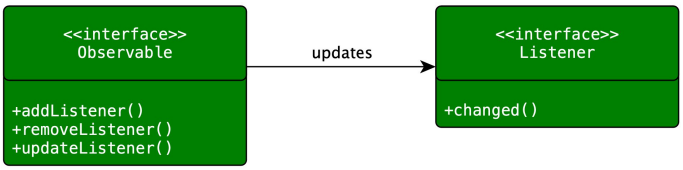
Observer Pattern (Beobachter)



Das Observer Pattern ermöglicht es, ein oder mehrere Objekte über eine Zustandsänderung eines anderen Objektes zu informieren und darauf zu reagieren.

Implementation

Für die Implementierung muss ein Observerable/Subject eine Funktion haben, welche dann bei allen Listenern/Observern eine Funktion ausführt. Diese Funktion muss beim registrieren definiert werden.

```
@FunctionalInterface
interface Listener{
 void changed(Object e);
}

class Observable {
  private Object value;
  private List<Listener> listeners;
  public void setValue(Object value) {
    if(!this.value.equals(value)){
      this.value = value; updateListeners();
    }
  }
  public void updateListeners(){
    for(Listener l : listeners) l.changed(value);
  }
  public void addListener(Listener listener) {
     listeners.add(listener);
  }
}
```

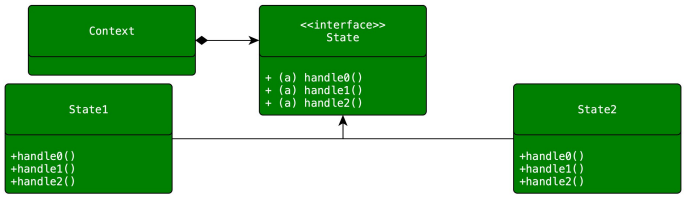
```
→ observable.addListener(e -> System.out.println(e));
```

Beim Ausführen von **observable.set(someVal)**, wird in der Konsole dann der Wert von **someVal** ausgegeben.

Probleme

1	Da das Ändern eines Observables oft einen Seiteneffekt bei einem Listener auslöst, können zyklische Abhängigkeiten entstehen. Das, wenn ein Observable geändert wird und dann ein Listener ein Observable ändert auf welches das erste Observable ein Listener hat und durch das wieder eine Änderung verursacht.
	Dem Problem kann entgegengewirkt werden indem Updates nur gesendet werden wenn sie den Wert effektiv geändert haben.

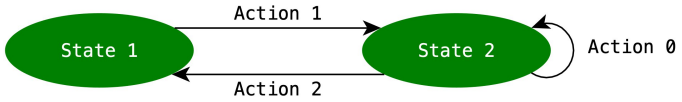
State Pattern (Zustand)



Das State Pattern wird verwendet um Anhand des Zustands eines Objekts sein eigenes Verhalten zu ändern. State Patterns lassen sich durch Statusdiagramme ausdrücken.

Implementierung

Es gibt zwei Arten State Patterns zu implementieren. Die eine ist der triviale Ansatz mit einem Switch-Case. Der OOP-Like Ansatz verwendet dazu Objekte. Dabei implementiert eine Abstrakte Klasse **AbstractState** alle Aktionen der Zustände welche alle zuerst einen Fehler werfen. Die Zustände implementieren die Aktionen welche sie effektiv ausführen. Damit das ganze etwas schöner aussieht, wird ein Interface **State** generiert.

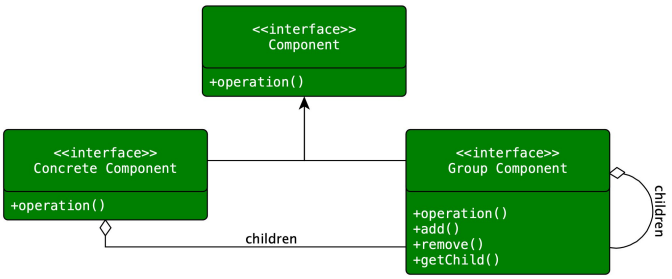



```
class StatePattern {
  private interface State {
    public void action0();
    public void action1();
    public void action2();
  }
  private abstract class AbstractState {
    public void action0() {throw new IllegalStateException();}
    public void action1() {throw new IllegalStateException();}
    public void action2() {throw new IllegalStateException();}
  }

  class State1 extends AbstractState{
    public void action1() { /* do smth */ state = state1; }
  }
  class State2 extends AbstractState{
    public void action0() { /* do smth */ state = state2; }
    public void action2() { /* do smth */ state = state1; }
  }

  private final State STATE1 = new state1();
  private final State STATE2 = new state2();

  public State state = STATE1;
}
```

```
→ state.action1(); state.action0(); state.action2() // valid
→ state.action0(); // throws IllegalStateException
```

Mithilfe des State Patterns ist es nun möglich, unmögliche Statusänderungen zu detektieren und darauf zu reagieren.

Composite Pattern (Zusammengesetzes)



Das Composite Pattern hilft, Einzelteile einer Implementierung zusammenzufassen. Das Ziel ist es, dass mehrere Objekte wie eines verwendet werden können, und Aufrufe auf das zusammengesetzte Objekt (wenn nötig rekursiv) an die Kinder übergeben werden. Die Gruppenobjekte erlauben es auch, andere Gruppen zu beinhalten, sowie simplere Objekte, welche auch "Leaves" genannt werden.

Implementierung

Für die Implementierung wird ein Interface benötigt, welches von der Gruppierungsklasse und von den eigentlichen Implementierungen erweitert wird.

```
interface File {
  public int size();
}
class Textfile extends File {
  private String content;
  public int size() { /* size calculation */ }
}
class Folder extends File {
  private List<File> files;
  public int size() {
    int size = 0;
    for(File f : file) {
      size += f.size();
    }
    return size();
  }
}
```

```
→ Folder f = new Folder(new File(), new Folder(new File()));
→ f.size() // returns the size of the two files
```

Funktionen welche vom Interface implementiert werden können ganz einfach aufgerufen werden und bei richtiger Implementierung ein Resultat für alle Teile erhalten.

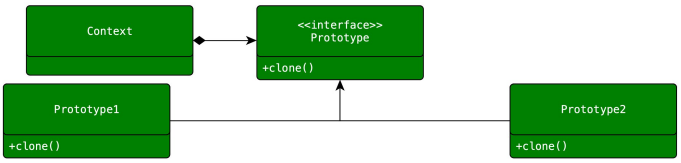
Probleme

1	Da Gruppen immer nur aus dem Interface bestehen, können nicht auf Funktionen von Unterklassen zugegriffen werden.
	Gruppenmitglieder können mit **instanceOf** sowie sowie einem Cast gecastet werden.
2	Wenn eine Gruppe **G1** ein Mitglied einer anderen Gruppe **G2** ist, und nun **G1** der Gruppe **G2** hinzugefügt wird, entsteht eine zyklische Abhängigkeit. Diese kann auch entstehen, wenn Gruppen sich indirekt einander hinzufügen.
	Bevor eine Gruppe **G1** hinzugefügt werden kann, muss für alle Gruppen in der hinzuzufügenden Gruppe **G2** geprüft werden, ob sie schon irgendwie der Gruppe **G1** hinzugefügt wurde. Falls ja, darf die Gruppe **G2** nicht hinzugefügt werden.

Immutables

Immutables sind nicht mutierbare Objekte. Das bedeutet dass für jede Änderung ein neues Objekt erstellt werden muss. Problemen wie Thread Safety oder inkonsistenten Zuständen wird damit jedoch entgegengewirkt. Zusätzlich müssen Immutables nicht kopiert werden, eine Referenz auf das Objekt reicht – da es ja nicht verändert werden kann. Beim Erstellen eines neuen Immutables können Abhängige Zustände auf ein Basisobjekt referenziert werden – wird dies fortgeführt entsteht ein grosser Baum an Immutables, welcher aufgrund der nicht-mutierbarkeit Sicher ist.

Prototype Pattern (Prototyp)



Das Prototype Pattern wird verwendet um Objekte über einen Prototypen zu generieren. Dies hat zwei Vorteile:

1. Komplexe Instanzierungen können umgangen werden
2. Die Werte des Prototypen können übernommen werden

Implementierung

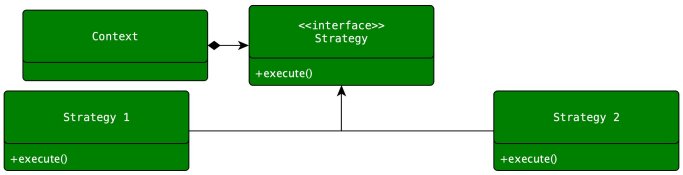
Für die Implementierung (in Java) müssen alle Objekte die clone Methode implementieren. Man kann etweder über Java-Cloning (implements Cloneable) oder via Copy Konstruktoren arbeiten. Wichtig, der Prototyp muss in den meisten Fällen eine Deep-Copy machen, wobei Java-Cloning nur Shallow-Copies erstellt.

```
class Point implements Cloneable {
  private int x;
  private int y;
  public Point clone(){
    try{
       return (Point)super.clone();
    } catch (CloneNotSupportedException e) {
      System.err.println(e);
    }
  }
}
class Line {
  private Point p1, p2;
  protected Line(Line l) {
    l.p1 = l.p1.clone(); l.p2 = l.p2.clone();
  }

  public Line clone(){ return new Line(this);}
}
```

```
→ Line l1 = new Line(new Point(0, 0), new Point(10, 10));
→ Line l2 = l1.clone();
```

Line **l2** hat nun denselben Inhalt wie **l1**, jedoch können beide Instanzen individuell verändert werden, ohne dass sie sich gegenseitig verändern.

Strategy Pattern (Strategie)



Das Strategy Pattern erlaubt es, eine Familie von Algorithmen zu definieren. So kann ein Algorithmus zur Laufzeit bestimmt und eingesetzt werden. Ein weiterer Vorteil des Strategy Patterns ist es, dass neue Algorithmen problemlos implementiert werden können, ohne in anderen Klassen etwas zu ändern.

Implementierung

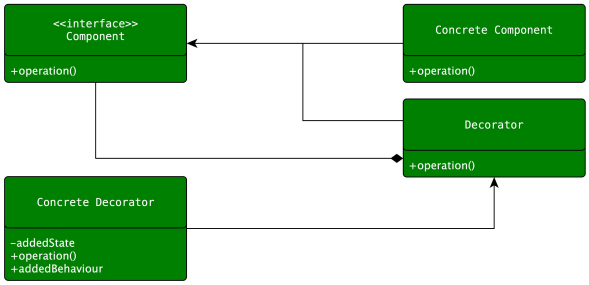
Für die Implementierung des Strategy Patterns wird eine Klasse benötigt, welche mehrere Algorithmen benötigt.

```
class SecureChannel {
  private interface Algorithm {
    public Data encrypt(Data d);
    public Data decrypt(Data d);
  }
  public AES implements Algorithm {
    public Data encrypt(Data d) { /* do smth */ }
    public Data decrypt(Data d) { /* do smth */ }
  }
  public IDEA implements Algorithm {
    public Data encrypt(Data d) { /* do smth */ }
    public Data decrypt(Data d) { /* do smth */ }
  }
  private Algorithm alg;
  public void sendData(Data d) { write(alg.encrypt(d); }
  public Data receive() { return alg.decrypt(read()); }
}
```

```
→ SecureChannel sc1 = new SecureChannel(new AES());
→ SecureChannel sc2 = new SecureChannel(new IDEA());
```

Nun hat man zwei sichere Kanäle, wobei der erste mit AES, der zweite mit IDEA verschlüsselt.

Decorator (Dekorierer)



Das Decorator Pattern hilft, Eigenschaften von Komponenten zu ergänzen. Somit müssen nicht die verschiedenen Dekoratoren für jede Komponente implementiert werden. Dabei bleibt das Interface dasselbe, weshalb die Komponenten wie das originale Objekt verwendet werden kann.

Implementierung

```
interface Component {
  public void operation();
}
abstract Decorator implements Component { }
class MathComponent implements Component {
  public void operation() { /* do math */ }
}
class ConsoleDecorator extends Decorator {
  private Component inner;
  public ConsoleDecorator(Component inner) {
    this.inner = inner;
  }
  public void operation() {
    System.out.println("Executing a " + inner.getClass());
    inner.operation();
  }
}
→ Component comp = new ConsoleDecorator(new MathComponent);
```

Man erhält einen MathComponent, welcher mit dem Konsolendekorator dekoriert wurde. Das bedeutet, man kann **comp.operation()** ausführen, und erhält zum einen eine Meldung auf der Konsole, und zum anderen wird die Matheoperation ausgeführt.

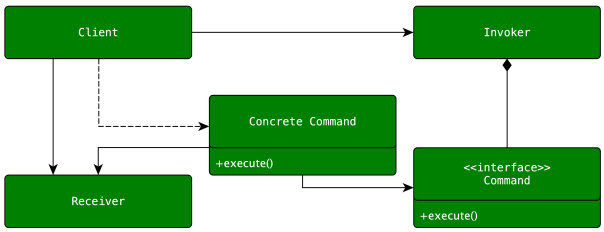
Probleme

1	Identität der ursprünglich dekorierten Klasse geht verloren. Das bedeutet also, dass **comp** vom Typ **ConsoleDecorator** ist, jedoch nicht dass es sich eigentlich um einen **MathComponent** handelt.
	– Methode, welche den Typ der Basisklasse zurück gibt. – Klassen, welche auf den unteren Typ zugreifen selbst dekorieren, dass sie auch über den oberen Typ funktionieren
2	Manche Dekoratoren schliessen sich gegenseitig aus. Als Beispiel wäre ein Fixierungs- und ein Animationsdekorator auf einer Figur. Je nach Implementierung wird dann nur der zuerst oder der später installierte Dekorator ausgeführt, oder es wird ein Dekorator immer priorisiert.
	– **Sich ausschliessende Dekoratoren können nicht gleichzeitig installiert sein.** (Fordert jedoch, dass die gegenseitig inkompatiblen Dekoratoren irgendwo festgehalten werden) – **Delegation der Methoden an den obersten Dekorator.** Im Beispiel des Animationsdekorators würde der Animationsdekorator einen **move** Befehl ausführen, jedoch das **move** der inneren Figur aufrufen. Mit der Delegation würde das **move** bis zum obersten wandern, und von dort gegen unten ausgeführt. Der Fixierungsdekorator gibt jedoch das **move** nicht mehr weiter nach unten und wird dadurch priorisiert.*

* Um so implementiert werden zu können, müssen die Komponenten sich jeweils ihren Parent merken. Wenn sie einen Parent haben, dann geben sie den Aufruf weiter an den Parent. Wenn nicht, dann führen sie Methode effektiv aus.

```
// Figure "interface"
abstract class Figure {
  public abstract void moveImpl(int dx, int dy);
  public final void move(int dx, int dy) {
    if (parent != null) {
      parent.move(dx, dy);
    } else {
      moveImpl(dx, dy);
    }
  }

  private Figure parent;
  public final Figure getParent() { return parent; }
  public void setParent(Figure parent) {
    this.parent = parent;
  }

}
// Parent is set when decorator is created, move implementation
// defines that when parent is present, run their impl.
```

Command (Kommando)



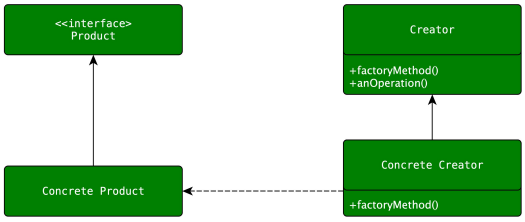
Das Command Pattern formt Aktionen in Objekte um. Das bedeutet, dass jede Aktion zwischen zwei Objekten als Objekt abgespeichert werden kann – quasi Kapseln von Requests in Objekte.

- Der **Client** generiert **Concrete Commands** und setzt die Informationen welche benötigt werden um später ausgeführt zu werden.
- Der **Invoker** entscheidet wann ein Kommando ausgeführt wird
- Der **Receiver** weiss wie die Kommandos ausgeführt werden müssen und führt diese auch effektiv aus, wenn der **Invoker** das Kommando ausgeführt haben möchte.

```
interface Command {
  public void execute();
}
class HelloCommand implements Command {
  public void execute() { System.out.println("Hello world!"); }
}
class Receiver {
  public void runCommand(Command command){
    command.execute();
  }
}
class Invoker {
  List<Command> commandLog = new ArrayList<>();
  Receiver receiver;

  public Invoker(Receiver receiver){
     this.receiver = receiver;
  }
  public void addCommand(Command command){
    commandLog.add(command);
    receiver.runCommand(command);
  }
}
→ Invoker invoker = new Invoker(new Receiver));
→ invoker.addCommand(new HelloCommand());// Outputs Hello World
```

Client.run führt das Command Pattern Beispiel aus. In diesem Beispiel ist der Invoker nur da um die Kommandos zu loggen. Ein erweiterter **Invoker** würde zum Beispiel eine Undo/Redo Logik implementieren. Meistens ist **Client** und **Receiver** dieselbe Klasse und es wird nicht so extrem getrennt.

Factory Method (Fabrikmethode)



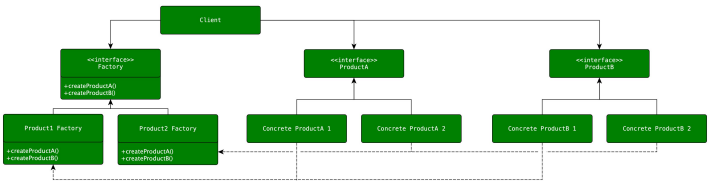
Das Factory Method Pattern beschreibt das Grundkonzept der Initialisierung & Instanzierung über Methoden anstatt über **new**. Das bedeutet, dass neue Objekte via einen Methodenaufruf (vorparametrisiert) generiert werden. Somit muss man sich keine Gedanken machen, welche Konstruktorargumente mitgegeben werden müssen.

Implementierung

```
interface GuiElement {}
abstract class GuiCreator{
  public abstract GuiElement create();
}
class Button extends GuiCreator implements GuiElement {
  public static GuiElement create() { return new Button(); }
}
class TextField extends GuiCreator implements GuiElement {
  public static GuiElement create() { return new TextField(); }
}
→ GuiElement button = Button.create();
```

Die Elemente können ganz einfach mit den Factory Methoden generiert werden. Dies ist vor allem nützlich, wenn bei jeder Neuerzeugung noch andere Schritte getan werden müssen.

Abstract Factory (Abstrakte Fabrik)



Das Abstract Factory Pattern ist eine Mischung zwischen dem Strategy und dem Factory Method Pattern. Es ermöglicht zur Laufzeit Factories mit demselben Interface zu wechseln. Die Strategie an dem ganzen die jewelige Implementation der Factory, wobei die Strategie welche gewechselt wird eine Factory ist.

Implementierung

```
interface Factory {
  public ProductA createProductA();
  public ProductB createProductB();
}
interface ProductA {}
interface ProductB {}

class ConcreteProductA1 implements ProductA {}
class ConcreteProductA2 implements ProductA {}
class ConcreteProductB1 implements ProductB {}
class ConcreteProductB2 implements ProductB {}


class Product1Factory implements Factory {
  public ProductA createProductA() {
    return new ConcreteProductA1();
  }
  public ProductB createProductB() {
    return new ConcreteProductB1();
  }
}
class Product2Factory implements Factory {
  public ProductA createProductA() {
    return new ConcreteProductA2();
  }
  public ProductB createProductB() {
    return new ConcreteProductB2();
  }
}
→ Factory factory1 = new Product1Factory();
→ ProductA productA = factory1.createProductA();
```

Zur Laufzeit muss definiert werden, welche **Factory** verwendet wird. Der einzige Punkt welcher unterscheidet ob es sich um Produkt 1 oder Produkt 2 handelt ist bei der Zuweisung der **Factory** mit **new**.

Dependency Injection (Abhängikeitsinjektion)

Die Dependency Injection ist kein Design Pattern. Ein Framework dafür wird jedoch oft mithilfe von des Factory Patterns aufgebaut. Dependency Injection in Spring zum Beispiel vereinfacht das Aufsetzen von Objekten zum Start der Applikation, indem die Objekte von Spring selbst instanziiert werden - die Konfiguration passiert in einer separaten .xml Datei.

Implementierung

```
// Java Code
class Bean {
  private String name;
  private List<String> list;
  private Another a;

  public Bean(String name, List<String> list, Another a) {
    this.name = name;
    this.list = list;
    this.a = a;
  }
}
class Another {
  private String name;
  public Another(String name) { this.name = name; }
}



// Spring Application context file "context.xml"
<bean id="bean" class="Bean" scope="prototype">
  <constructor-arg><value>A</value></constructor-arg>
  <constructor-arg>
    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </list>
  <constructor-arg><ref bean="another"/></constructor-arg>
</bean>

<bean id="another" class="AnotherBean">
  <property name="name"><value>another</value></property>
</bean>

→ ApplicationContext context
  = new ClassPathXmlApplicationContext("context.xml");
→ Bean bean = context.getBean("bean" Bean.class)
```

Mit Dependency Injection generierte Klassen können geladen werden und dann wie normale instanziierte Objekte verwendet werden.

Mit Spring können noch weitere Container erzeugt werden als nur Listen.

java.util.**List**	java.util.**Set**

`<property name="list">`
`<list>`
`<value>string</value>`
`<ref bean="foo"/>`
`<ref bean="bar"/>`
`</list>`
`</property>`
	`<property name="set">`
	`<set>`
	`<value>string</value>`
	`<ref bean="foo"/>`
	`<ref bean="bar"/>`
	`</set>`
	`</property>`

java.util.**Map**	java.util.**Props**

`<property name="map">`
`<map>`
`<entry key="key">`
`<value>`
`<ref bean="foo"/>`
`</value>`
`</entry>`
`</map>`
`</property>`
	`<property name="props">`
	`<props>`
	`<prop key="key1">`
	`bar1`
	`</prop>`
	`<prop key="key2">`
	`bar2`
	`</prop>`
	`</props>`
	`</property>`