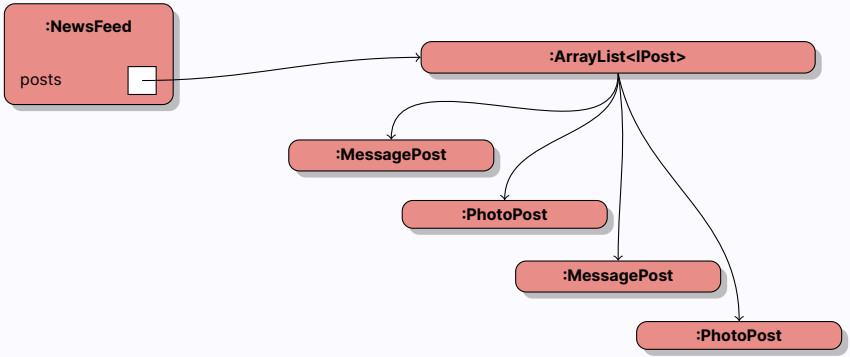
- Die Collection `list` kann sowohl `MessagePost`-Exemplare als auch `PhotoPost`-Exemplare aufnehmen.
- Die Methoden `add` und `get` der Collection nutzen dann `IPost` als Parameter bzw. Rückgabetyp.

Polymorphe Methodenaufrufe

- Ein Methodenaufruf auf einem Objekt kann zu unterschiedlichen aufgerufenen Methoden führen.
- Welche Methode genau ausgeführt wird entscheidet das Objekt: Je nachdem ob in `post` ein `MessagePost` oder ein `PhotoPost` steckt, wird durch `post.display()` die `display`-Methode aus `MessagePost` oder aus `PhotoPost` aufgerufen.

```
ArrayList<IPost> posts = new ArrayList<>();
public void show(){
  for (IPost post: posts){
    post.display();
    System.out.println();
  }
}
```

Polymorphe Collection und Methodenaufrufe



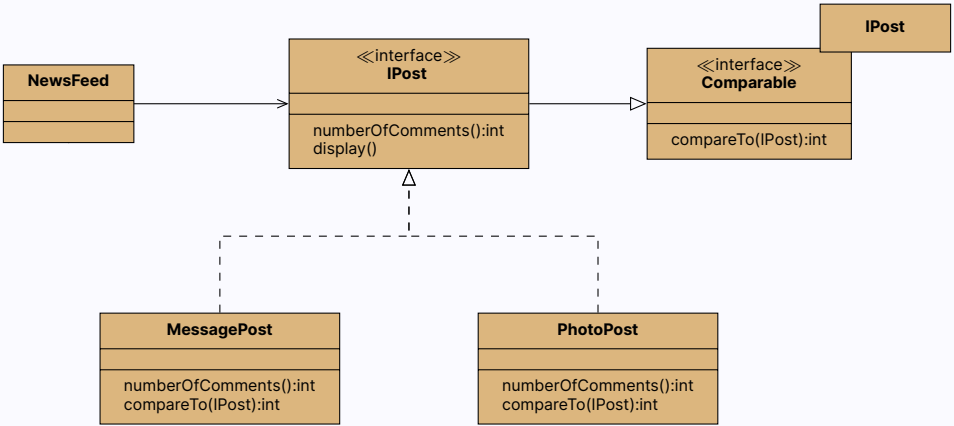
- **Polymorpher Methodenaufruf**: `post.display();` ist die `display`-Methode aus `MessagePost` für ein `MessagePost`-Exemplar und die Methode aus `PhotoPost` für ein `PhotoPost`-Exemplar.

Zusammenfassung

- Schnittstellenvererbung erlaubt die Formulierung von Verarbeitungsstrategien (Methode `show` in `Newsfeed`) auf Basis einer Schnittstelle unabhängig von den implementierenden Klassen (`MessagePost`, `PhotoPost`).
- Nutzen der Schnittstellenvererbung
  - Vermeidet Code-Duplikate in den Strategieklassen
  - Die Strategieklassen werden wiederverwendbar
  - Vereinfacht Wartung weil Erweiterungen möglich werden, wo sonst Modifikationen erforderlich sind.
- Variablen, Collections und Methodenaufrufe sind polymorph.
- Der Nutzen ist so hoch, dass wann immer möglich Schnittstellen genutzt werden sollten und nicht konkrete Klassen.

Das Interface `Comparable`

- `NewsFeed` möchte eine Collection von `Post`-Objekten nach der Anzahl der Kommentare sortieren.



Nutzung von Comparable<E>

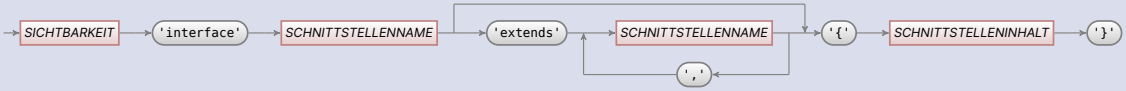
```
public interface Comparable<E>{  //
    Teil des JDK
  public int compareTo(E e);
}

public interface IPost extends
    Comparable<IPost> {
  public void display();
  public int numberOfComments();
}

public class MessagePost implements
    IPost {
  private ArrayList<String> comments;
  ...
  public void display(){ ... }
  public int numberOfComments() {
      return comments.size();}
  public int compareTo(IPost other) {
    return numberOfComments()-other.
        numberOfComments();
  }
  ...
}
```

Klassendefinition, Schnittstellen (Syntax)

Verallgemeinert (Schnittstellendefinition)



Verallgemeinert (Klassendefinition, Schnittstellenimplementierung)

Vergleichen mit `compareTo`

- Die Methode `compareTo(IPost other)` soll in der Lage sein, das aktuelle Objekt (ein `IPost`) mit einem anderen `IPost` zu vergleichen.
- Durch den Typparameter von `Comparable` ist die Einhaltung einer strengen Typisierung garantiert.
- Die über die `compareTo` Methode definierte Ordnung einer Klasse nennt man auch die natürliche Ordnung der Klasse.
- Das Ergebnis des Vergleichs soll durch eine Zahl ausgedrückt werden:
  - Dabei ist das Objekt x kleiner dem Objekt y, genau dann, wenn `x.compareTo(y) < 0`.
  - Dabei ist das Objekt x größer dem Objekt y, genau dann, wenn `x.compareTo(y) > 0`.
  - Dabei ist das Objekt `x.equals(y)`, genau dann, wenn `x.compareTo(y) == 0`.

Sortieren einer Sammlung von Comparables

Das Sortieren einer Liste von `IPost` implementierenden Objekten ist nun sehr einfach:

```
public class NewsFeed {
  private List<IPost> posts;
  public void sortPosts() {
     Collections.sort(posts);
  }
}
```

Es gibt leider noch einiges zu beachten:

- Die Objekte einer Liste müssen miteinander vergleichbar sein. Da die `compareTo(IPost)` Methode nur Exemplare von `IPost` implementierenden Klassen miteinander vergleichen kann, ist sicherzustellen, dass keine Objekte, die Nicht-`IPost` sind in die Collection kommen können. Dies ist z.B. durch die durchgängige Verwendung von Generics möglich.

Typhierarchie

- Anstatt die Schnittstelle `IPost` die Schnittstelle `Comparable` erweitern zu lassen, könnte man auch auf die Idee kommen in Post zwei Schnittstellen zu implementieren: `IPost` und `Comparable`.

```
public class MessagePost implements IPost, Comparable<IPost> {
  ...
  public void display(){ ... }
  public int numberOfComments() {return comments.size();}
  public int compareTo(IPost other) {
    return other.numberOfComments()-numberOfComments();
  }
  ...
}
```

Sortieren mit Comparator

- Wenn die durch `compareTo` definierte Ordnung nicht kompatibel zu `equals` ist, bietet es sich an, stattdessen mit einem `Comparator` zu arbeiten.

```
public interface IPost{
  public int numberOfComments(); ...
}
public class CommentComparator implements Comparator
    <IPost> {
  public int compare(IPost p1, IPost p2) {
    return p1.numberOfComments()-p2.numberOfComments
        ();
  }
}
public class NewsFeed{
  private List<IPost> posts;
  public void sortPosts(){
     Collections.sort(posts, new CommentComparator()
         );
  }
}
```

Zu folgenden Themen später mehr...

- Interfaces können **`default`** Implementierungen von Methoden definieren. (Foliensatz 11)

und für Fortgeschrittene...

- Dependency Inversion
- Dependency Injection
- Factories
- Abstract Factories

Fragen?

Für weitere Fragen im Nachgang können Sie mich gerne über Moodle oder via E-Mail kontaktieren!

Konzepte: Zusammenfassung I

- **Interface** Eine Interface (Eine Schnittstelle) definiert die Methoden, die von allen Implementationen implementiert werden. Sie dient zum einheitlichen Ansprechen der Implementationen.
- **Implementation** Eine Implementation (Realisierung) einer Schnittstelle muss alle Methoden, die in der Schnittstelle definiert sind, implementieren, dass heißt Code für diese Java Methode zur Verfügung stellen.
- **Polymorphie** Polymorphie ist die Eigenschaft einer typsicheren Sprache, dass Objekte mit unterschiedlichen Typen und damit mit unterschiedlichem Status und Verhalten an derselben Stelle verwendet werden dürfen.
- **Subtyping** Subtyping erlaubt es, mehrere Klassen unter einer gemeinsamen Schnittstelle anzusprechen. Dabei kann der Code, der das Interface benutzt, für alle implementierenden Klassen wiederverwendet werden.

Interface

Eine Interface (Eine Schnittstelle) definiert die Methoden, die von allen Implementationen implementiert werden. Sie dient zum einheitlichen Ansprechen der Implementationen.

Implementation

Eine Implementation (Realisierung) einer Schnittstelle muss alle Methoden, die in der Schnittstelle definiert sind, implementieren, dass heißt Code für diese Java Methode zur Verfügung stellen.

Polymorphie

Polymorphie ist die Eigenschaft einer typsicheren Sprache, dass Objekte mit unterschiedlichen Typen und damit mit unterschiedlichem Status und Verhalten an derselben Stelle verwendet werden dürfen.

Subtyping

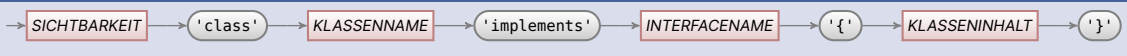
Subtyping erlaubt es, mehrere Klassen unter einer gemeinsamen Schnittstelle anzusprechen. Dabei kann der Code, der das Interface benutzt, für alle implementierenden Klassen wiederverwendet werden.

Syntaxdiagramme I

Interfacedefinition

- Die Sichtbarkeit einer Schnittstelle ist in der Regel **`public`**.
- Die Sichtbarkeit darin deklarierter Methoden muss nicht explizit mit **`public`** angegeben werden. Häufig wird dies auch weggelassen.

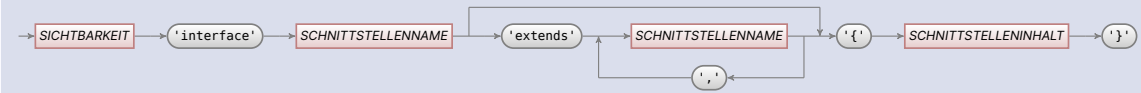
Interface-Implementation



- Der Klasseninhalt muss Implementationen für alle Methoden der Schnittstelle enthalten.

Syntaxdiagramme II

Verallgemeinert (Schnittstellendefinition)



Verallgemeinert (Klassendefinition, Schnittstellenimplementierung)