Diagram of an applet demonstrating BorderLayout.
      Each section of the BorderLayout contains a Button corresponding to its position in the layout, one of:
      North, West, Center, East, or South.//Borderlayout

import java.awt.\*;

import java.applet.Applet;

public class buttonDir extends Applet {

public void init() {

setLayout(new BorderLayout());

add(new Button("North"), BorderLayout.NORTH);

add(new Button("South"), BorderLayout.SOUTH);

add(new Button("East"), BorderLayout.EAST);

add(new Button("West"), BorderLayout.WEST);

add(new Button("Center"), BorderLayout.CENTER);

}

}

import java.awt.\*;

Shows 6 buttons in rows of 2. Row 1 shows buttons 1 then 2.
 Row 2 shows buttons 3 then 4. Row 3 shows buttons 5 then 6. import java.applet.Applet;

public class ButtonGrid extends Applet {

public void init() {

setLayout(new GridLayout(3,2));

add(new Button("1"));

add(new Button("2"));

add(new Button("3"));

add(new Button("4"));

add(new Button("5"));

add(new Button("6"));

}

}

## ****Exception throw Beispiel:****

**package** Exeptionspackage;

// Test.java

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** KontoAusnahme {

// Ausnahmen vom Typ KontoAusnahme werden weitergereicht

// und führen zum Abbruch des Programms.

Konto kto = **new** Konto(4711, 500);

/\*

\* Beim nächsten methodenaufruf "kto.zahleAus", wird zuerst überprüft ob der Betrag nicht

\* unter 0 ist. Danach wird noch kontrolliert, ob überhaupt genügend

\* Geld auf dem Konto vorhanden ist, um den Betrag auszuzahlen. Da dies nicht der Fall ist,

\* wird eine Excetpion geworfen, welche in der KOntoAusnahme Klasse definiert ist.

\* \*/

kto.zahleAus(1000);

kto.zahleAus(300);

kto.info();

}

}

**package** Exeptionspackage;

// Konto.java

**public** **class** Konto {

**private** **int** kontonummer;

**private** **double** saldo;

**public** Konto() {

}

**public** Konto(**int** kontonummer) {

**this**.kontonummer = kontonummer;

}

**public** Konto(**int** kontonummer, **double** saldo)

**throws** KontoAusnahme {

**if** (saldo < 0)

**throw** **new** KontoAusnahme("Negativer Saldo: " + saldo);

**this**.kontonummer = kontonummer;

**this**.saldo = saldo;

}

**public** **int** getKontonummer() {

**return** kontonummer;

}

**public** **void** setKontonummer(**int** nr) {

kontonummer = nr;

}

**public** **double** getSaldo() {

**return** saldo;

}

**public** **void** setSaldo(**double** betrag) **throws** KontoAusnahme {

**if** (betrag < 0)

**throw** **new** KontoAusnahme("Negativer Saldo: " + betrag);

saldo = betrag;

}

**public** **void** zahleEin(**double** betrag) **throws** KontoAusnahme {

**if** (betrag < 0)

**throw** **new** KontoAusnahme("Negativer Betrag: " + betrag);

saldo += betrag;

}

**public** **void** zahleAus(**double** betrag) **throws** KontoAusnahme {

**if** (betrag < 0)

**throw** **new** KontoAusnahme("Negativer Betrag: " + betrag);

**if** (saldo < betrag)

**throw** **new** KontoAusnahme("Betrag > Saldo");

saldo -= betrag;

}

}

**package** Exeptionspackage;

// KontoAusnahme.java

**public** **class** KontoAusnahme **extends** Exception {

**public** KontoAusnahme() {

}

**public** KontoAusnahme(String message) {

**super**(message);

}

}

Es gibt unterschiedliche Typen von Entwurfsmustern. Ursprünglich wurden folgende Typen unterschieden:

[**Erzeugungsmuster**](https://de.wikipedia.org/wiki/Erzeugungsmuster)**(englisch *creational patterns*)**

Dienen der Erzeugung von Objekten. Sie entkoppeln die Konstruktion eines Objekts von seiner Repräsentation. Die Objekterzeugung wird gekapselt und ausgelagert, um den Kontext der Objekterzeugung unabhängig von der konkreten Implementierung zu halten, gemäß der Regel: „Programmiere auf die Schnittstelle, nicht auf die Implementierung!“

[**Strukturmuster**](https://de.wikipedia.org/wiki/Strukturmuster)**(englisch *structural patterns*)**

Erleichtern den Entwurf von Software durch vorgefertigte Schablonen für Beziehungen zwischen Klassen.

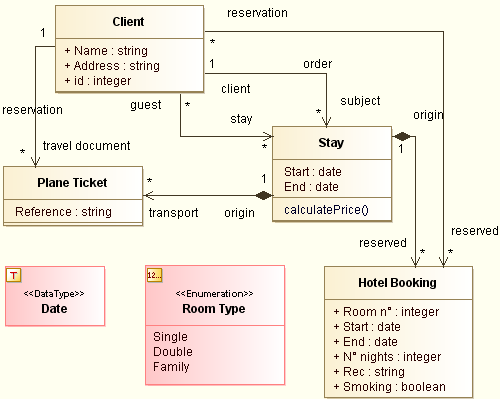
[**Verhaltensmuster**](https://de.wikipedia.org/wiki/Verhaltensmuster_(Software))**(englisch *behavioral patterns*)**

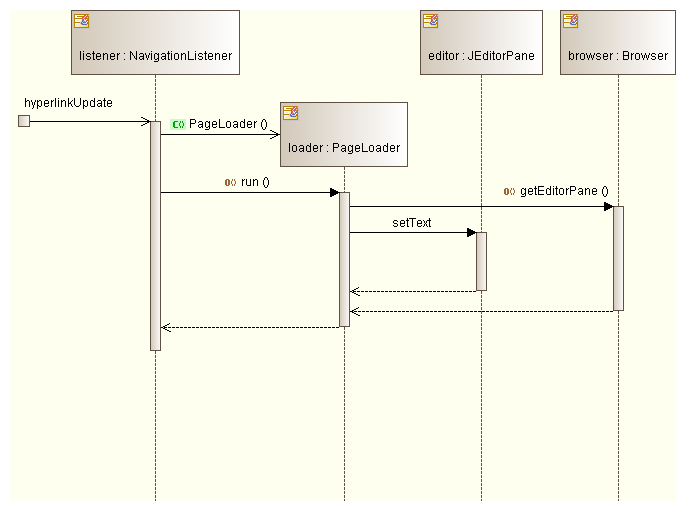
Modellieren komplexes Verhalten der Software und erhöhen damit die Flexibilität der Software hinsichtlich ihres Verhaltens.

Später kamen noch weitere Typen für diejenigen Entwurfsmuster, die in keine der drei genannte Typen passten, hinzu. Darunter z.B.

**Muster für**[**objektrelationale Abbildung**](https://de.wikipedia.org/wiki/Objektrelationale_Abbildung)

Dienen der Ablage und des Zugriffes von Objekten und deren Beziehungen in einer relationalen Datenbank.[[3]](https://de.wikipedia.org/wiki/Entwurfsmuster#cite_note-PoEAA-3)

[](https://www.google.ch/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRw&url=https://www.modeliosoft.com/de/ressourcen/diagrammbeispiele/klassen-und-paket-diagramme.html&ei=Lo6CVZrZL4arsAGso4HYBQ&bvm=bv.96041959,d.bGg&psig=AFQjCNH06wj2IxhLvD3TPcbBSecSXVc59Q&ust=1434705833995129)

[](https://www.google.ch/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRw&url=https://www.modeliosoft.com/en/resources/diagram-examples/sequence-diagrams.html&ei=mI6CVfSLG8OcsAHevpjIDw&bvm=bv.96041959,d.bGg&psig=AFQjCNFbuieHPR78rXhEz5DVemsP7SETCg&ust=1434705915079100)

Das **Factory Method Entwurfsmuster** dient der Entkopplung des Clients von der konkreten Instanziierung einer Klasse. Das erstellte Objekt kann elegant ausgetauscht werden. Oft wird es zur Trennung von (zentraler) Objektverarbeitung und (individueller) Objektherstellung verwendet.

Der Erstellungscode eines Objektes (Product genannt) wird in einer eigenen Klasse (Creator, Factory) ausgelagert. Dieser Creator ist abstrakt und delegiert die konkrete Objektinstanziierung wiederrum an seiner Unterklasse. Erst die Unterklasse entscheidet welches Product erstellt wird. Da der Client sich komplett auf Abstraktion stützt (sowohl beim Creator als auch bei den Products), ist er vollkommen von den Implementierungen entkoppelt und unabhängig.

Der Creator ist abstrakt, und kennt nur die abstrakte Schnittstelle vom Product und instanziiert nicht ein konkretes Productobjekt, sondern lässt seine Unterklassen entscheiden, welches konkrete Product erzeugt werden soll. Dazu definiert es eine abstrakte Methode (die namensgebende factoryMethod()), die es in seiner createProduct()-Methode aufruft und die seine Unterklassen implementieren müssen. Unterklassen (ConcreateCreators) implementieren diese Methode und geben ein ConcreteProduct zurück.

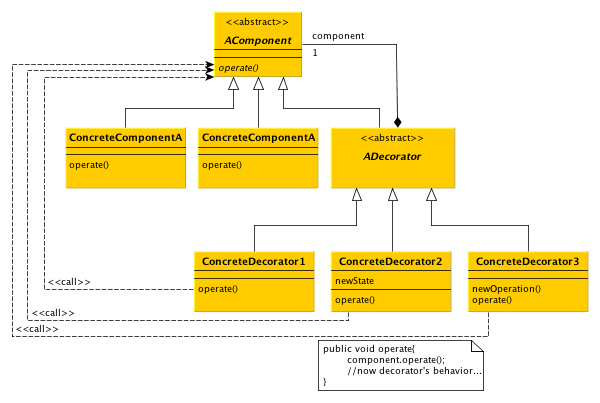
Das **Singleton Entwurfsmuster** sorgt dafür, dass es von einer Klasse nur eine einzige Instanz gibt und diese global zugänglich ist.

Damit es nur eine einzigartige Instanz gibt, muss eine Instanziierung durch den Client verhindert werden. dafür wird der Konstruktur privat deklariert. Nun kann einzig der Singletoncode selbst das Singleton instanziieren.

Weiterhin definiert die Singletonklasse eine global verfügbare Methode, in der diese einzigartige Singletoninstanz zurückgegeben wird. In Java wird dies mit den Modifiern public und static erreicht. Der Singletoncode muss (in der Methode) sicherstellen, dass immer nur ein und dasselbe Objekte an den Client gelangt. Die verschiedenen Varianten, dies zu realisieren, werden im Kapitel Variationen diskutiert.

Das **Observer Pattern** ermöglicht, dass sich Objekte (Observer, beobachtendes Objekt) bei einem anderem Objekt (Subject, beobachtetes Objekt) registrieren und fortan vom diesem informiert werden, sobald es sich ändert.

Für die Observer wird eine einheitliche Schnittstelle (Interface) mit mindestens einer Aktualisierungsmethode definiert. Diese wird vom Subject im Falle von Aktualisierungen aufgerufen und ist in den meisten Fällen mit näheren Daten zur Änderung parametrisiert. Konkrete Observer implementieren das Interface und damit die Aktualisierungsmethode und bestimmen somit, wie der Observer auf die Benachrichtigung reagieren soll.

Das Subject benötigt Administrationsmethoden, damit sich Observer an- und abmelden können. Meldet sich ein Observer an, so nimmt das Subject es in seine Liste der zu benachrichtigen Objekte auf. Treten nun Änderungen am Subjectzustand auf, so werden alle registrierten Observer informiert (notifyObservers()). Dies geschieht, in dem über die Observerliste des Subjects iteriert wird, und auf jedem Observer die Aktualisierungsmethode (update()) aufgerufen wird.

Das **Decorator Design Pattern** ermöglicht das dynamische Hinzuf**ü**gen von Fähigkeiten zu einer Klasse. Dazu wird die Klasse, dessen Verhalten wir erweitern möchten (Component, Komponente), mit anderen Klassen (Decorator, Dekorierer) dekoriert (vgl. engl. "to wrap": umhüllen). Das heißt der Decorator umschließt (enthält) die Component. Der Decorator ist vom selben Typ wie das zu dekorierende Objekt, hat somit die gleiche Schnittstelle und kann an derselben Stelle wie die Component benutzt werden. Er delegiert Methodenaufrufe an seine Component weiter und führt sein eigenes Verhalten davor oder danach aus.

Eine Component kann mit beliebig vielen Decorators dekoriert werden, um so seine Fähigkeiten immer weiter auszubauen.

**Iterator Pattern** - Greift auf alle Elemente einer Sammlung zu.

Häufig besteht die Notwendigkeit auf alle Elemente einer Sammlung (engl. Collection) (z.B. eines Arrays oder einer Liste) zuzugreifen - beispielsweise um diese an eine Funktion zu übergeben. Der Iterator ermöglicht es, dies sequentiell (d.h. in einer bestimmten Reihenfolge) zu tun, ohne dabei wissen zu müssen, wie die Sammlung aufgebaut ist.

Beispielsweise könnten natürliche Zahlen (oder andere Objekte, auf denen eine Ordnung definiert ist) in einer Liste oder einem Binärbaum gespeichert abgelegt sein. Wollen wir die bisher abgelegten Zahlen z.B. mit einer besonderen Formatierung ausgeben, stehen wir vor dem Problem, dass sich die Algorithmen zum Traversieren (d.h. Durchlaufen) eines Baumes und einer Liste stark unterscheiden - wir müssten in unserem Programm also eine Abfrage einbauen, die - je nachdem ob wir es mit einem Baum oder einer Liste zu tun haben - zu unterschiedlichen Unterprogrammen springt. Der Einsatz eines Iterators macht diese umständliche Vorgehensweise überflüssig: Über die eigentliche Sammlung wird einfach ein passender Iterator "gestülpt" und wir können nacheinander auf die einzelnen Zahlen zugreifen, indem wir uns der (einfachen und immer gleichen) Schnittstelle des Iterators bedienen (polymorphe Traversierung).

Weiterhin erlaubt das Iterator-Muster die mehrfache und gleichzeitige Traversierung derselben Sammlung: Dazu reicht es aus, mehrere Iterator-Objekte zu erzeugen

**Das Composite Entwurfsmuster** ermöglicht es, eine verschachtelte (Baum)Struktur einheitlich zu behandeln, unabhängig davon, ob es sich um ein atomares Element oder um ein Behälter für weitere Elemente handelt. Der Client kann elegant mit der Struktur arbeiten.

Es wird eine gemeinsame Schnittstelle für die Elementbehälter (Composite, Kompositum; Aggregat, Knoten) und für die atomaren Elemente (Leaf, Blatt) definiert: Component. Diese Schnittstelle Component definiert die Methoden, die gleichermaßen auf Composites und auf Leafs angewandt werden sollen. Composites delegieren oft Aufrufe (operate()) an ihre Components, die atomare Leafs oder wiederrum zusammengesetzte Composites seien können.

Dies vereinfacht den Clientcode (beispielsweise beim Wandern/Traversieren durch die Struktur oder das Verwalten dieser), da der Client nicht mehr zwischen Composite und Leaf unterscheiden muss und allgemeingültigen Code schreiben kann!