# Inhalt






Motivation

Unsere Ansprüche und unsere Probleme

Design patterns

Kernbereiche

Creational patterns

Factory method
Abstract Factory
Builder

Structural patterns

Adapter
Facade
Decorator

Behavioral patterns

Visitor
Strategy
Command
Weitere Themen

Gute OOP-Programmierung ist *schwer*

- Objekte $\neq$ Objekte, wird aber so vermittelt (siehe jede Einführung in OOP)
- Ansprüche an gutes Design diametral—welche Ansprüche haben wir überhaupt?
- Fehler im Design werden oft erst sehr spät offensichtlich

$\Rightarrow$ Wir brauchen gute *Heuristiken*

Wir hätten gerne Code, der...

1. ...funktioniert
2. ...verständlich ist
3. ...erweiterbar ist

und das *langfristig*.

Wir müssen Code schreiben, der...

1. ...schnell geschrieben ist
2. ...ein schlecht definiertes Problem löst
3. ...mit fremdem Code interagiert

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (“Gang of Four”, GoF)
- Design pattern: allgemeine Lösungen für häufig auftretende Probleme
- Ursprüngliche Idee aus der Architektur (Christopher Alexander)
- Pattern language: Bibliothek von Patterns (hier: Softwaretechnik)

## Creational pattern

Hilft bei der Erstellung von Objekten

## Structural pattern

Vereinheitlicht Code. Hilft, *encapsulation* zu erhalten.

## Behavioral pattern

Betreffen das Zusammenspiel von Objekten.

| | |
|---|---|
| Situation: | Polymorphie mit schöner Vererbungshierachie |
| Problem: | Für $new()$ müssen wir immer noch explizit die konkrete Klasse kennen |
| Lösung: | Klassen sollten selbst für ihre Konstruktion verantwortlich sein! |

```
class Unit {
    public static Unit create( String name, Point position ) {
        if( name.equals( "Space marine" ) ) {
            return new SpaceMarine( position );
        } else if ( name.equals( "Tank" ) ) {
            ...
        }
        assert false : "Unknown unit " + name;
    }
}
```

Situation: Eine Gruppe von Objekten soll austauschbar gemacht werden

Problem: Für $new()$ müssen wir immer noch explizit die konkrete Klasse kennen

Lösung: Eine neue abstrakte Klasse, die für die Konstruktion verantwortlich ist!

```
...
Unit soldier;
if( gameMode == GAME_SPACE ) {
    soldier = new SpaceMarine();
} else if ( gameMode = GAME_MEDIEVAL ) {
    solider = new Knight();
}
...
```

```
interface UnitFactory {
    Unit createSoldier();
    Unit createTank();
    ...
}
...
Unit soldier = unitFactory.createSoldier();
```

```
interface UnitFactory {
    Unit createSoldier();
    Unit createTank();
    ...
}

class SpaceUnitFactory extends UnitFactory {
    public Unit createSolider() {
        return new SpaceMarine();
    }

    public Unit createTank() {
        return new SpaceTank();
    }
}

class MedievalUnitFactory extends UnitFactory {
    public Unit createSolider() {
        return new Knight();
    }

    public Unit createTank() {
        return new Cannon();
    }
}
```

Situation: Objekte mit vielen Variablen
Problem: Zu viele Konstruktoren!
Lösung: Klassen sollten selbst für ihre Konstruktion verantwortlich sein!

(Ein echter Builder ist noch etwas komplizierter)

```
class SpaceMarine extends Unit {
    private boolean hasStimpackUpgrade;
    private boolean hasHEMunition;
    private int health;
    ...
    public SpaceMarine() { ... }
    public SpaceMarine( Point position ) { ... }
    public SpaceMarine( Point position, boolean hasStimpack ) { ... }
    ...
}
```

Etwas besser mit setter/getter:

```
SpaceMarine marine = new SpaceMarine();
marine.setPosition( spawnPosition );
marine.setHealth( MAX_MARINE_HEALTH );
if( upgrades.isAvailable(UPGRADES_STIMPACK) ) {
    marine.setStimpack(true);
}
...
```

# Unter Umständen noch besser:

```
class SpaceMarineBuilder {
    private boolean hasStimpackUpgrade = false;
    private boolean hasHEMunition = false;
    private int health = MAX_MARINE_HEALTH;
    ...
    public SpaceMarineBuilder setHealth( int health ) {
        this.health = health;
        return this;
    }

    public SpaceMarineBuilder setStimpack( boolean stimpack ) {
        ...
    }

    public SpaceMarine build() {
        SpaceMarine res = new SpaceMarine() ;
        res.setHealth( health );
        ...
        return res;
    }
}

...
Unit marine = new SpaceMarineBuilder().setStimpack( true ).build();
```

Situation: Wiederbenutzung alten Codes

Problem: Interfaces/Contracts passen nicht in neues Konzept, alter Code soll/kann nicht verändert werden

Lösung: Was nicht passt,...

```
class OldDisplayLibrary {
    ...
    public void drawRect( int x, int y, int width, int height ) { ... }
}

class DisplayLibrary {
    private OldDisplayLibrary lib;

    public void drawRect( Point start, Point end ) {
        lib.drawRect( start.x, start.y, end.x−start.x, end.y−start.y );
    }
}
```

Situation: Großes Projekt, viel Code

Problem: Einfache Aufgaben benötigen überproportional viel Code

Lösung: Verstecken wir den Kram.

```
interface NetworkSessionFacade {
    void moveUnit( int id, Point position );
    void sendMessage( int playerId, String text );
}

class ServerSession implements NetworkFacade {
    ...
    public void moveUnit( int id, Point position ) {
        if( !socket.isConnected() )
            throw new NetworkException();
        DataOutputStream out = new DataOutputStream(
                                        socket.getOutputStream());
        out.write( MSG_TYPE_MOVEMENT );
        out.write(":"+id);
        out.write(":"+position.x+":"+position.y );
        out.flush();
        ...
    }
}

class P2PSession implements NetworkFacade { ... }
```

Situation: Großes Projekt, viel Code
Problem: Klasse soll erweitert werden, aber *alte* Klasse soll weiterhin verfügbar sein
Lösung: Dekorieren.

```
class Unit {
    private int health;
    ...
    public void receiveDamage( int damage ) {
        health -= damage;
        if( health <= 0 )
            die();
    }
}

class InvincibleUnit extends Unit {
    private Unit decoratedUnit;

    public InvincibleUnit( Unit unit ) {
        this.decoratedUnit = unit;
    }

    ...
    @Override
    public void receiveDamage( int damage ) {}
}
```

| | |
|---|---|
| Situation: | Ein neues Feature soll eingeführt werden, aber es passt nicht in die Klassenhierarchie |
| Problem: | Wir können es "von außen" einführen, aber dazu benötigen wir häßliche Dinge wie $instanceof$ |
| Lösung: | Indirection |

```
interface UnitVisitor {
    void visitTank( Tank tank );
    void visitSpaceMarine( SpaceMarine marine );
void visitMedic( Medic medic );
}

abstract class Unit {
    ...
public abstract void accept( UnitVisitor visitor );
}

class Tank extends Unit {
    ...
    @Override
    public void accept( UnitVisitor visitor ) {
        visitor.visitTank( this );
    }
}
```

```
class ArmyStrengthVisitor implements UnitVisitor {
    private int strength = 0;

    public void visitTank( Tank tank ) {
        if( tank.hasFraggUpgrade() )
            strength += 8;
        else
            strength += 4;
    }

    public void visitSpaceMarine( SpaceMarine marine ) {
        strength += 1;
    }

    public void visitMedic( Medic medic ) { }
}
```

| | |
|---|---|
| Situation: | Redundanter Code |
| Problem: | Ähnliche Algorithmen, aber Details verhindern gemeinsamen Code |
| Lösung: | Details in Klasse auslagern |

Beliebtes Beispiel ist Vergleichsbasiertes sortieren mit $compare(...)$

```
class SpaceMarine extends Unit {
    private Weapon weapon;
    ...
    public void attack( Unit other ) {
        if( weapon.inRange( getPosition(), other.getPosition() )
            weapon.shoot( getPosition(), other );
    }
}

interface Weapon {
    boolean inRange( Point source, Point target );
    void shoot( Unit other );
}
```

```
class PlasmaRifle implements Weapon {
    public boolean inRange( Point source, Point target ) {
        return source.distanceTo( target ) < 120.0;
    }

    public void shoot( Unit other ) {
        if( Math.random() < 0.8 )
            other.receiveDamage( 7.5 );
    }
}

class RPG implements Weapon {
    public boolean inRange( Point source, Point target ) {
        return true;
    }

    public void shoot( Unit other ) {
        if( other.isVehicle() )
        other.receiveDamage( 50 );
        else
            other.reciveDamage( 15 );
    }
}
```

Situation: GUI-Programmierung
Problem: Viele Möglichkeiten, das gleiche zu tun: Menüs, Buttons, Shortcuts...
Lösung: Befehle in Klassen auslagern

```
abstract class UnitCommand {
    protected Unit unit;
    ...
    public abstract void execute();
}

class MoveUnitCommand extends UnitCommand {
    private Point target;

    public void execute() {
        unit.moveTo( target );
    }
}
```

- Antipatterns (Singleton!)
- Code smells
- Refactoring, Refactoring to Patterns
- Model-View-Controller