

Eine schematische Lösung für ein häufig wiederkehrendes Problem  
schematisch:

- kein Code
- kein Algorithmus
- sondern Skizzierung einer Lösung durch beliebige Objekte und ihre Beziehungen

"Trickkiste" des erfahrenen Programmierers,

## Factory Pattern

Beispiel

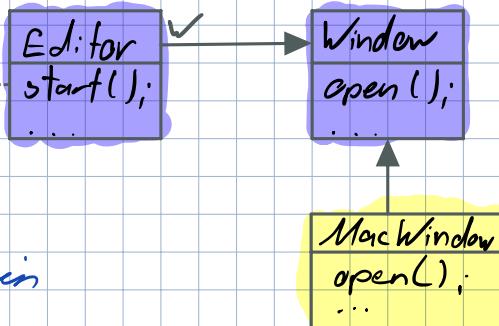
```
w = new Window();  
w.open(...);  
...
```



Problem

- Editor ist auf Windows festgelegt
- Wie bringt man den Editor dazu, mit neuen Fenstertypen (z.B. MacWindows) zu arbeiten?

```
w = new Window();  
w.open(...);  
...
```

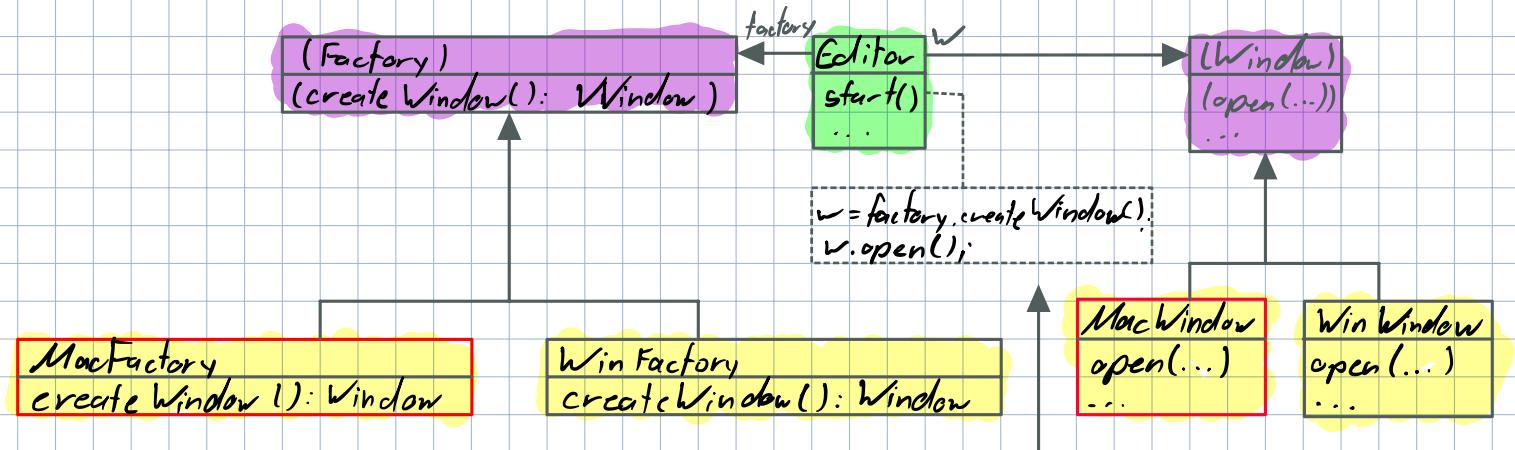


Erzeugt nach wie vor ein  
Windows - Objekt!

Man müsste den Code der start - Methode ändern  
• oft nicht möglich  
• unheimlich fehleranfällig

Zweck Erzeugen eines Objekts mit beschränkter Schnittstelle aber mit variablen dynamischen Typs

Lösung Objekt nicht mit new erzeugen, sondern von einem "Fabrikobjekt" anfordern



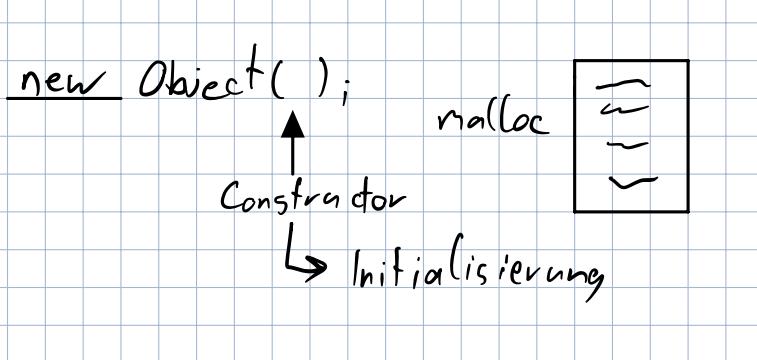
Initialisierung am Programmbeginn

factory = new MacFactory();

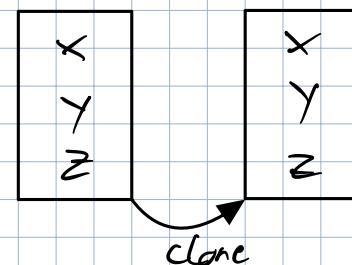
lässt dynamischen Typ offen:

garantiert nur, dass irgendeine Windows - Variante geliefert wird

## Prototype Pattern



Zwecke Möglichkeit:



Das Prototype Pattern ist ein Entwurfsmuster, das verwendet wird, um neue Objekte zu erstellen, indem es existierende Objekte kopiert. Es wird eine Vorlage verwendet, die als Prototyp bezeichnet wird, auf die das neue Objekt basiert.

## Shallow Copy von Objekten

class Figure implements Clonable

```
int x, y;  
String name;  
public Figure (int x, int y, String name) { ... }  
public Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}  
...  
}
```

class Circle extends Figure {

```
Color color;  
public Circle (int x, int y, String name, Color c) { ... }  
...  
}
```

Figure f = new Circle(5, 5, "circle", red);

Figure g = (Figure) f.clone();

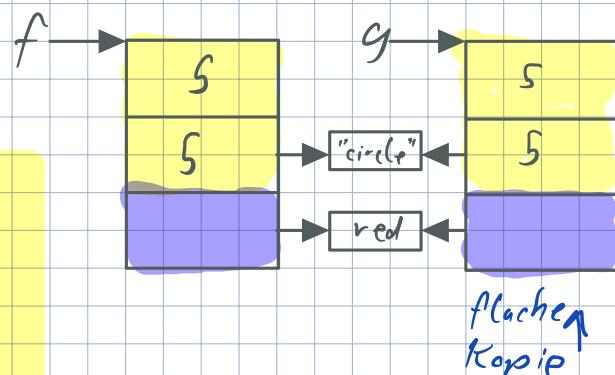
## Deep Copy von Objekten

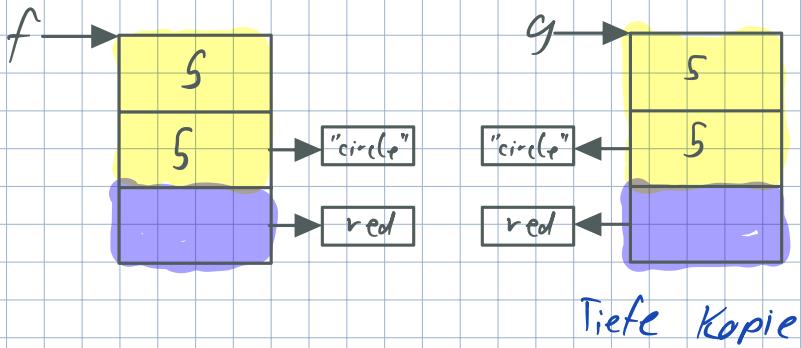
class Figure implements Clonable {

```
int x, y;  
String name;  
public Object clone() throws ... {  
    Figure f = (Figure) super.clone();  
    f.name = new String(name);  
    return f;  
}  
...  
}
```

class Circle extends Figure {

```
Color color;  
public Object clone() throws ... {  
    Circle c = (Circle) super.clone();  
    c.color = (Color) color.clone();  
    return c;  
}  
...  
}
```



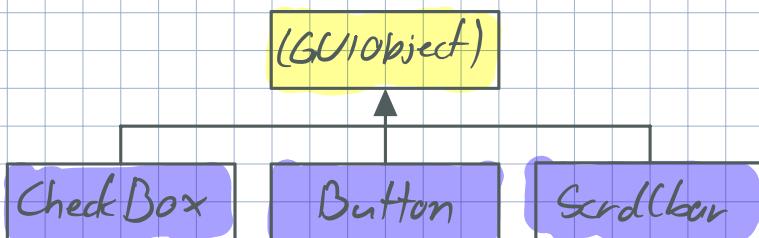


## Familie

Zweck Zusammenfassung von Varianten

Lösung Abstrakte Klasse mit ihren Unterklassen

Beispiel:



- Alle Familienmitglieder haben die gleiche Schnittstelle
- jedes Familienmitglied kann als GUIMObject verwendet werden

## Adapter / Wrapper

Zweck Fremde Klasse zu einer bestehenden Familie kompatibel machen

Lösung: Neues Familienmitglied anlegen, das Familiennachrichten in Nachrichten an die fremde Klasse umsetzt (Forwarding)

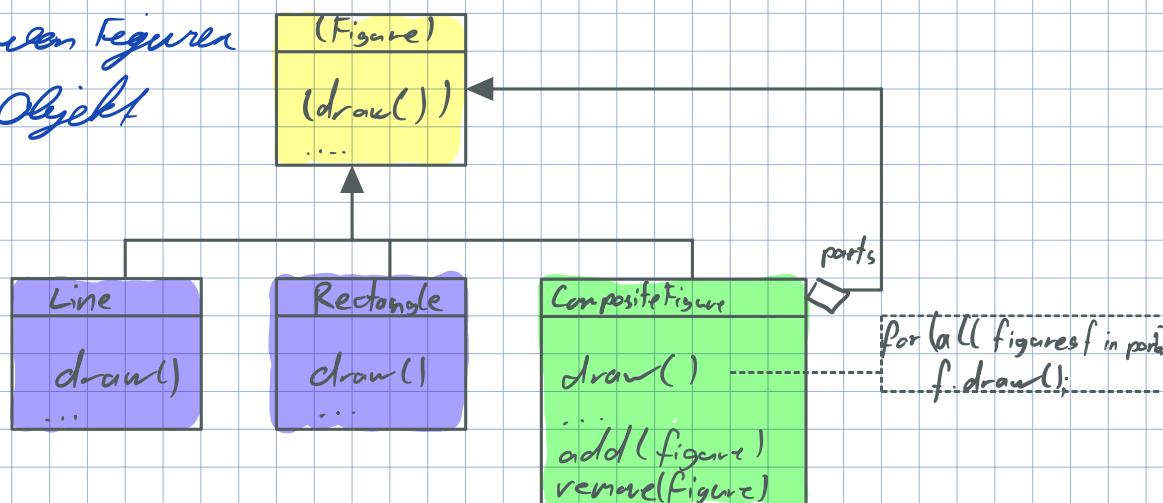
## Composite Pattern

Zweck: Zusammengesetztes Objekt, das wie ein Einzeldobjekt behandelt werden kann

Lösung: Zusammengesetztes Objekt ist Mitglied der Familie

Beispiel

Gruppieren von Figuren  
in einem Objekt



Alle Figure-Operationen sind auch auf Figurengruppen anwendbar.

Oft ist auch ein parent-zeiger von den Komponenten zum Kompositum nötiglich

## Decorator Pattern

Zweck: Neues Verhalten zu einer Klasse hinzufügen, ohne die Klasse zu ändern und ohne eine Unterklasse zu erländern.

```
abstract class KaffeeDecorator extends Kaffee {
```

```
protected Kaffee kaffee
public KaffeeDecorator (Kaffee kaffee) {
```

```
    this.kaffee = kaffee;
```

```
} public abstract double get Preis();
```

```
abstract class Kaffee {  
    public abstract double getPrice();  
}
```

```
class Espresso extends Kaffee {  
    public double getPrice();  
    {  
        return 1.99;  
    }  
}
```

```
class Milchdecorator extends KaffeeDecorator {  
    public Milchdecorator (Kaffee kaffe) {  
        super(kaffe);  
    }  
    public double getPrice() {  
        return kaffe.getPrice() * 0.25;  
    }  
}
```

```
public class DecoratorExample {  
    public static void main (String[] args) {  
        Kaffee meinKaffee = new ZuckerDecorator(new MilchDecorator(new Espresso()));  
        System.out.println(meinKaffee.getPrice());  
    }  
}
```

## Proxy-Pattern

```
public interface Image {  
    /**  
     * Display the image  
     */  
    public void displayImage();  
}
```

```
public class RealImage implements Image {  
    private String filename;  
    public RealImage (String filename) {  
        this.filename = filename;  
        loadImageFromDisk();  
    }  
    private void loadImageFromDisk () {  
        // Potentially expensive operation  
        System.out.println("Loading..." + filename);  
    }  
    public void displayImage () {  
        System.out.println("Displaying " + filename);  
    }  
}
```

```

private class ProxyImage implements Image
{
    private String filename;
    private Image image;

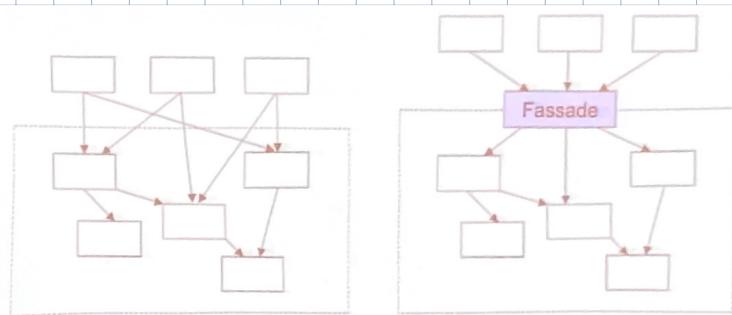
    public Proxy Image (String filename)
    {
        this.filename = filename;
    }

    public void displayImage()
    {
        if (image == null)
        {
            image = new RealImage(filename), // load only on demand
        }
        image.displayImage();
    }
}

```

### Fassade

zweck Ansprechpartner für ein Subsystem sein  
 Lösung künstliches Objekt als Vermittler zwischen Benutzern und Subsystemen



### Beispiel

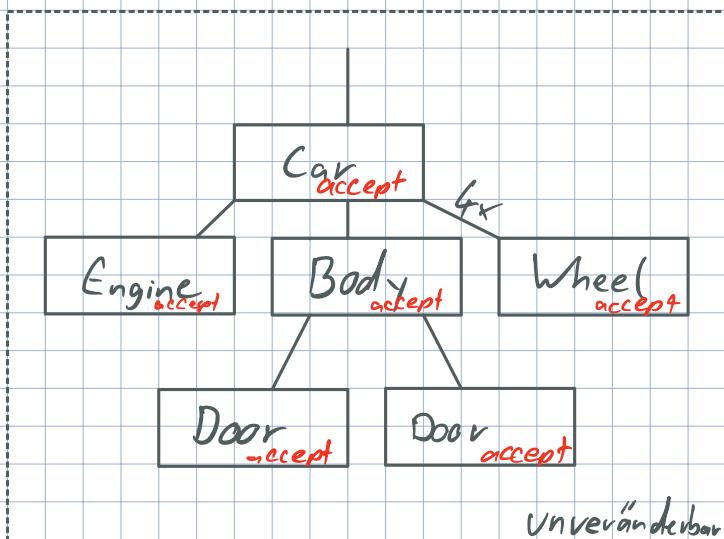
- Schnittstelle zu einem Statistikpaket: Benutzer kann Statistikknoten über eine einheitliche Schnittstelle verwenden

### Vorteile

- Benutzer müssen die Einzelkette des Systems nicht verstehen
- Fassade kann maßgeschneiderte Funktionen für verschiedene Benutzer anbieten
- Entkopplung kann sich ändern, ohne dass sich die Fassade ändert

Iterator Pattern => bekannt

Visitor Pattern



- Gewicht berechnen
- Bildschirm zeichnen
- Ausgeben
- Preis berechnen

```

interface Visitor {
    1. usage 2. implementation
    void visit(Wheel wheel);
    1. usage 2. implementation
    void visit(Engine engine);
    1. usage 2. implementation
    void visit(Body body);
    1. usage 2. implementation
    void visit(Car car);
}
  
```

}

```

class Engine {
    void accept(Visitor visitor) {
        visitor.visitEngine(this);
    }
}
  
```

}

class PrintVisitor implements Visitor {

```

public void visitWheel(Wheel wheel) {
    sout("Visiting " + wheel.getName() + " wheel");
}
  
```

```

public void visitEngine(Engine engine) {
    sout("Visiting engine");
}
  
```

```

public void visitBody(Body body) {
    sout("Visiting body");
}
  
```

```

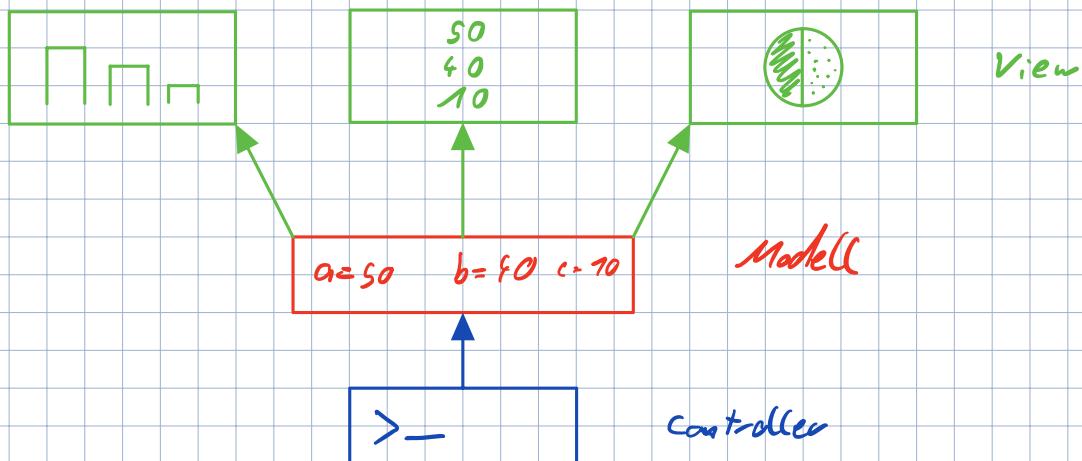
public void visitCar(Car car) {
}
  
```

```

    sout("Visiting car");
    car.body.accept(this);
    for(int i=0; i<car.wheels.length; ++i)
        car.wheels[i].accept(this);
    car.engine.accept(this);
}
  
```

}

## Model View Controller (MVC) / Observer Pattern



Sobald das Modell geändert wird, werden alle Sichten benachrichtigt  
Zweck: alle Sichten sind immer gleich

- Sichten melden sich beim Modell als Beobachter an
- Modell benachrichtigt zur gewünschten Zeit alle seine Beobachter

