

Folien zur Vorlesung **Softwaretechnik**

Fakultät für Informatik
Sommersemester 2017

Stefan Dissmann

Folien zur Vorlesung **Softwaretechnik**

Teil 1: Einführung **Abschnitt 1.1: Überblick**

Softwaretechnik

ist die Disziplin in der Informatik, die sich beschäftigt mit

- Planung,
- Konstruktion,
- Erstellung,
- Überprüfung,
- Einführung,
- Betrieb und
- Wartung

von Softwaresystemen (Programmen).

Zielsetzungen:

- ❑ funktional geeignete Software herstellen
- ❑ qualitativ gute Software herstellen:
führt meist auch zu kostengünstigem Betrieb
- ❑ nachhaltig funktionale Eignung und Qualität sicherstellen:
führt meist auch zu kostengünstiger Wartung
- ❑ Einhalten des geplanten Entwicklungsaufwands

Zielsetzung: nachhaltig funktional geeignete Software

Die Funktionalität einer Software hängt ab von

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld.

Eignung wird dadurch sichergestellt, dass

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld

im Rahmen der Entwicklung analysiert und interpretiert werden.

Nachhaltigkeit wird dadurch sichergestellt,
dass Prognosen zu möglichen Änderungen an

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld

in die Analyse einbezogen

und während der Softwarekonzeption technisch berücksichtigt werden.

Zielsetzung: nachhaltig qualitativ gute Software

Qualitätsmerkmale

sind Aspekte, die in ihrer Gesamtheit die Qualität eines Softwareprodukts ausmachen.

Qualität von Software

wird aus verschiedenen Perspektiven unterschiedlich wahrgenommen:

- andere Gewichtung der Merkmale,
- andere mögliche Ausprägungen der Merkmale,
- andere Maße zum Beschreiben und Prüfen der Merkmale

Perspektiven sind:

Benutzung, Entwicklung, Administration, Betrieb, Beschaffung, ...

Literatur: Liggesmeyer, Peter: Software-Qualität – Testen, Analysieren und Verifizieren von Software, S. 1-48

http://link.springer.com/chapter/10.1007/978-3-8274-2203-3_1

Hoffmann, Dirk W.: Software-Qualität, S. 1-26

http://link.springer.com/chapter/10.1007/978-3-540-76323-9_1

Qualitätsmerkmale (Beispiele)

- ❑ Zuverlässigkeit
= korrektes Verhalten der Software über die Zeit (Fehler je Zeiteinheit)
- ❑ Robustheit/Fehlertoleranz
= Verhalten der Software bei unerwarteten Informationen aus der Umgebung
- ❑ Performanz
= Ausführungsschnelligkeit der Software
- ❑ Effizienz
= Wirtschaftlichkeit, mit der eine Software ihre Aufgaben erfüllt
- ❑ Skalierbarkeit
= Anpassung der Software an die Betriebsrealität
- ❑ Wartbarkeit
= Anpassung der Software an zukünftige – teilweise unbekannte – Anforderungen
- ❑ Interoperabilität
= Kooperationsfähigkeit mit anderer Software
- ❑ Portabilität
= Übertragbarkeit auf verschiedene Plattformen
- ❑ Bedienbarkeit/Verständlichkeit
= Brauchbarkeit für den Anwender

Anmerkungen:

- ❑ Unterscheidung möglich:
 - *interne* Qualitätsmerkmale beziehen sich auf die Implementierung der Software
 - *externe* Qualitätsmerkmale beziehen sich auf die Nutzung der Software

- ❑ Qualitätsmerkmale können sich gegenseitig fördern oder behindern:
 - Skalierbarkeit kann zu besserer Effizienz führen.
 - Portabilität kann die Interoperabilität einschränken.

Probleme bei der Erstellung von Software

**sind häufig eine Folge von
Konkurrenzsituationen zwischen**

- ❑ funktionaler Eignung der Software,
- ❑ guter Qualität der Software und
- ❑ kostengünstiger Erstellung von Software,
die führt zu
 - fehlender Definition von Anforderungen
 - fehlender Dokumentation von Anforderungen und Entscheidungen
 - fehlender Absicherung der Qualität der Umsetzung

**Die folgenden beiden Beispiele zeigen öffentlich gewordene Misserfolge
bei der Erstellung von Softwaresystemen.**

Probleme bei der Erstellung von Software – Beispiele

FBI Virtual Case File

- ❑ Ziel: einheitliche elektronische Speicherung aller Fallakten des FBI
- ❑ Laufzeit: 2000 – 2005, dann Projekt erfolglos abgebrochen
- ❑ Probleme:
 - ständige Änderungen der funktionalen (und qualitativen) Anforderungen
 - ständiger Personalwechsel im Projektmanagement
 - Softwareentwicklung durch Mitarbeiter ohne ausreichende technische Kompetenz
- ❑ Schaden: 170.000.000 US\$

fiscus-Software

- ❑ Ziel: bundeseinheitliche Software für Lohn-/Einkommenssteuererhebung
- ❑ Laufzeit: 1993 – 2005, dann Projekt erfolglos abgebrochen
- ❑ Probleme:
 - wechselnde politische Vorgaben
 - wechselnde technische Vorgaben
 - erzwungene Weiter- und Wiederverwendung vorhandener Software
 - mehrfacher Neustart des Projekts
- ❑ Schaden: 1.000.000.000 €

Aufbau der Vorlesung

Konzept zur Vermittlung der Inhalte in dieser Vorlesung:

- ❑ anknüpfen an die bereits bekannten Aspekte der Softwareentwicklung:
objektorientierte Programmierung mit Java
- ❑ ausgehend von der Programmierung
 - Präsentation von (allgemein nützlichen) Programmierkonzepten
 - Einführen von abstrakteren Beschreibungsformen
 - Nutzung einer umfangreichen Beispielsoftware zur Veranschaulichung

Vorteile dieses Konzepts:

- ❑ bessere Verständlichkeit der vorgestellten Konzepte während der Vermittlung
- ❑ bessere Möglichkeiten, um Beispiele zu formulieren

Konsequenz:

- ❑ Aufbau der Vorlesung folgt nicht dem Ablauf einer Softwareentwicklung

Inhalte in der Vorlesung

- ❑ Ideen der objektorientierten Softwaregestaltung
- ❑ Objektorientierte Konzepte in Java
- ❑ UML-Notation
- ❑ Technische Gestaltung von Software
- ❑ Testen von Software
- ❑ Vorgehensweisen zur Softwareentwicklung

Beispielsoftware *SWT-Starfighter*

- ❑ Arcade-Spiel
- ❑ Spieler (=Raumschiff) muss sich gegen Monster verteidigen
- ❑ zweidimensionale Graphik
- ❑ Steuerung über Tasten

- ❑ vollständig implementiert in Java
- ❑ vollständiger Programmtext ist verfügbar
- ❑ Entwickler ist SWT-Tutor: Dominic Starzinski

- ❑ Vorteile für die Veranstaltung
 - **ein** durchgängiges Beispiel
 - Änderungen im Programmtext können selbst ausprobiert werden
 - visuelle Ausrichtung des Spiels hilft, die Wirkung von Änderungen zu beobachten
 - die Projektaufgaben zum Erwerb der Studienleistung werden als Erweiterungen des Spiels definiert

kurze Präsentation

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Anforderungen an zukünftige (Weiter-)Entwicklung:

- ☐ weitere Monster
- ☐ anderes Verhalten von Monstern
- ☐ weitere Anzeigen
- ☐ (etwas) anderer Spielablauf
- ☐ andere Grafiken

- ☐ ähnliche Spiele mit anderem Kontext

**=> Die notwendigen Änderungen müssen bereits bei der Entwicklung
vorbereitet werden:**

Wartbarkeit!

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:

The diagram consists of two ovals. The left oval is black and contains the text 'allgemeines Framework für Arcade-Spiele'. The right oval is red and contains the text 'konkrete Ergänzungen für SWT-Starfighter'.

**allgemeines Framework
für
Arcade-Spiele**

**konkrete Ergänzungen
für
*SWT-Starfighter***

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:



**allgemeines Framework
für
Arcade-Spiele**

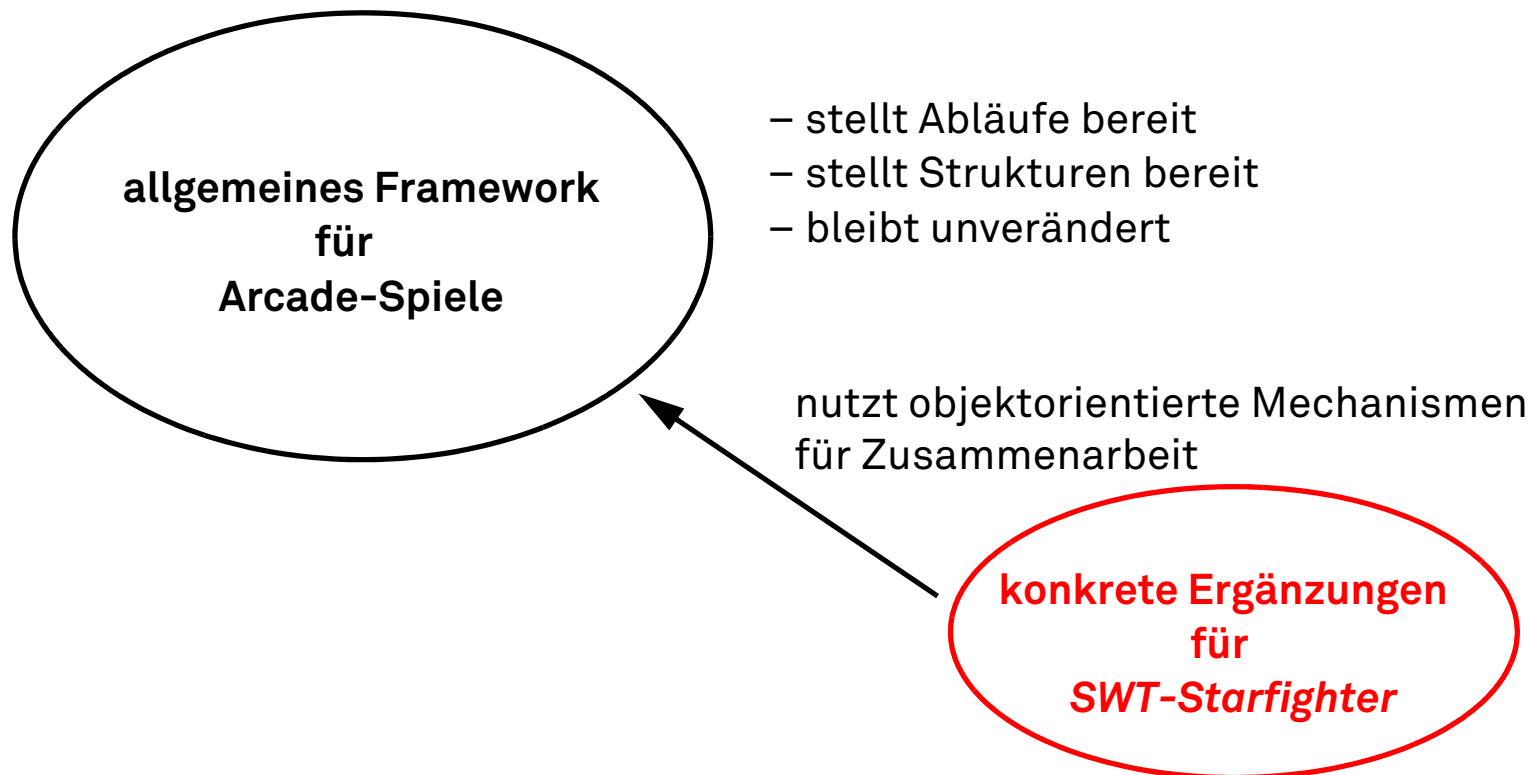
- stellt Abläufe bereit
- stellt Strukturen bereit
- bleibt unverändert

**konkrete Ergänzungen
für
*SWT-Starfighter***

Beispielsoftware SWT-Starfighter

(Fortsetzung)

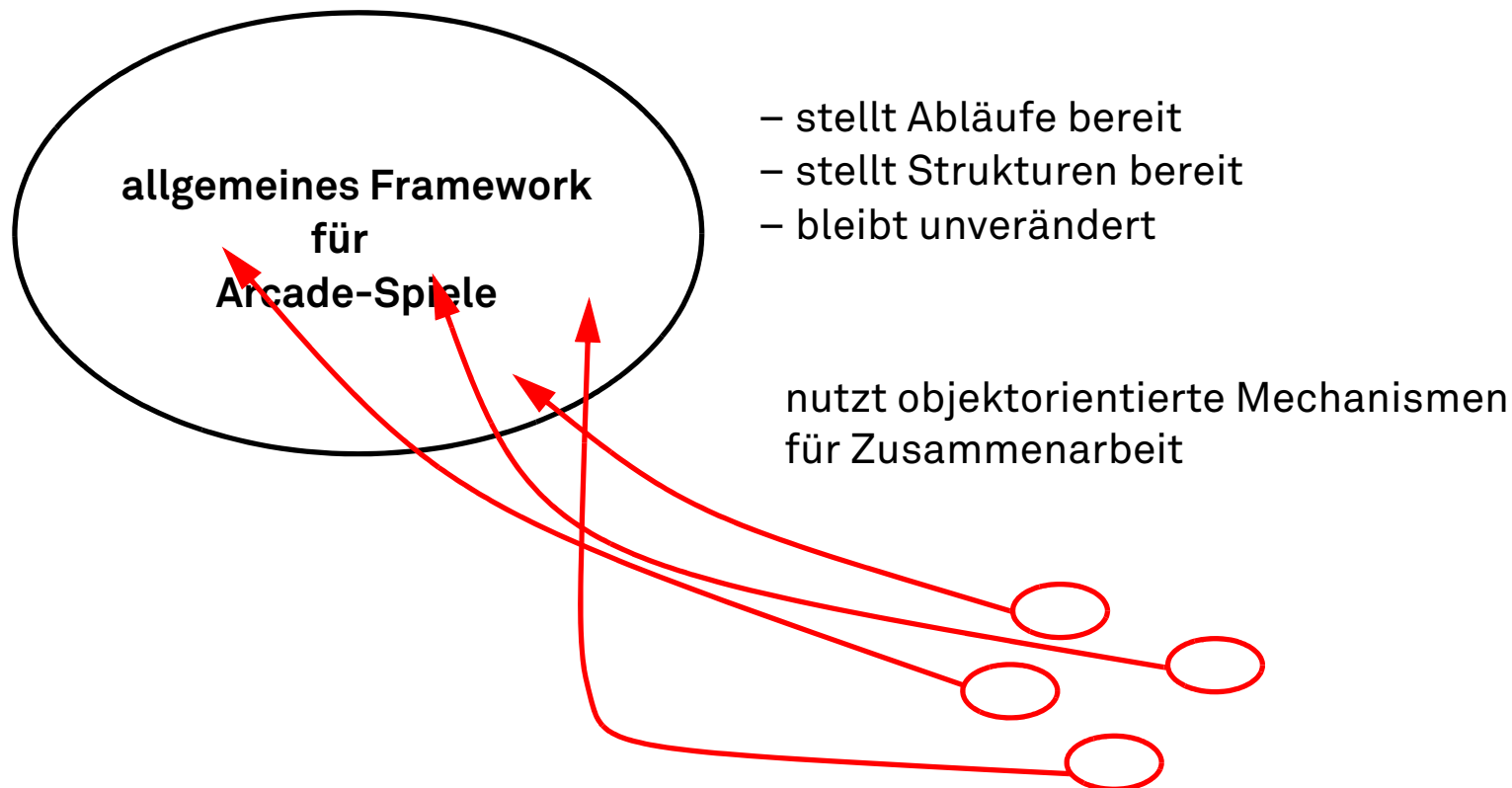
Vorbereitung auf Änderungen/Erweiterungen:



Beispielsoftware SWT-Starfighter

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:



Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

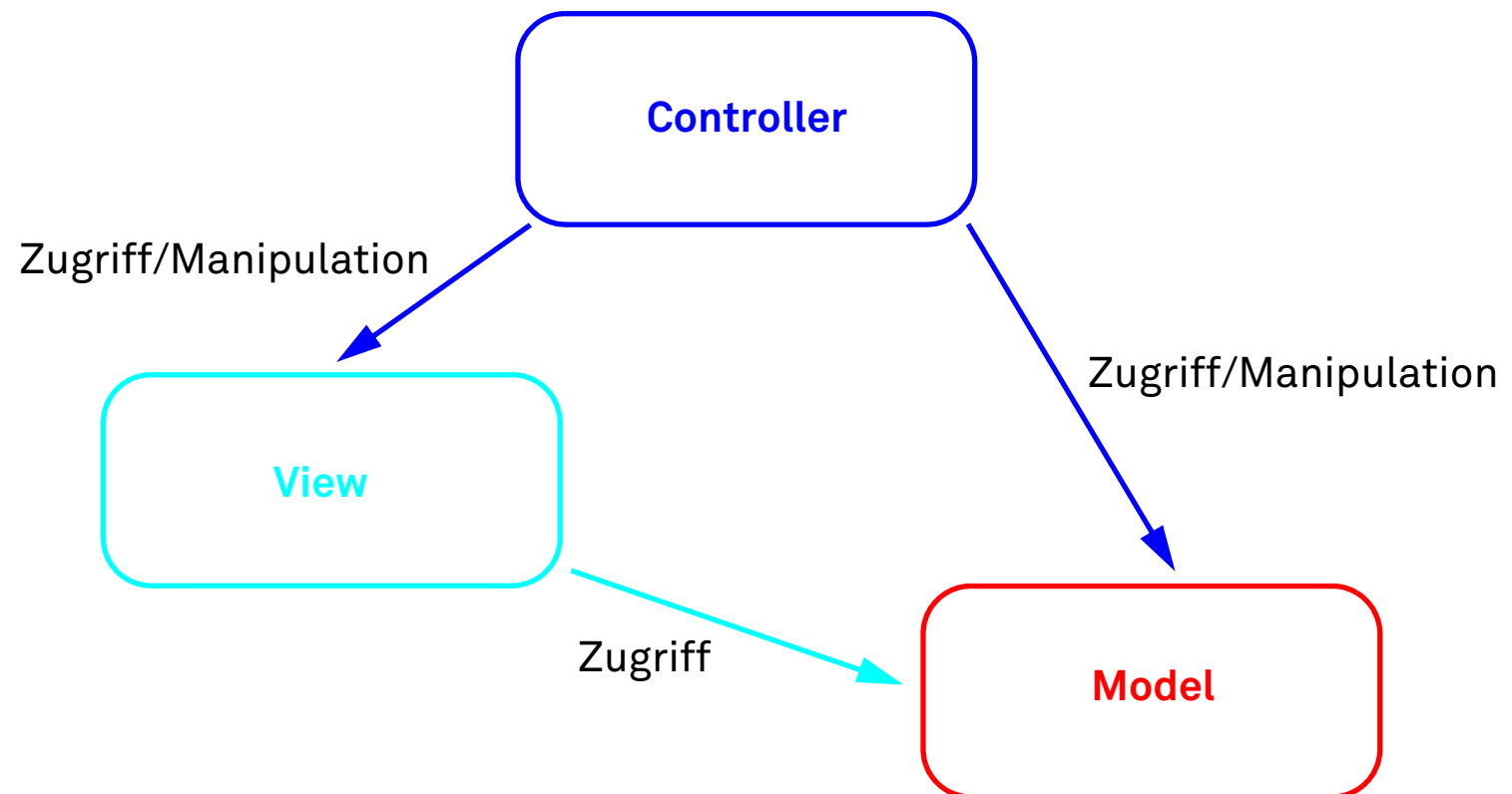
weitere Vorbereitung auf Änderungen/Erweiterung:

Architekturstil ***Model – View – Controller (MVC)***

- ❑ getrennte Behandlung der drei Aufgabenbereiche
 - Model (Datenmodell):
unabhängig von Präsentation und Abläufen
 - View (Präsentation/Benutzungsschnittstelle):
visualisiert Programmzustände, verarbeitet aber keine Daten,
reagiert auf die Änderung von Daten
 - Controller (Steuerung der Abläufe):
verbindet Präsentation und Datenmodell durch Ausführung von Aktionen
- ❑ Die technische Ausgestaltung von MVC kann sehr unterschiedlich erfolgen.

Architekturstil Model - View - Controller (MVC)

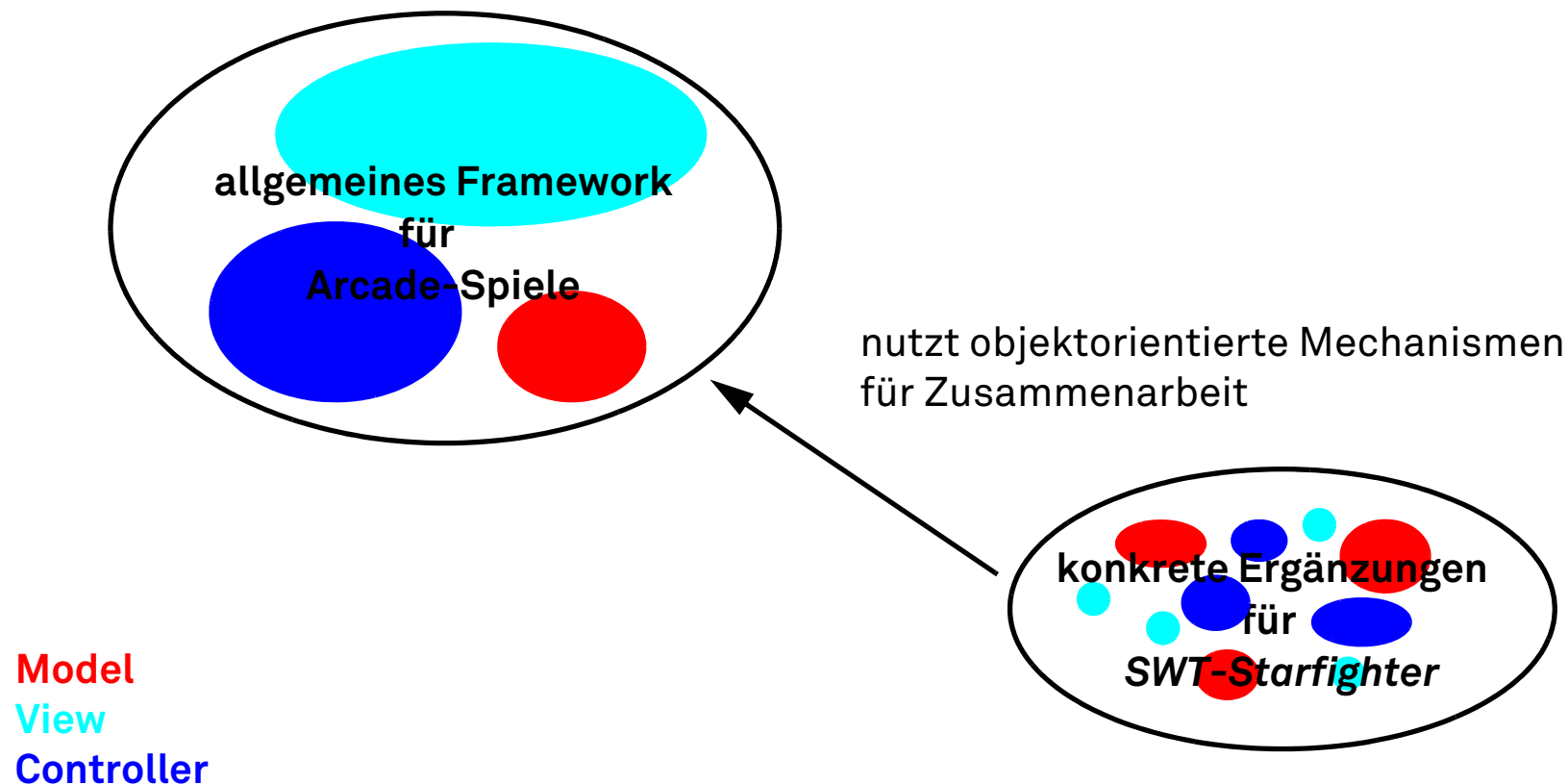
Grundlegende Idee der Zusammenarbeit



Beispielsoftware SWT-Starfighter

(Fortsetzung)

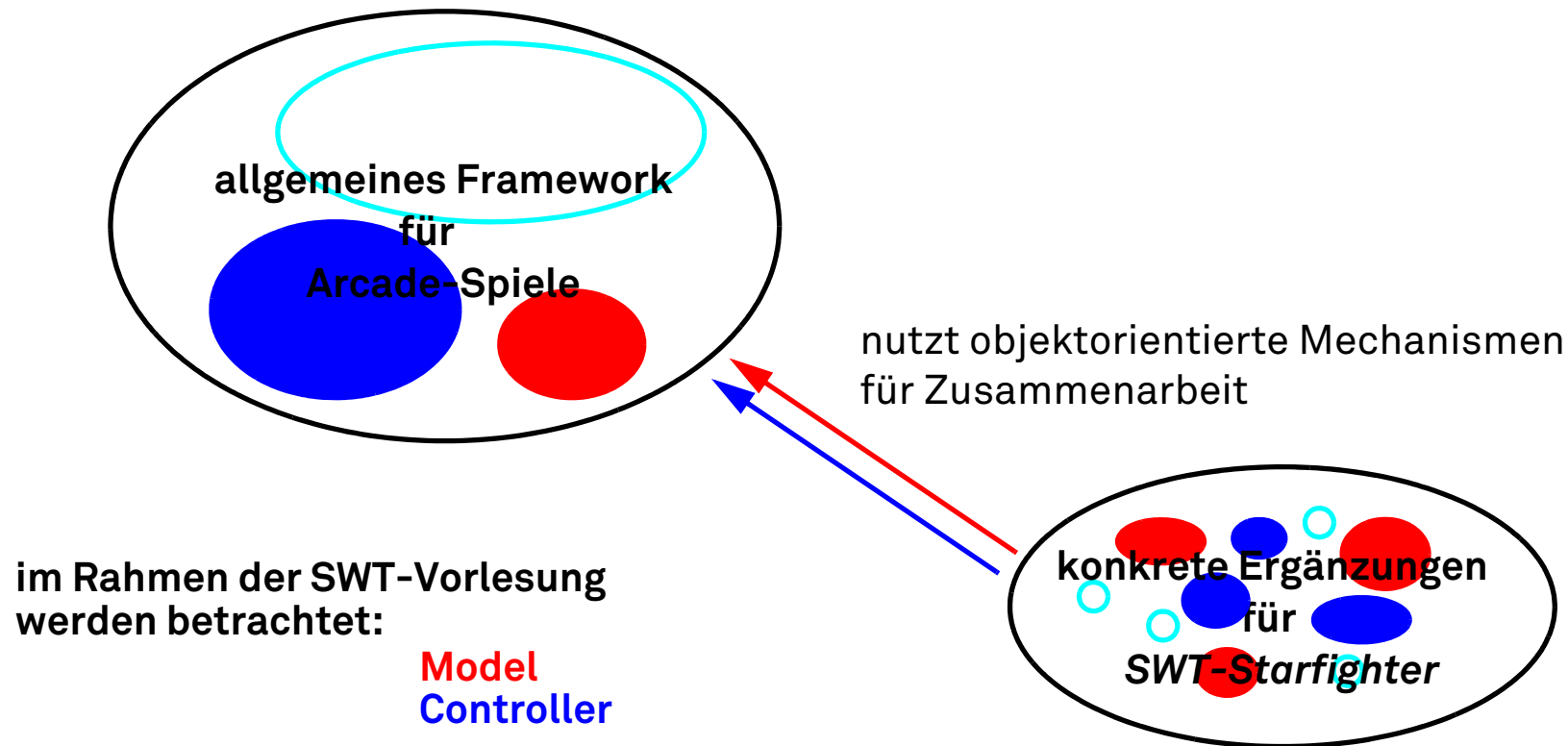
Framework und MVC



Beispielsoftware SWT-Starfighter

(Fortsetzung)

Framework und MVC



Folien zur Vorlesung **Softwaretechnik**

Abschnitt 1.2: Ideen der objektorientierten Softwaregestaltung

Objektorientierte Systeme

Die Arbeitsweise objektorientierter Systeme orientiert sich an der realen Welt:

Lösen einer Aufgabenstellung durch
Zusammenarbeit von handelnden Entitäten (Objekten),
basierend auf Kommunikation (über definierte Nachrichten) mit
Einschränkungen, die durch Regeln (objektorientierte Prinzipien) gegeben sind:

- ❑ **Kapselung**
Jedes Objekt grenzt sich ab und verletzt auch nicht die Grenzen anderer Objekte.
- ❑ **Lokalität**
Daten und zugehörige Handlungen sind in einem Objekt zusammengefasst.
- ❑ **Geheimhaltung**
Jedes Objekt stellt nach außen nur die für seinen Aufgabebewältigung notwendigen Informationen bereit.
- ❑ **Autonomie**
Jedes Objekt entscheidet selbst über die von ihm ausgeführten Handlungen.

Objektorientierte Systeme

(Fortsetzung)

Konsequenzen aus der technischen Umsetzung objektorientierter Systeme:

- ❑ konzeptionelle Vollständigkeit
Jedes (korrekt implementierte) Objekt kann alle geforderten Aufgaben erfüllen.
- ❑ wohldefinierte Leistung
Für jedes Objekt ist sein Leistungsumfang bekannt.
- ❑ wohldefinierte Kommunikation
Für jedes Objekt ist bekannt, in welcher Weise seine Leistungen abgerufen werden können.
- ❑ Teamfähigkeit
Alle Objekte arbeiten immer kooperativ zusammen.

Objektorientierte Systeme

(Fortsetzung)

Anmerkungen:

- ❑ Objektorientierte Programmiersprachen stellen Konzepte bereit, um objektorientierte Systeme implementieren zu können.
- ❑ Objektorientierte Programmiersprachen stellen darüber hinaus weitere Konzepte bereit, um die Konstruktion objektorientierter Systeme zu vereinfachen.
- ❑ Art und Umfang dieser Konzepte unterscheiden sich bei den verschiedenen objektorientierten Programmiersprachen.

Im Rahmen dieser SWT-Vorlesung wird aufgrund der Vorkenntnisse der teilnehmenden Studierenden nur **Java** als Programmiersprache betrachtet.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 1.3: Objektorientierte Konzepte in Java

Paket

- ❑ wesentliches Ziel:
Schaffen einer konzeptionellen Struktur innerhalb einer großen Anzahl von Klassen.
- ❑ weitere Ziele:
Schaffen eines Namensraums, so dass in einem Projekt mehrere Klassen mit gleichem Namen möglich sind.
- ❑ Syntax:
 - Zuordnung zu einem Paket: **package** edu.udo.cs.swtsf.core;
 - Einbeziehen eines Pakets: **import** edu.udo.cs.swtsf.core.player.Player;
 - individueller Zugriff in anderen Paketen: java.util.ArrayList
- ❑ Randbedingung:
Paketname muss der Verzeichnisstruktur entsprechen. (Programmierungsumgebung)
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.core;
import java.util.ArrayList;
...
import edu.udo.cs.swtsf.core.player.Player;
...
public class Game { ... }
```

Klasse

- ❑ wesentliches Ziel:
Definition von Eigenschaften einer Gruppe von gleichförmigen Objekten.
- ❑ weitere Ziele:
Festlegen eines i.W. durch die Methoden der Klasse gegebenen Datentyps.
- ❑ Syntax:
 - Deklaration: **public class** HudElement { ... }
 - Benutzung: HudElement
- ❑ Randbedingung:
Klassenname muss dem Dateinamen entsprechen. (Programmierungsumgebung)
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.view;  
import edu.udo.cs.swtsf.core.Game;  
public class HudElement {  
    ...  
}
```

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
                    int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {                                     Attribute
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
                      int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
                     int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
    { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Konstruktor

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
        int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Methoden

Innere Klasse

- ❑ wesentliches Ziel:
Gemeinsames Verwalten konzeptionell zusammengehörender Klassen.
- ❑ weitere Ziele:
Zugriff auf geschützte Eigenschaften der umgebenden Klasse.

- ❑ Anmerkungen:
 - Innere Klassen führen **nicht** zu "geschachtelten" Objekten.
 - Auch innere Klassen dienen nur zur Definition von Eigenschaften einer Gruppe von gleichförmigen Objekten.

Instanzeigenschaften (Instanzattribute/Instanzmethoden)

- ❑ wesentliche Ziel:
Individualisierung von Objekten.
- ❑ weitere Ziele:
 - Jedes durch einen Konstruktor der Klasse erzeugte Objekt (=Instanz) besitzt eigene Speicherbereich für seine Attribute.
 - Aufrufe von Methoden für ein Objekt beziehen sich auf die für dieses Objekt gespeicherten Attributwerte.
- ❑ Beispiel:

```
public class HudElement {  
    private String text = "";  
    public void setText(String value) {  
        if (value == null) { throw new IllegalArgumentException(); }  
        text = value;  
    }  
    public String getText() {  
        return text;  
    }  
    ...  
}
```

Instanzattribut

statische Eigenschaften (statische Attribute/statische Methoden)

- ❑ wesentliches Ziel:
(Globale) Eigenschaften ohne explizites Erzeugen eines Objekts verfügbar machen.
- ❑ weitere Ziele:
Gemeinsame Attribute für alle Objekte einer Klasse schaffen.
- ❑ Syntax:
Deklaration mit dem Schlüsselwort **static**
- ❑ Randbedingung:
Statische Methoden dürfen nicht auf Instanzattribute und Instanzmethoden zurückgreifen, da kein zugehöriges Objekt existiert.
- ❑ Beispiele:

```
static final String TITLE_TEXT = "SWT - Starfighter";
```

```
public static void main(String[] args){ ... }
```

Vererbung

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der eigene Objekte vorgibt und der Objekte von verschiedenen (Unter-)Klassen zusammenfasst.
- ❑ weitere Ziele:
Weitergeben von Eigenschaften an die Deklarationen der Unterklassen
- ❑ Syntax:
 - in der Deklaration der Unterklasse: **extends**
 - eine Klasse kann sich gegen das "Geerbt-Werden" wehren: **final class**
- ❑ Randbedingung:
Jede Klasse erbt von genau einer Oberklasse.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.example;  
import edu.udo.cs.swtsf.core.Target;  
public class MonsterEasy extends Target {  
    public MonsterEasy() {  
        setMaxHitpoints(2);  
        setSize(32);  
        ...  
    }  
}
```

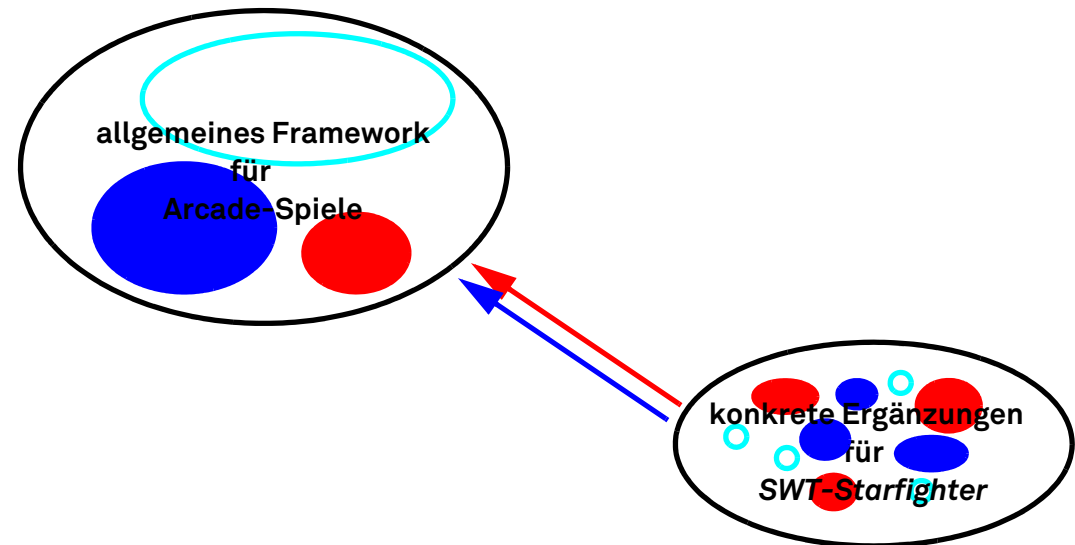
geerbte Methoden




Zugriffsrecht

- ❑ wesentliches Ziel:
Sichtbarkeit von Eigenschaften einschränken, um fehlerhafte Nutzung zu verhindern.
- ❑ Syntax:
 - allgemein sichtbar: **public**
 - im Paket und allen Unterklassen sichtbar: **protected**
 - im Paket sichtbar: *(keine Angabe = package)*
 - nur innerhalb der Klasse sichtbar: **private**
- ❑ Beispiel:

Nutzung von **protected**
im Framework,
um Ergänzungen zu
ermöglichen.



Abstrakte Klasse

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der **keine eigenen Objekte** ermöglicht, der aber Objekte von verschiedenen (Unter-)Klassen zusammenfasst.
- ❑ weitere Ziele:
Weitergeben von Eigenschaften an die Deklarationen der Unterklassen.
- ❑ Syntax:
 - Deklaration der abstrakten Klasse: **abstract**
 - Deklaration abstrakter Methoden: **abstract** 
- ❑ Randbedingungen:
 - Eine abstrakte Methode führt zwangsläufig zu einer abstrakten Klasse.
 - Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- ❑ Beispiel:
public abstract class Target extends Entity { ... }

Target soll immer nur als Oberklasse dienen.
Target enthält keine abstrakten Methoden.

Interface

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der **keine** eigenen Objekte ermöglicht,
der aber Objekte von verschiedenen (Unter-)Klassen oder Interfaces zusammenfasst.
- ❑ weitere Ziele:
(Weitergeben von Methoden; keine Deklaration von Attributen)
- ❑ Syntax:
 - Deklaration: **public interface** Group<E> { ... }
 - Nutzung: **public class** BufferedGroup<E> **implements** Group<E> { ... }
- ❑ **Randbedingungen:**
 - Eine Klasse kann viele Interfaces implementieren.
 - Ein Interface kann von vielen Interfaces erben.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.util;  
public interface Group<E> {  
    public void add(E element);  
    public void remove(E element);  
    public default int count(E element) { ... }  
    ...  
}
```


Polymorphie

- ❑ wesentliches Ziel:
Ausführen der Implementierung einer Methode, die "nah" an der Deklaration der Klasse liegt – unabhängig vom Typ der auf das Objekt verweisenden Referenz.
- ❑ Syntax:
Deklarieren der gleichen Methode in verschiedenen Klassen einer Vererbungshierarchie durch **Überschreiben** der Methode in einer Unterklasse.
- ❑ Randbedingungen:
 - Signaturen (=Name+Parameterliste) der Methoden müssen exakt übereinstimmen.
 - Eine Methode kann sich gegen Überschreiben wehren: **final**
 - Die Auswahl der Signatur erfolgt durch den Compiler.
 - Die Auswahl der ausgeführten Implementierung erfolgt durch das Laufzeitsystem.
- ❑ Anmerkung:
Die geeignete Nutzung von Vererbung, abstrakten Klassen und Interfaces beruht wesentlich auf dem Konzept der Polymorphie.

Generische Klasse/generisches Interface


- ❑ wesentliches Ziel:
Deklaration eines Datentyps, der mit Objekten anderer Klassen typsicher umgehen kann.
- ❑ Syntax:
Deklaration von Typparametern: < >
- ❑ Anmerkungen:
 - Die erlaubten Typargumente können durch die Angabe von beschränkenden Regeln (**super**, **extends**) präzisiert werden.
 - Die technische Umsetzung in Java ist etwas problematisch, um Kompatibilität mit alten Java-Versionen zu erhalten.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.util;
public interface Group<E> {
    public void add(E element);
    public void remove(E element);
    public default int count(E element) { ... }
    ...
}
```

Deklaration des Typparameters

Nutzung des Typparameters

anonyme Klasse

- ❑ wesentliche Ziele:
 - Anlegen einer Klasse und Erzeugen von deren einzigem Objekt in einem Schritt innerhalb einer anderen Klasse.
 - Vermeiden der expliziten Deklaration einer nur genau einmal genutzten Klasse.
- ❑ weitere Ziele:
Zugriff auf finale Eigenschaften der umgebenden Klasse.
- ❑ Syntax:
Deklaration und Erzeugung gemeinsam: **new** Target { ... } 
- ❑ Randbedingung:
Die anonyme Klasse muss eine deklarierte Klasse erweitern oder ein deklariertes Interface implementieren.

Lamda-Ausdruck

- ❑ wesentliche Ziele:
 - Anlegen einer Klasse und Erzeugen von deren einzigem Objekt in einem Schritt innerhalb einer anderen Klasse.
 - Vermeiden der expliziten Deklaration einer nur genau einmal genutzten Klasse.
 - Verbessern der Lesbarkeit/Reduktion des Schreibaufwands.
- ❑ weitere Ziele:
Zugriff auf finale Eigenschaften der umgebenden Klasse.
- ❑ Syntax:
Parameterliste -> Methodenrumpf
- ❑ Randbedingungen:
Compiler erwartet ein Objekt eines Interfaces mit genau einer abstrakten Methode.
- ❑ Vorgegebene Deklaration:

```
public interface BulletHitStrategy {  
    public void onHit(Bullet host, Target target);  
}
```

- ❑ Beispiel:

```
public static final BulletHitStrategy BULLET_DAMAGE_ON_HIT =  
(bullet, target) -> { target.addHitpoints(-bullet.getDamage());};
```

Enumeration

- ❑ wesentliches Ziel:
Anlegen eines Typs mit endlich vielen vorgegebenen Werten.
- ❑ Syntax:
Deklaration mit: **enum**
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.view;  
public enum HudElementOrientation {  
    TOP, BOTTOM;  
}
```



Zusammenarbeit von Objekten

- ❑ wesentliches Ziel:
Bereitsstellen der Leistung des Softwarersystems.
- ❑ Syntax:
Aufruf der Methoden von erreichbaren Objekten mit der Angabe von Argumenten für die Parameter: . - Notation (Dereferenzierung)
- ❑ Beispiel:

```
target.addHitpoints(-bullet.getDamage());
```

Referenz auf Objekt

Reflection/Introspection

- ❑ wesentliches Ziel:
 - Untersuchung eines Programms durch sich selbst während der Ausführung.
 - Eventuell auch Manipulation der festgestellten Eigenschaften.
- ❑ Syntax:
In Java möglich durch Nutzung der Klasse **Class**.

```
public class Class<T> {  
    public String getName() { ... }  
    public Class<? super T> getSuperclass() { ... }  
    public Class<?>[] getInterfaces() { ... }  
    public Method[] getMethods() { ... }  
    public Method[] getDeclaredMethods() { ... }  
    public Field[] getFields() { ... }  
    public Field[] getDeclaredFields() { ... }  
    public boolean isAnonymousClass() { ... }  
    ...  
}
```