OO Ontwerpen II

Design patterns

Iemand heeft onze problemen al opgelost. We leren waarom en hoe we de kennis en ervaring kunnen inzetten van andere ontwikkelaars die dezelfde weg bij het ontwerpprobleem hebben afgelegd en de trip hebben overleefd.

- De beste manier om patterns te gebruiken is ze uit het hoofd te leren, om vervolgens de plaatsen waar we ze kunnen toepassen te herkennen in onze ontwerpen en in bestaande applicaties.
- We krijgen hergebruik van ervaring

> De enige constante die je altijd tegenkomt is **VERANDERING**

Soorten patterns:

- Strategy Pattern
- Simple Factory
- Decorator Pattern
- Observer Pattern
- Façade Pattern
- State Pattern
- MVC Pattern

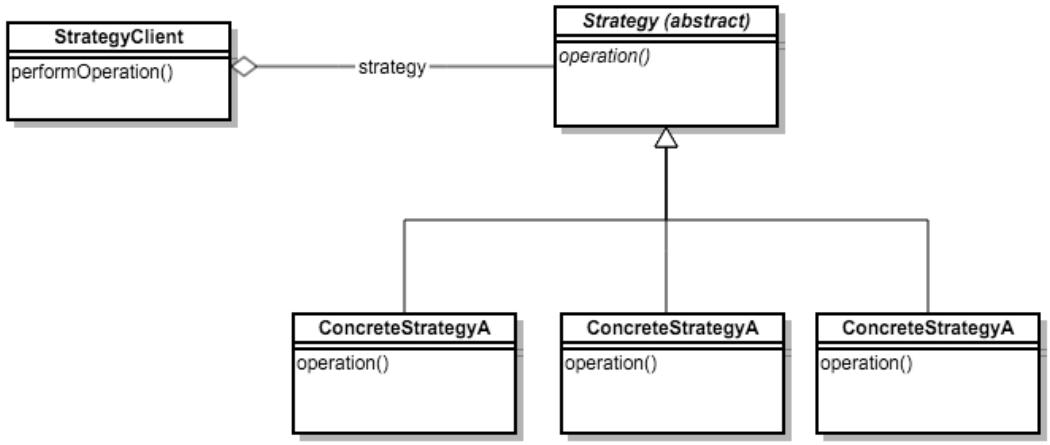
1. Strategy Pattern

Voorbeeld: Eenden
Gedrag van objecten.

1.1. DEFINITIE

> **Het Strategy Pattern** definieert een familie algoritmen, isoleert ze en maakt ze uitwisselbaar. Strategy maakt het mogelijk om het algoritme los van de client die deze gebruikt, te veranderen.

1.2. UML Diagram



(*Note:* De dubbele lijn die je in de diagrammen zit moet niet, dat is gewoon de plaats waar de attributen zouden komen...)

1.3. ONTWERPRINCIPE

> Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven.

M.a.w bevat onze code een aspect dat verandert, bijvoorbeeld met ieder nieuwe systeemeis, dan weten we dat we een gedrag hebben dat eruit gelicht moet worden en moet worden afgezonderd van alle code die niet verandert.

1.4. TWEEDE ONTWERPPRINCIPE

> Programmeer naar een interface, niet naar een implementatie.

Zo kan je er voor zorgen dat je geen sterke koppeling krijgt.

1.5. DERDE ONTWERPPRINCIPE

> Geef een compositie de voorkeur boven overerving.

```
class SomeClass extends OtherClass // Extends = overerving
{
    private SpecialClass specialClass; // Compositie
}
```

1.6. CODE

```
/**
 * Abstract Class
 */
public abstract class Duck
{
    private QuackBehavior quackBehavior;
    private FlyBehavior flyBehavior;

    public void setFlyBehavior(FlyBehavior flyBehavior)
    {
        this.flyBehavior = flyBehavior;
    }

    public void setQuackBehavior(QuackBehavior quackBehavior)
    {
        this.quackBehavior = quackBehavior;
    }

    public String performQuack()
    {
        return quackBehavior.quack();
    }

    public String performFly()
    {
        return flyBehavior.fly();
    }
}

/**
 * Interfaces
 */
public interface FlyBehavior
```

```
{
    public String fly();
}

public interface QuackBehavior
{
    public String quack();
}

/**
 * Concrete Implementations
 */
public class FlyNoWay implements FlyBehavior
{
    public String fly()
    {
        return "Ik kan niet vliegen ze dude..";
    }
}

public class FlyWithWings implements FlyBehavior
{
    public String fly()
    {
        return "Zonder vleugelkes ist wel moeilijk ze..";
    }
}

public class LoudQuack implements QuackBehavior
{
    public String quack()
    {
        return "QUACK QUACK QUACK";
    }
}

public class SilentQuack implements QuackBehavior
{
    public String quack()
    {
        return "quack... shhht... quack... shht...";
    }
}

public class MallardDuck extends Duck
{
    public MallardDuck()
    {
        setQuackBehavior(new LoudQuack());
        setFlyBehavior(new FlyWithWings());
    }

    public String display()
    {
```

```
        return "Ik ben een echte wilde eend";
    }
}

public class WeirdDuck extends Duck
{
    public WeirdDuck()
    {
        setQuackBehavior(new SilentQuack());
        setFlyBehavior(new FlyNoWay());
    }

    public String display()
    {
        return "Ik ben een rare eend";
    }
}

/**
 * Objects
 */
Duck duck1 = new MallardDuck();
System.out.println(duck1.peformQuack());

Duck duck2 = new WeirdDuck();
System.out.println(duck2.peformQuack());
```

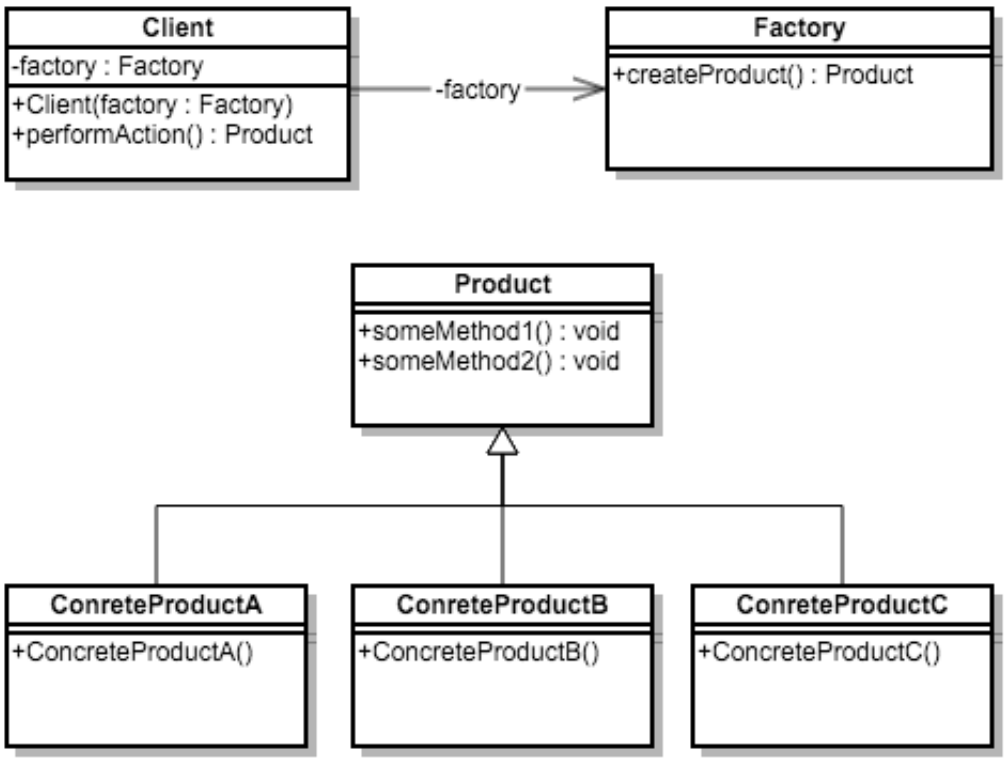
2. Simple Factory Pattern

Voorbeeld: Pizzeria
Creëren van klassen.

2.1. DEFINITIE

> We nemen de code voor de creatie op en verplaatsen deze naar een ander object dat alleen maar het maken van pizza's als taak zal hebben. Dit object noemen we **Factory**.
>
> Wanneer een methode, een klasse retourneert uit verschillende klassen, met een gelijkaardige superklasse.

2.2. UML DIAGRAM



De Client van de Factory gaat naar de Factory om een instantie van een product te maken.
In de Factory bevindt zich een methode, de creatie-methode. Die creatie-methode bevat vaak een parameter, zoals een type (String), waarvan de factory een instantie zal maken. Daarna geeft de Factory de aangemaakte instantie terug aan de Client. De client kan nu allerlei bewerkingen op deze instantie uitvoeren.

2.3. CODE

```
public class PizzaFactory
{
    public Pizza createPizza(String type)
    {
        switch(type.toLowerCase())
        {
            case "cheese": return new CheesePizza();
            case "pepperoni": return new PepperoniPizza();
            case "clam": return new ClamPizza();
            case "veggie": return new VeggiePizza();
            default: return null;
        }
    }
}

public class PizzaStore
{
    private PizzaFactory factory;

    public PizzaStore(PizzaFactory factory)
    {
        this.factory = factory;
    }

    public Pizza orderPizza(String type)
    {
        Pizza pizza = factory.createPizza(type);

        if (pizza != null)
        {
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
            // ...
        }

        return pizza;
    }
}

/**
 * Objecten
 */
PizzaStore store = new PizzaStore(new PizzaFactory());
store.orderPizza("cheese");
```

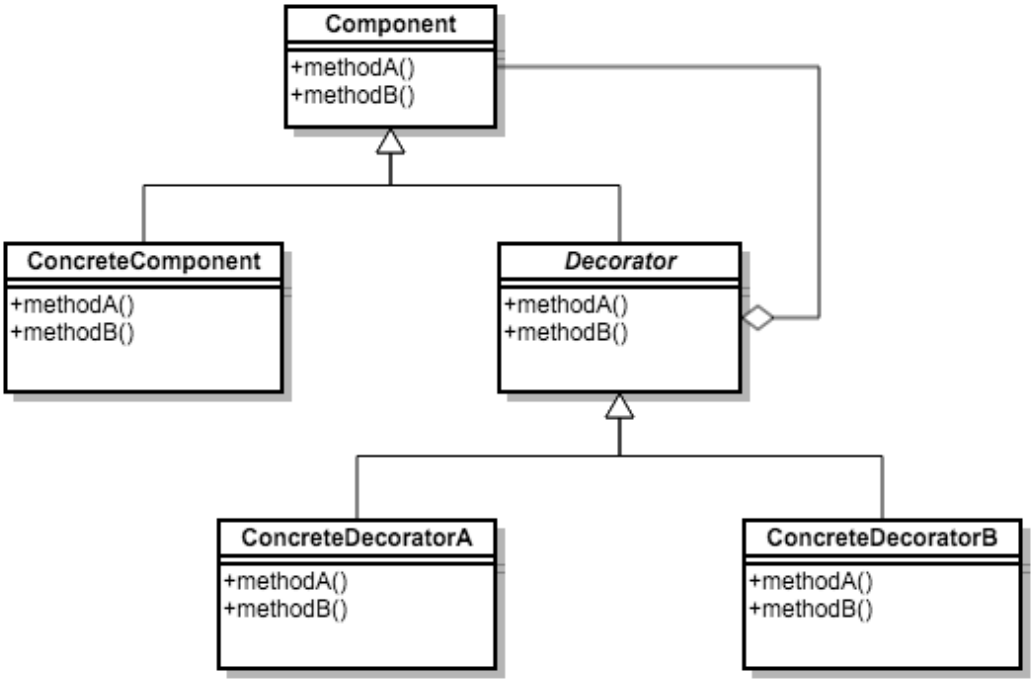
3. Decorator Pattern

Voorbeeld in de cursus: Koffie

3.1. DEFINITIE

> Het **Decorator Pattern** kent dynamisch additionele verantwoordelijkheden toe aan een object. Decorators bieden een flexibel alternatief voor het gebruik van subklassen om functionaliteiten uit te breiden.

3.2. UML DIAGRAM



3.3. OCP (Open Closed Principle)

Dit is een van de principes van de **SOLID** principles.

> Klassen moeten open zijn voor uitbreiding, maar gesloten voor verandering.

Ons doel is dat klassen eenvoudig uitgebreid kunnen worden om nieuw gedrag te incorporeren zonder de bestaande code te wijzigen. Hoe we dat gaan bereiken?

3.4. CODE (Interface manier)

```
public interface Versiering // Component
{
    public String versier();
}

public class Kerstboom implements Versiering // Concrete Component
{
    public String versier()
    {
        return "Kerstboom";
    }
}

public class KerstBal implements Versiering // Decorators
{
    private Versiering versiering;
    private int aantal;

    public KerstBal(int aantal, Versiering versiering)
    {
        this.aantal = aantal;
        this.versiering = versiering;
    }

    public String versier()
    {
        return versiering.versier() + ', met ' + aantal + ' kerstballen';
    }
}

public class KerstSlinger implements Versiering // Decorators
{
    private Versiering versiering;

    public KerstSlinger(Versiering versiering)
    {
        this.versiering = versiering;
    }

    public String versier()
    {
        return versiering.versier() + ', met een slinger';
    }
}

/**
 * Objecten
 */
Versiering kerstBoom = new KerstBoom();
```

```
Versiering kerstBoomMetBallen = new KerstBal(10, new KerstBoom());
Versiering kerstBoomMetBallenEnEenSlinger = new KerstBal(10, new
KerstSlinger(new KerstBoom()));
Versiering kerstBoomMetSlinger = new KerstSlinger(new KerstBoom());

System.out.println(kerstBoom.versier());
System.out.println(kerstBoomMetBallen.versier());
System.out.println(kerstBoomMetBallenEnEenSlinger.versier());
System.out.println(kerstBoomMetSlinger.versier());
```

3.5. CODE (Abstracte manier)

```
public interface Versiering // Component
{
    public String versier();
}

public class Kerstboom implements Versiering // Concrete Component
{
    public String versier()
    {
        return "Kerstboom";
    }
}

public abstract class Decoratie implements Versiering  // Decorator
{
    private Versiering versiering;

    public Decoratie(Versiering versiering)
    {
        this.versiering = versiering;
    }

    @Override
    public String versier()
    {
        return versiering.versier();
    }
}

public class KerstBal extends Decoratie // Concrete Decorator
{

    private int aantal;

    public KerstBal(int aantal, Versiering versiering)
    {
        super(versiering);
        this.aantal = aantal;
```

```
    }

    public String versier()
    {
        return super.versier() + ', met ' + aantal + ' kerstballen';
    }
}

public class KerstSlinger extends Decoratie // Concrete Decorator
{
    public KerstSlinger(Versiering versiering)
    {
        super(versiering);
    }

    public String versier()
    {
        return super.versier() + ', met een slinger';
    }
}

/**
 * Objecten
 */
Versiering kerstBoom = new KerstBoom();
Versiering kerstBoomMetBallen = new KerstBal(10, new KerstBoom());
Versiering kerstBoomMetBallenEnEenSlinger = new KerstBal(10, new
KerstSlinger(new KerstBoom()));
Versiering kerstBoomMetSlinger = new KerstSlinger(new KerstBoom());

System.out.println(kerstBoom.versier());
System.out.println(kerstBoomMetBallen.versier());
System.out.println(kerstBoomMetBallenEnEenSlinger.versier());
System.out.println(kerstBoomMetSlinger.versier());
```

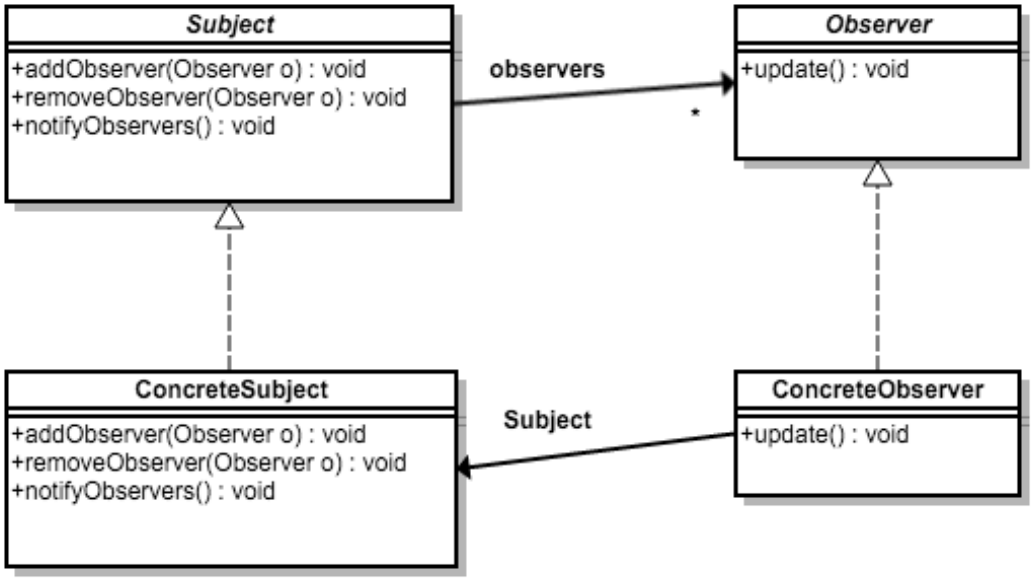
4. Observer Pattern

Gedrag van objecten.

4.1. DEFINITIE

> Het **Observer Pattern** definieert een één-op-veel-relatie (1:n) tussen objecten, zodanig dat wanneer de toestand van een object verandert, alle afhankelijke objecten worden bericht en automatisch worden geüpdatet.

4.2. UML DIAGRAM



4.3. De kracht van zwakke koppeling

> Streef naar ontwerpen met een zwakke koppeling tussen de objecten die samenwerken.

Zwak gekoppelde ontwerpen stellen ons in staat flexibele OO-systemen te bouwen die met verandering om kunnen gaan, omdat ze de wederzijdse afhankelijkheid tussen objecten erg klein maken.

4.4. CODE

Voorbeeld: Weather Data

```
public interface Subject
{
    public void addObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObserver();
}

public interface Observer
{
    public void update(double temp, dobule humidity, double pressure);
}

public interface DisplayElement
{
    public void display();
}

public class WeatherData implements Subject
{
    private double temperature, humidity, pressure;
    private Set<Observer> observers;

    public WeatherData()
    {
        observers = new HashSet<>();
    }

    public void setMeasurements(double temperature, double humidity, double
pressure)
    {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;

        // Dit is de belangrijkste magische stap,
        // Als je dit niet doet wordt de observer niet aangeroepen
        this.notifyObservers();
    }

    public double getTemperature()
    {
        return temperature;
    }

    public double getHumidity()
    {
        return humidity;
    }

    public double getPressure()
    {
        return pressure;
    }
```

```
    @Override
    public void addObserver(Observer o)
    {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o)
    {
        observers.remove(o);
    }

    @Override
    public void notifyObserver()
    {
        /**
         * In dit geval maken we gebruik van het "PUSH"-model
         * Dit wilt zeggen dat we de gegevens al meegeven met de update
methode.
         * Je kan ook de "PULL"-methode gebruiken, dit wilt zeggen dat je de
interfaces kan her-gebruiken
         * omdat je geen gegevens meegeeft, maar in de "update" methode van
de observer, een eventuele getter aanroept.
         */
        // Java 8
        observers.forEach(observer -> observer.update(temperature, humidity,
pressure));

        // Java 7
        for(Observer o : observers)
        {
            o.update(temperature, humidity, pressure);
        }
    }
}

public class CurrentConditionsDisplay implements Observer, DisplayElement
{
    private Subject weatherData;
    private double temperature, humidity;

    public CurrentConditionsDisplay(Subject weatherData)
    {
        this.weatherData = weatherData;

        // Deze stap is zeer belangrijk, want je moet je registreren als een
observer
        weatherData.addObserver(this);
    }

    public void update(double temp, dobule humidity, double pressure)
    {
        this.temperature = temp;
```

```
        this.humidity = humidity;
        display();
    }

    public void display()
    {
        System.out.printf("Actuele weergesteldheid %.1f graden en %.1f %%
luchtvochtigheid%n", temperature, humidity);
    }
}

/**
 * Objecten
 */
WeatherData weatherData = new WeatherData();
CurrentConditionsDisplay currentDisplay = new
CurrentConditionsDisplay(weatherData);

weatherData.setMeasurements(80, 65, 30.4f);
weatherData.setMeasurements(82, 70, 29.2f);
weatherData.setMeasurements(78, 90, 29.2f);
```

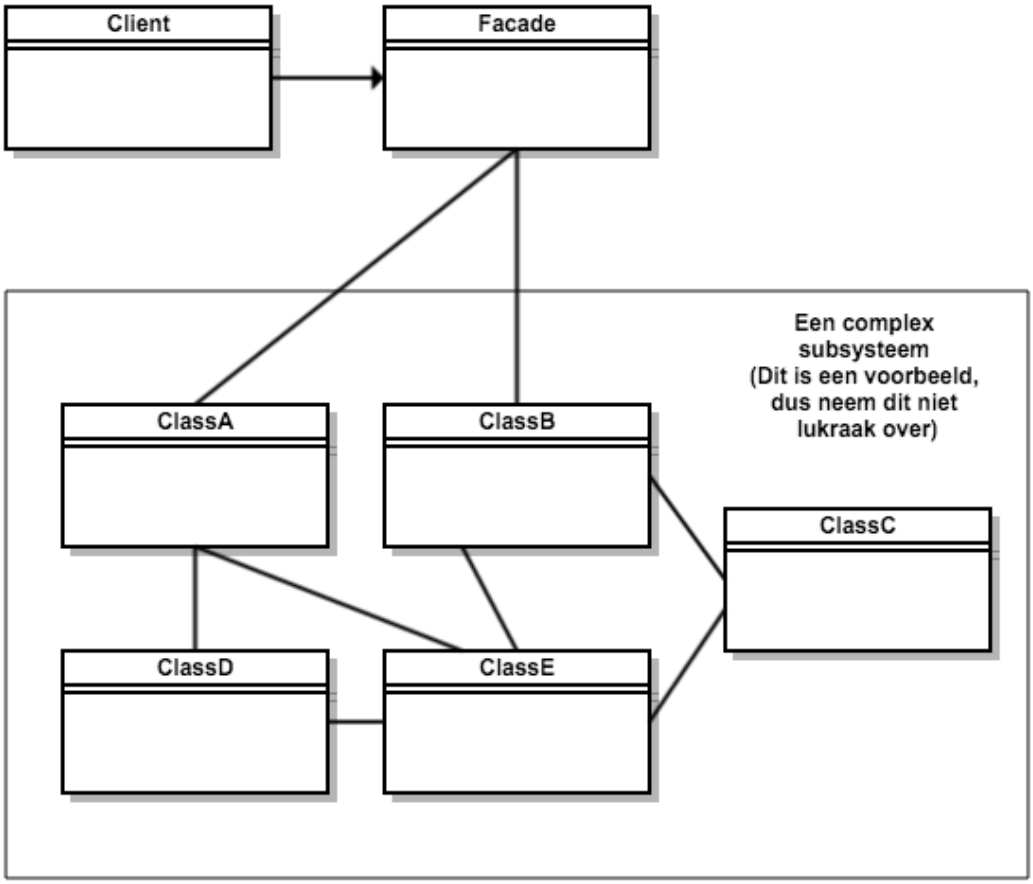
5. Façade Pattern

Structuur van objecten

5.1. DEFINITIE

> Het **Façade Pattern** zorgt voor een vereenvoudigde interface naar een verzameling interfaces in een subsysteem. De façade definieert een interface op een hoger niveau zodat het gebruikt van het subsysteem vereenvoudigt.

5.2. UML DIAGRAM



5.3. Het principe van Kennisabstractie

> Praat alleen met je directe vrienden. Hoe minder je weet hoe beter.

Dit principe voorkomt dat we ontwerpen maken waarbij veel klassen gekoppeld worden, zodat veranderingen in het ene deel van het systeem invloed hebben op andere delen. Wanneer je veel afhankelijkheden aanbrengt tussen klassen, dan maak je een teer systeem dat duur in onderhoud is en bovendien voor anderen moeilijk te begrijpen.

5.4. CODE

Voorbeeld: Home Theater

```
public class HomeTheaterFacade
{
    private Amplifier amp; private Tuner tuner;
    private DvdPlayer dvd; private CdPlayer cd;
    private Projector projector; private TheaterLights lights;
    private Screen screen; private PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp, Tuner tuner, DvdPlayer dvd,
CdPlayer cd, Projector projector, Screen screen, TheaterLights lights,
PopcornPopper popper)
    {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    public void watchMovie(String movie)
    {
        System.out.println("Get ready to watch a movie...");

        popper.on();
        popper.pop();

        lights.dim(10);
        screen.down();

        projector.on();
        projector.wideScreenMode();

        amp.on();
        amp.setDvd(dvd);
        amp.setSurroundSound();
        amp.setVolume(5);

        dvd.on();
        dvd.play(movie);
    }
```

```
    public void endMovie()
    {
        System.out.println("Shutting movie theater down...");

        popper.off();
        lights.on();
        screen.up();
        projector.off();
        amp.off();

        dvd.stop();
        dvd.eject();
        dvd.off();
    }
}
```

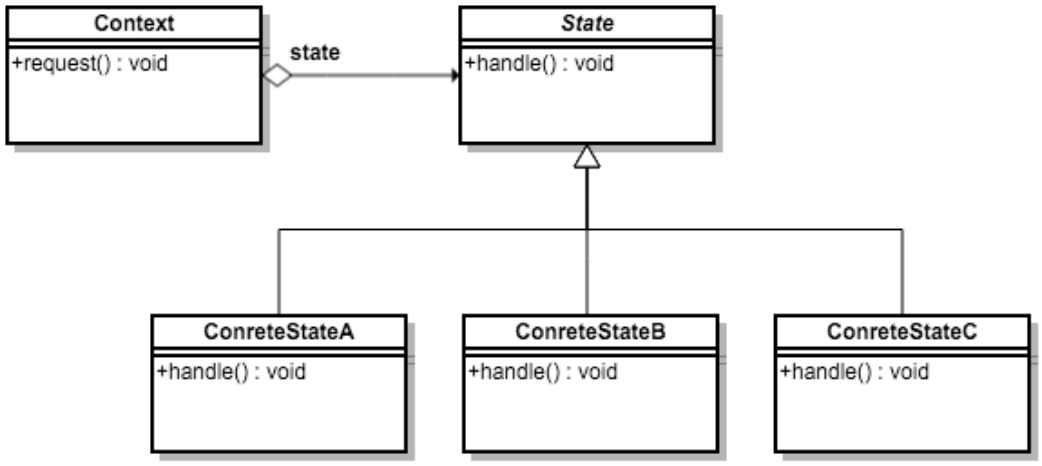
6. State Pattern

Gedrag van objecten

6.1. DEFINITIE

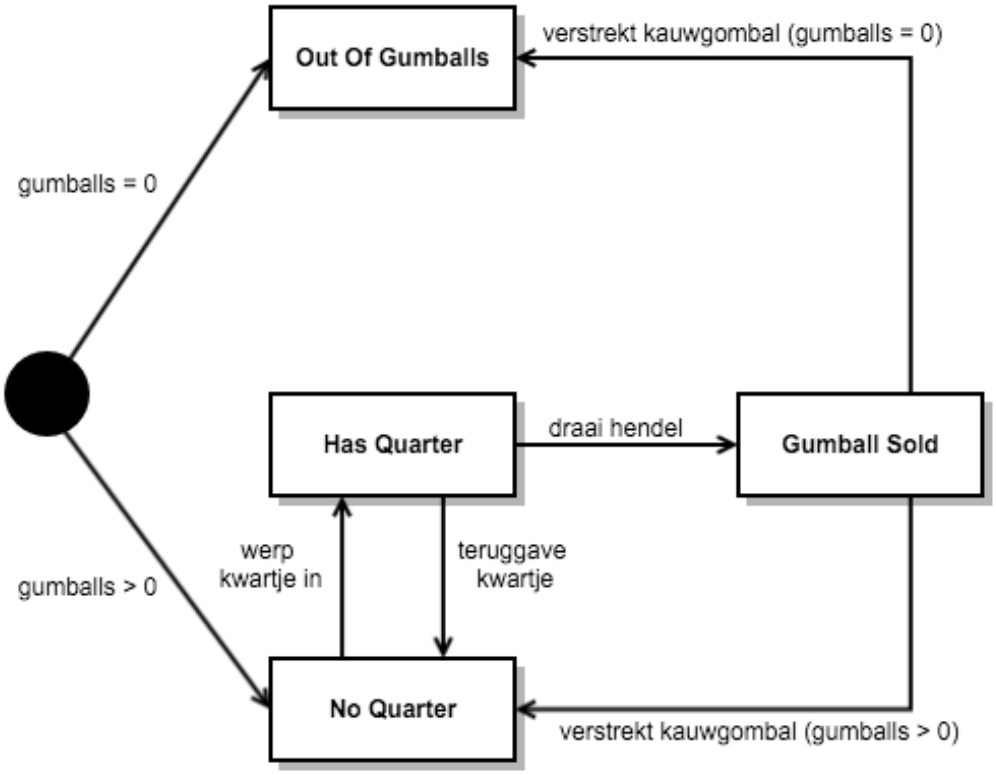
> Het **State Pattern** maakt het voor een object mogelijk zijn gedrag te veranderen wanneer zijn interne toestand verandert. Het object lijkt van klasse te veranderen.

6.2. UML DIAGRAM



6.3. STATE MACHINE DIAGRAM

Voorbeeld van een state machine diagram:



6.4. CODE

Voorbeeld: kauwgomballen

```
abstract class GumballMachineState
{
    protected GumballMachine gumballMachine;

    protected GumballMachineState(GumballMachine gumballMachine)
    {
        this.gumballMachine = gumballMachine;
    }

    protected String insertQuarter()
    {
        return "You can't insert a quarter";
    }

    protected String ejectQuarter()
    {
        return "You haven't inserted a quarter";
    }

    protected String turnCrank()
    {
        return "You can't turn";
    }
```

```
    protected String dispense()
    {
        return "You need to pay first";
    }
}

public class GumballMachine
{

    private GumballMachineState currentState;
    private int count = 0;

    public GumballMachine(int numberGumballs)
    {
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            toState(new NoQuarterState(this));
        } else {
            toState(new OutOfGumballsState(this));
        }
    }

    public String insertQuarter()
    {
        return currentState.insertQuarter();
    }

    public String ejectQuarter()
    {
        return currentState.ejectQuarter();
    }

    public String turnCrank()
    {
        String msg1 = currentState.turnCrank();
        String msg2 = currentState.dispense();
        return String.format("%s\n%s", msg1, msg2);
    }

    protected String releaseBall()
    {
        if (count != 0) {
            count = count - 1;
        }
        return "A gumball comes rolling out the slot...";
    }

    public int getCount()
    {
        return count;
    }

    public void refill(int count)
```

```
    {
        this.count = count;
        toState(new NoQuarterState(this));
    }

    protected void toState(GumballMachineState state)
    {
        currentState = state;
    }

    @Override
    public String toString()
    {
        StringBuilder result = new StringBuilder();
        result.append("\nMighty Gumball, Inc.");
        result.append("\nJava-enabled Standing Gumball Model");
        result.append("\nInventory: " + count + " gumball");
        if (count != 1) {
            result.append("s");
        }
        result.append("\n");
        result.append("Machine is " + currentState + "\n");
        return result.toString();
    }
}

class HasQuarterState extends GumballMachineState
{
    private Random randomWinner = new java.util.Random();

    protected HasQuarterState(GumballMachine gumballMachine)
    {
        super(gumballMachine);
    }

    @Override
    protected String ejectQuarter()
    {
        gumballMachine.toState(new NoQuarterState(gumballMachine));
        return "Quarter returned";
    }

    @Override
    protected String turnCrank()
    {
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.toState(new WinnerState(gumballMachine));
        } else {
            gumballMachine.toState(new SoldState(gumballMachine));
        }
        return "You turned...";
    }
```

```
    @Override
    public String toString()
    {
        return "waiting for turn of crank";
    }
}

class NoQuarterState extends GumballMachineState
{
    protected NoQuarterState(GumballMachine gumballMachine)
    {
        super(gumballMachine);
    }

    @Override
    protected String insertQuarter()
    {
        gumballMachine.toState(new HasQuarterState(gumballMachine));
        return "You inserted a quarter";
    }

    @Override
    public String toString()
    {
        return "waiting for quarter";
    }
}

class OutOfGumballsState extends GumballMachineState
{
    protected OutOfGumballsState(GumballMachine gumballMachine)
    {
        super(gumballMachine);
    }

    @Override
    public String toString()
    {
        return "sold out";
    }
}

class SoldState extends GumballMachineState
{
    protected SoldState(GumballMachine gumballMachine)
    {
        super(gumballMachine);
    }

    @Override
    protected String dispense()
    {
        String msg = gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
```

```
            gumballMachine.toState(new NoQuarterState(gumballMachine));
        } else {
            msg = String.format("%s\n%s", msg, "Oops, out of gumballs!");
            gumballMachine.toState(new OutOfGumballsState(gumballMachine));
        }
        return msg;
    }

    @Override
    public String toString()
    {
        return "dispensing a gumball";
    }
}

class WinnerState extends GumballMachineState
{
    protected WinnerState(GumballMachine gumballMachine)
    {
        super(gumballMachine);
    }

    @Override
    protected String dispense()
    {
        String msg = "YOU'RE A WINNER! You get two gumballs for your
quarter\n";
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            msg += "\nOops, out of gumballs!";
            gumballMachine.toState(new OutOfGumballsState(gumballMachine));
        } else {
            msg += gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.toState(new NoQuarterState(gumballMachine));
            } else {
                gumballMachine.toState(new
OutOfGumballsState(gumballMachine));
            }
        }
        return msg;
    }

    @Override
    public String toString()
    {
        return "despensing two gumballs for your quarter, because YOU'RE A
WINNER!";
    }
}

/**
 * Objects
 */
```

```
GumballMachine gumballMachine = new GumballMachine(5);

System.out.println(gumballMachine);

System.out.println(gumballMachine.insertQuarter());
System.out.println(gumballMachine.turnCrank());

System.out.println(gumballMachine);

System.out.println(gumballMachine.insertQuarter());
System.out.println(gumballMachine.turnCrank());
System.out.println(gumballMachine.insertQuarter());
System.out.println(gumballMachine.turnCrank());

System.out.println(gumballMachine);
```

7. MVC Pattern

MVC is een combinatie van een paar patterns.

- **Strategy Pattern**
  Dit wordt gebruikt om de "input" om te zetten naar acties
- **Observer Pattern**
  Dit wordt gebruikt om de view te updaten als er iets gewijzigd wordt.
- **Composite Pattern**
  Dit wordt gebruikt om de GUI-componenten op te bouwen.