Hochschule Konstanz
Fakultät Informatik

Software Engineering

08 - Design Pattern II

Prof. Dr. Marko Boger

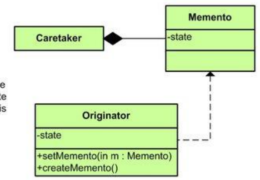
Hochschule Konstanz
Fakultät Informatik

Dienstag, 20. Mai		
Kandidat 1	8:00-8:40	Lehrprobe
	8:40-9:10	Der digitale Lückenschluss zwischen Shopfloor und IT-Welt
Kandidatin 2	11:30-12:10	Lehrprobe
	12:10-12:40	AI in Autonomous Driving: From Perception to Decision
Kandidatin 3	14:20-15:00	Lehrprobe
	15:00-15:30	Green Software Development: AI-supported software development and technologies for sustainable and energy-efficient solutions
Mittwoch, 21. Mai		
Kandidat 4	8:00-8:40	Lehrprobe
	8:40-9:10	Hybrid Optimization: The Interplay Between Classical Techniques and Modern AI
Kandidat 5	11:30-12:10	Lehrprobe
	12:10-12:40	Modernizing Applications Through Architectural Refactoring to Microservices

Pattern Cheat Sheet

- C Abstract Factory
- S Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- B Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor

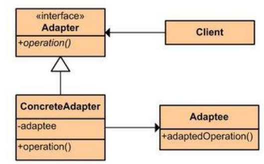
Memento



Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

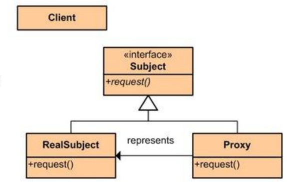
Adapter



Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

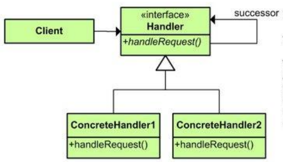
Proxy



Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

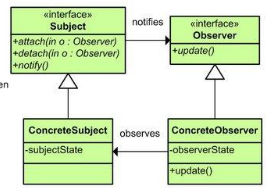
Chain of Responsibility



Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

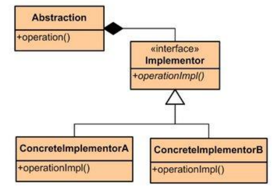
Observer



Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

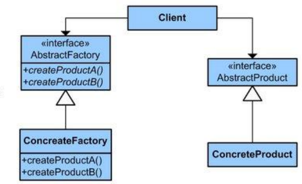
Bridge



Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

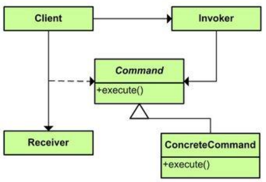
Abstract Factory



Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

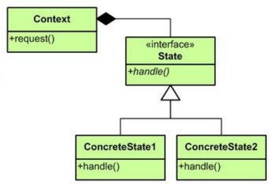
Command



Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

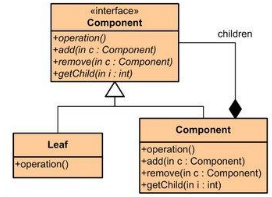
State



Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

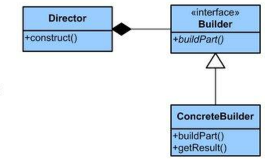
Composite



Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

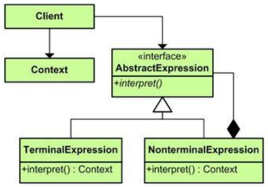
Builder



Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.

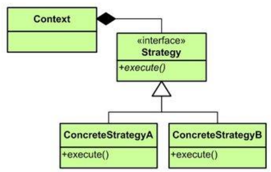
Interpreter



Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

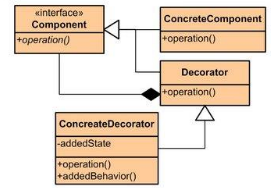
Strategy



Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

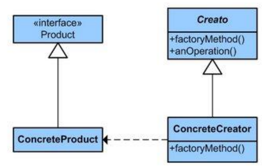
Decorator



Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

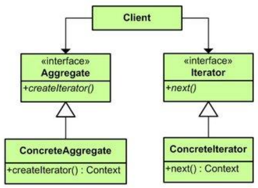
Factory Method



Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

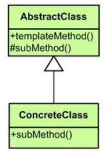
Iterator



Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

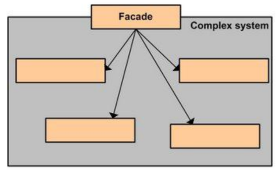
Template Method



Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

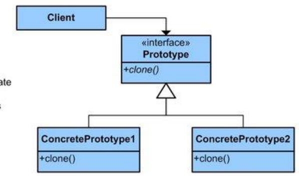
Facade



Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

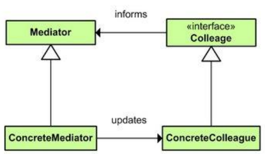
Prototype



Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

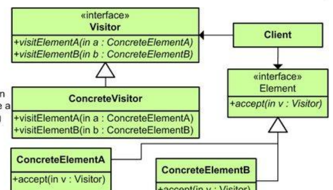
Mediator



Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

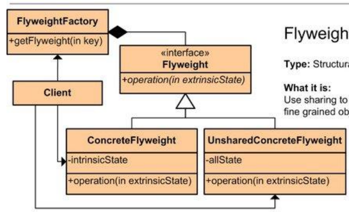
Visitor



Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

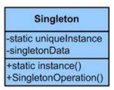
Flyweight



Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Singleton



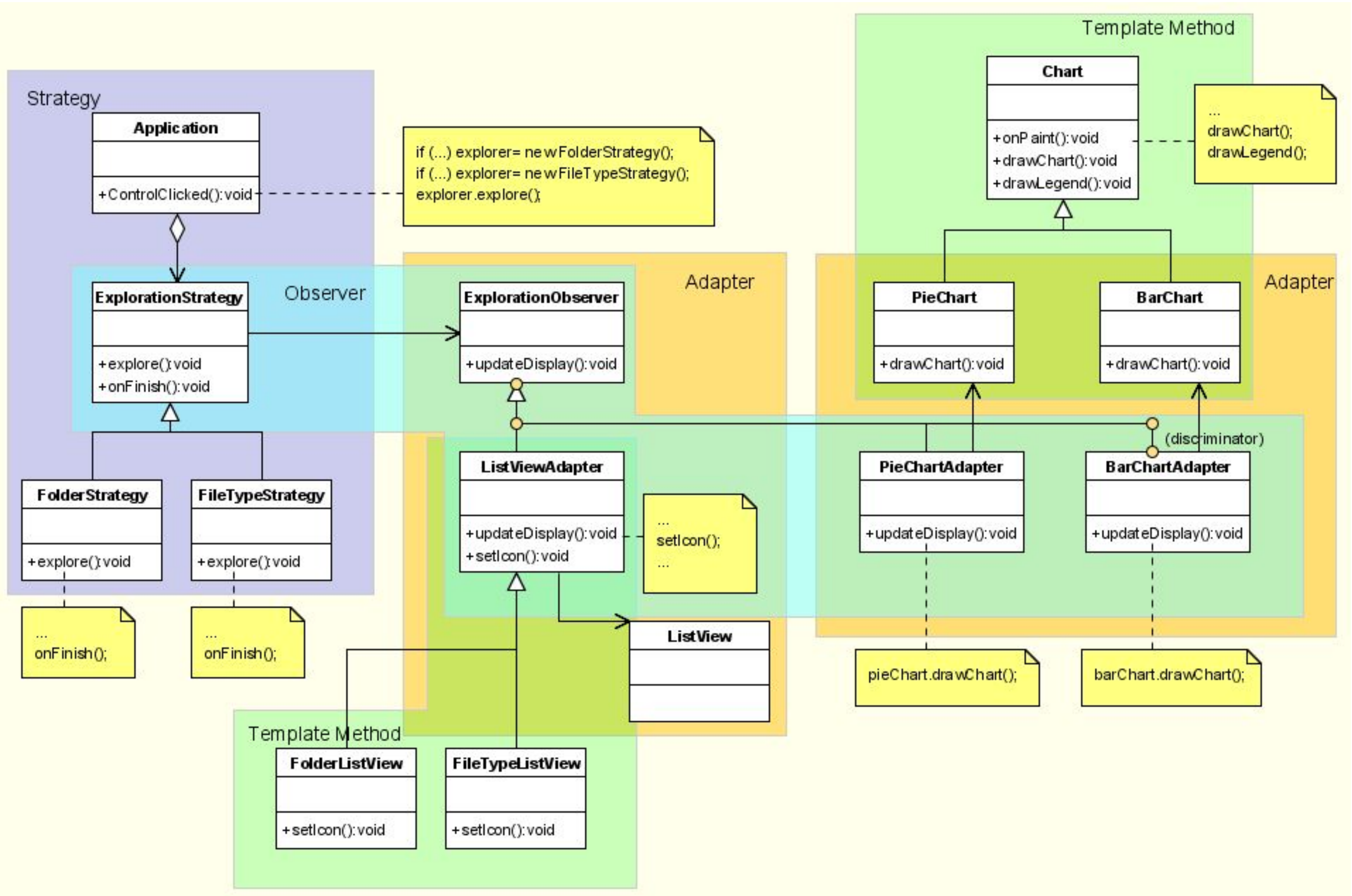
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

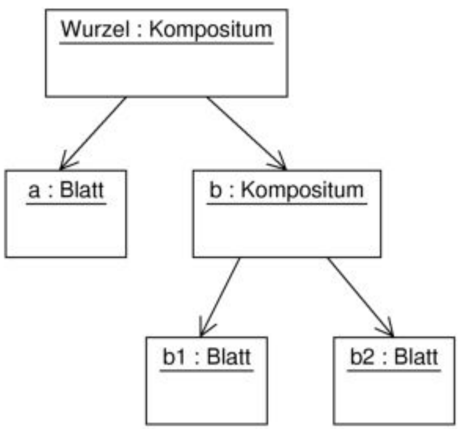
Copyright © 2007 Jason S. McDonald
http://McDonaldLand.wordpress.com
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

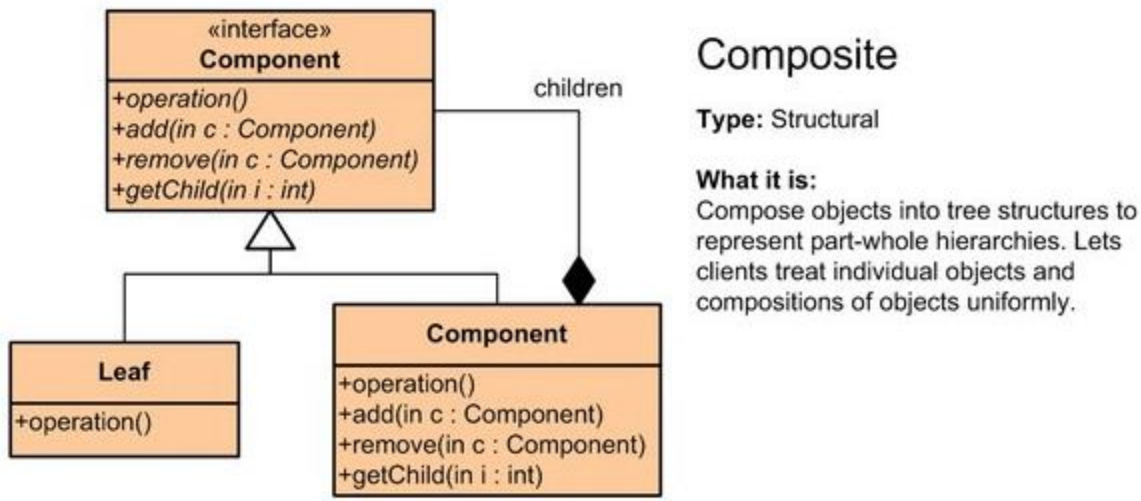
Structure of the Storage Explorer



Composite Pattern

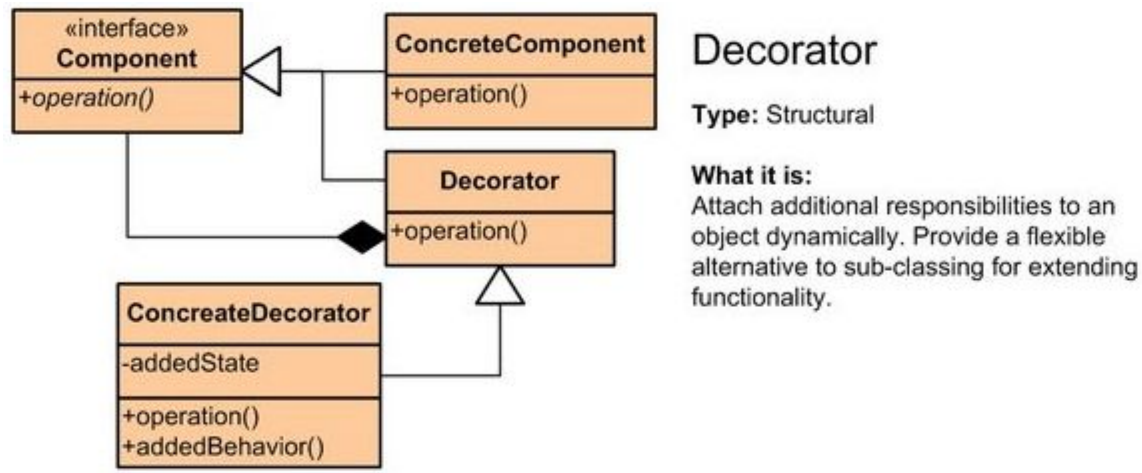
- Composite allows a group of objects to be treated in the same way as a single instance of an object





Decorator

add additional responsibilities dynamically to an object.

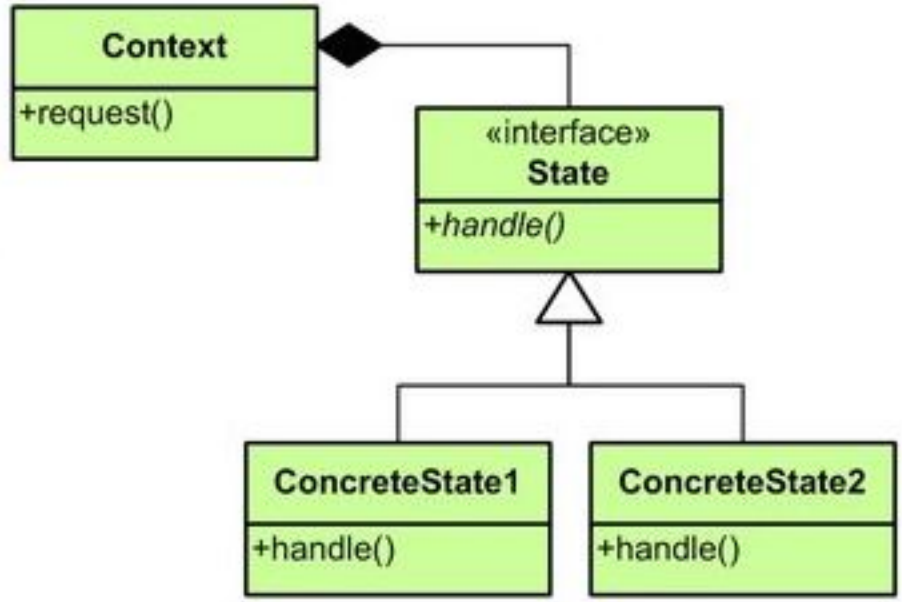


Similar structure, different purpose

State

Type: Behavioral

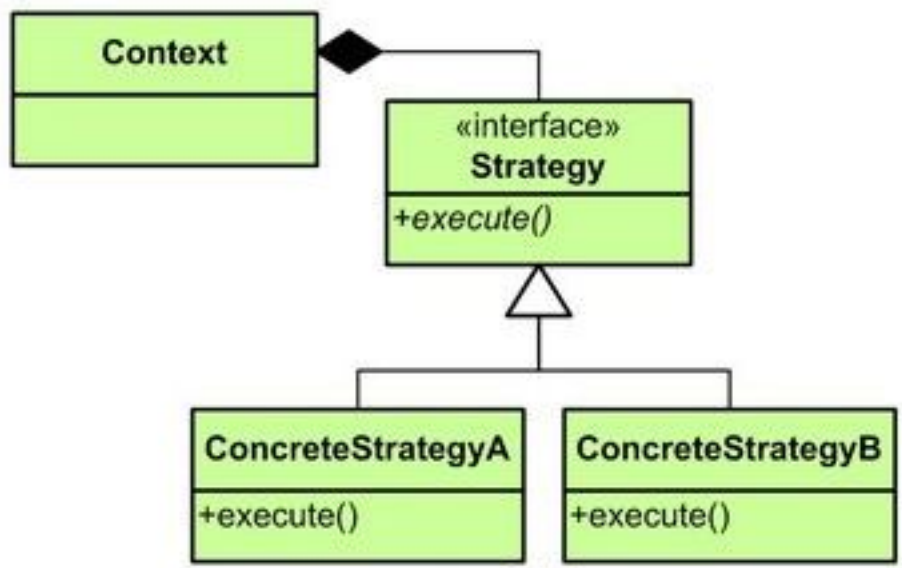
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



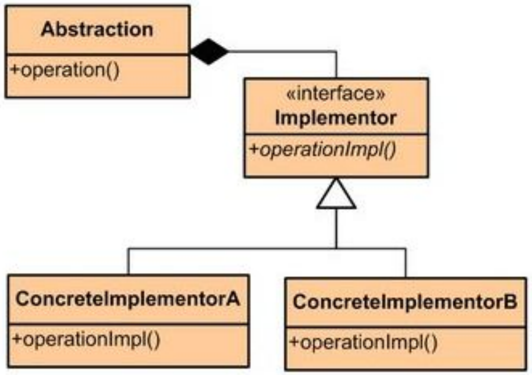
Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Similar structure, different purpose



Bridge

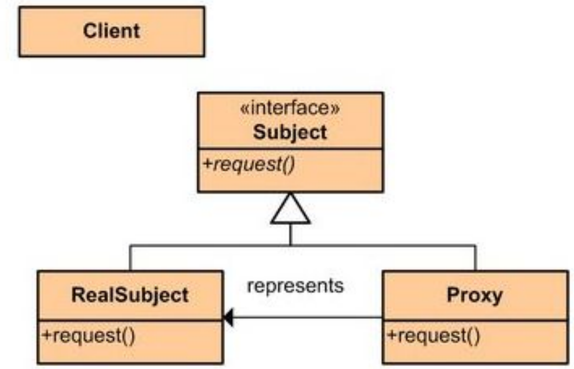
Type: Structural

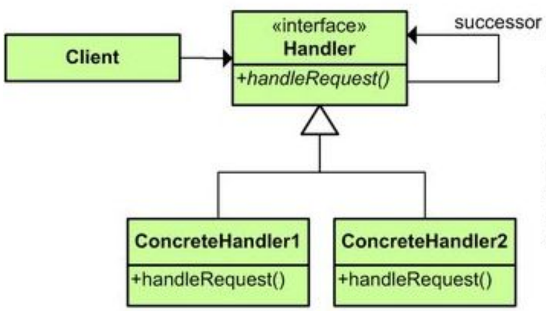
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.





Chain of Responsibility

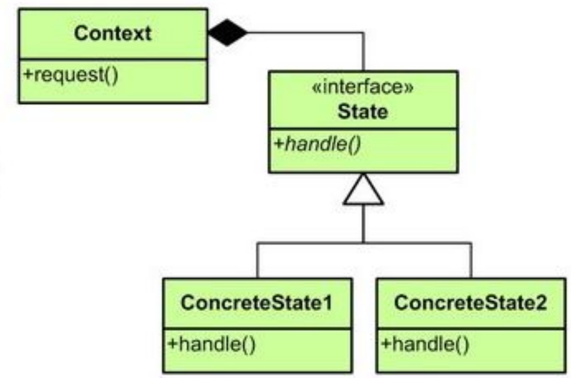
Type: Behavioral

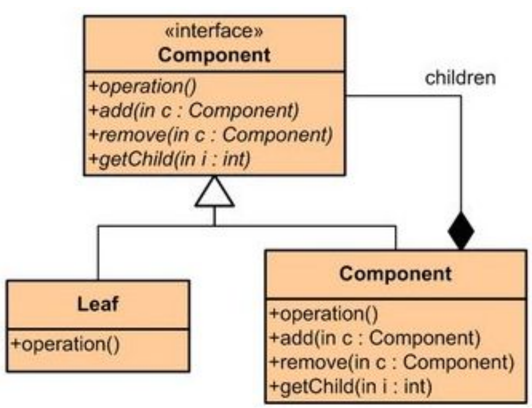
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.





Composite

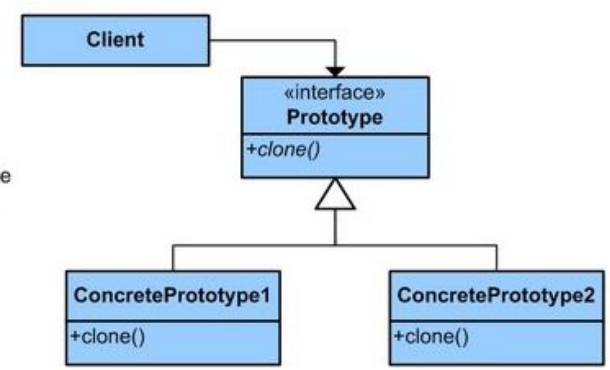
Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

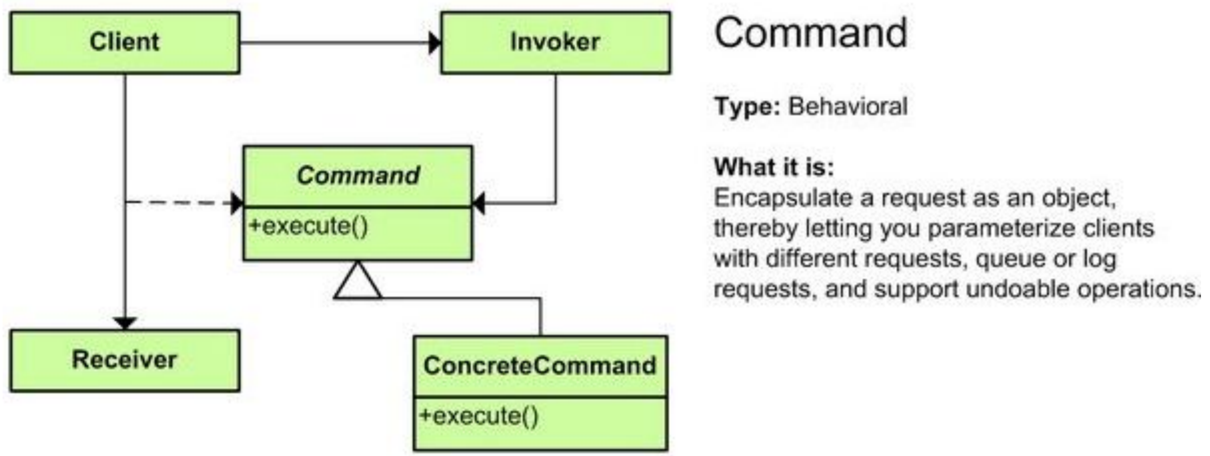


Command Pattern

The Command Pattern is used to encapsulate a method into a class with a normed Interface.

This allows to put the “method” on to a stack and execute it later.

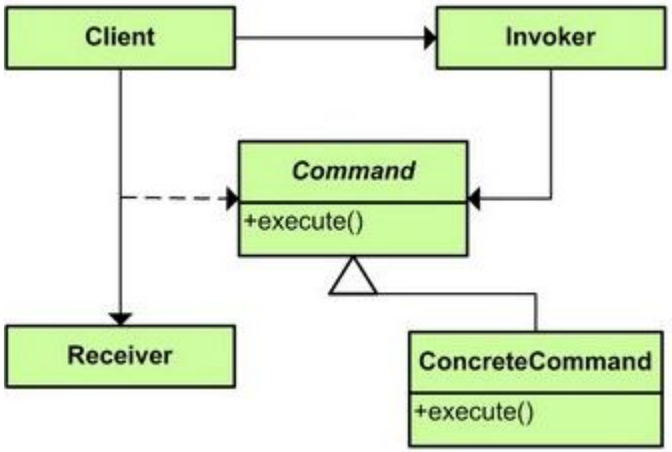
This is often used for Undo-Redo.



Hochschule Konstanz
Fakultät Informatik

An Undo-Mechanism

The Command has an execute, here undoStep



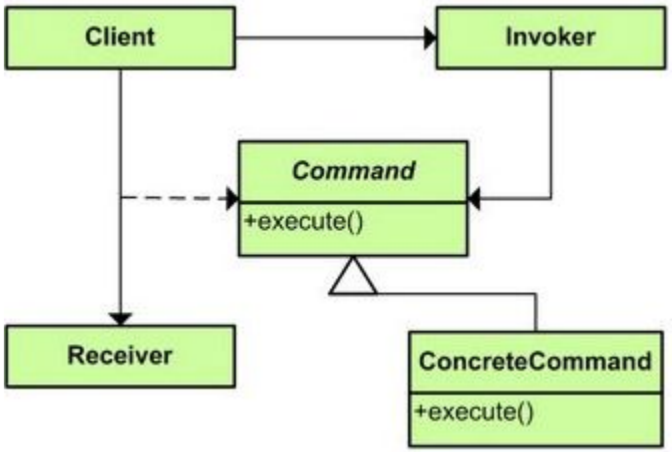

```
trait Command {
 def doStep:Unit
 def undoStep:Unit
 def redoStep:Unit
}
```

```
class SetCommand(row:Int, col: Int, value:Int, controller: Controller) extends Command {
 override def doStep: Unit =   controller.grid = controller.grid.set(row, col, value)
 override def undoStep: Unit = controller.grid = controller.grid.set(row, col, 0)
 override def redoStep: Unit = controller.grid = controller.grid.set(row, col, value)
}
```

An Undo-Mechanism

Hochschule Konstanz
Fakultät Informatik

The Invoker is here called UndoManager




```scala
class UndoManager {
 private var undoStack: List[Command]= Nil
 private var redoStack: List[Command]= Nil
 def doStep(command: Command) = {
   undoStack = command::undoStack
   command.doStep
 }
 def undoStep  = {
   undoStack match {
     case  Nil =>
     case head::stack => {
       head.undoStep
       undoStack=stack
       redoStack= head::redoStack
     }
   }
 }
```

Monads

A Monad is just a Monoid in the Category of Endofunctors

$$
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
$$

https://medium.com/@felix.kuehl/a-monad-is-just-a-monoid-in-the-category-of-endofunctors-lets-actually-unravel-this-f5d4b7dbe5d6

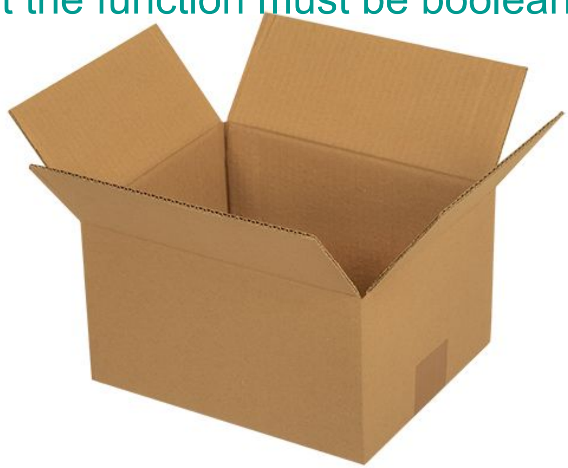
Monad as a Pattern

A Monad is a simple Design Pattern, just like Observer or State Pattern.

It is a container for algebraic structures.

A Monad has to have three methods:

- map
  - it opens the box, applies a function on the content, and but it back into a box
- flatmap
  - like map, but it can handle boxes in boxes
- filter
  - like map, but the function must be boolean



For-Comprehension is syntactic Sugar for map

A simple for-expression with one Generator is expanded to a map function

```
// Translation of For(1)
for (x <- e1) yield toValue(x)
```

```
// translated from compiler to:
e1.map(x => toValue(x) )
```

For can be used for any data structure with map

Complex for is translated to map, flatmap and filter

```
for {
  i <- 1 to n
  j <- 1 to i
  if isEven(i + j)
} yield(i, j)
```

is translated to

```
(1 to n) flatMap ( i =>
  (1 to i) filter (j => isEven(i+j))
    map (j => (i, j))
  )
)
```

Algebraic Structures
Monoid

A monoid is an algebraic structure with a single associative binary operation and an identity element. Monoids are semigroups with identity.

Group-like structures. The entries say whether the property is *required.*

	Totality*	Associativity	Identity	Divisibility	Commutativity
Semicategory	No	Yes	No	No	No
Category	No	Yes	Yes	No	No
Groupoid	No	Yes	Yes	Yes	No
Magma	Yes	No	No	No	No
Quasigroup	Yes	No	No	Yes	No
Loop	Yes	No	Yes	Yes	No
Semigroup	Yes	Yes	No	No	No
Monoid	Yes	Yes	Yes	No	No
Group	Yes	Yes	Yes	Yes	No
Abelian Group	Yes	Yes	Yes	Yes	Yes

*Closure, which is used in many sources, is an equivalent axiom to totality, though defined differently.

Rules for Monoids

- Associativity
  - For all a, b and c in S, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds.
- Identity element
  - There exists an element e in S such that for every element a in S, the equations $e \bullet a = a \bullet e = a$ hold.
- Examples
  - Strings with concat and empty word
  - 12 Hour Clock with add and zero
  - Integers with add and zero
  - Integers with modulo and 1
  - Lists with concat and ()
  - Sets

Monads and Monoids

The Monad acts as a container for a Monoid.

We can take an element out of the container, apply a function or transformation to the element, and put it into a new container.

For example, we can take a bottle from a crate, put a label on it, and put it into a new crate.

Option and Try are Monads

- Try and Option are Monads for handling a single content
- Think of it as a container with just one content for secure transportation
  - Like a box for a bottle of wine

Option is a Monad

Option can contain Some(thing) or None

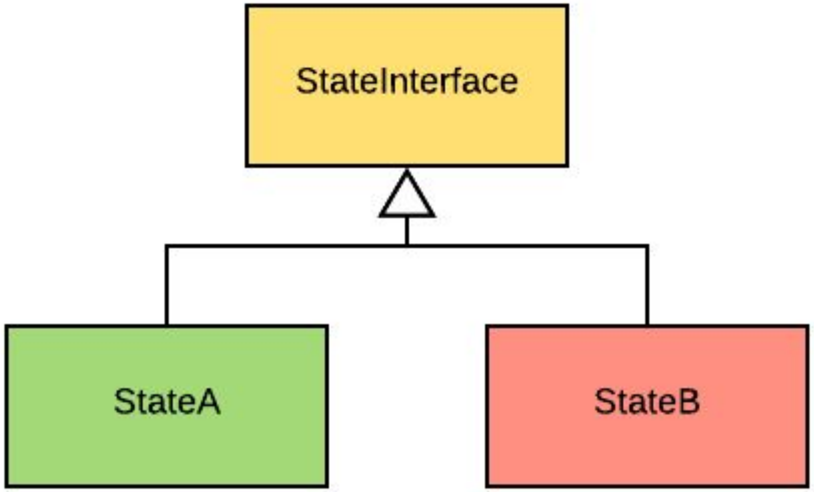
Try is a Monad

- A Try can contain a Success or a Failure

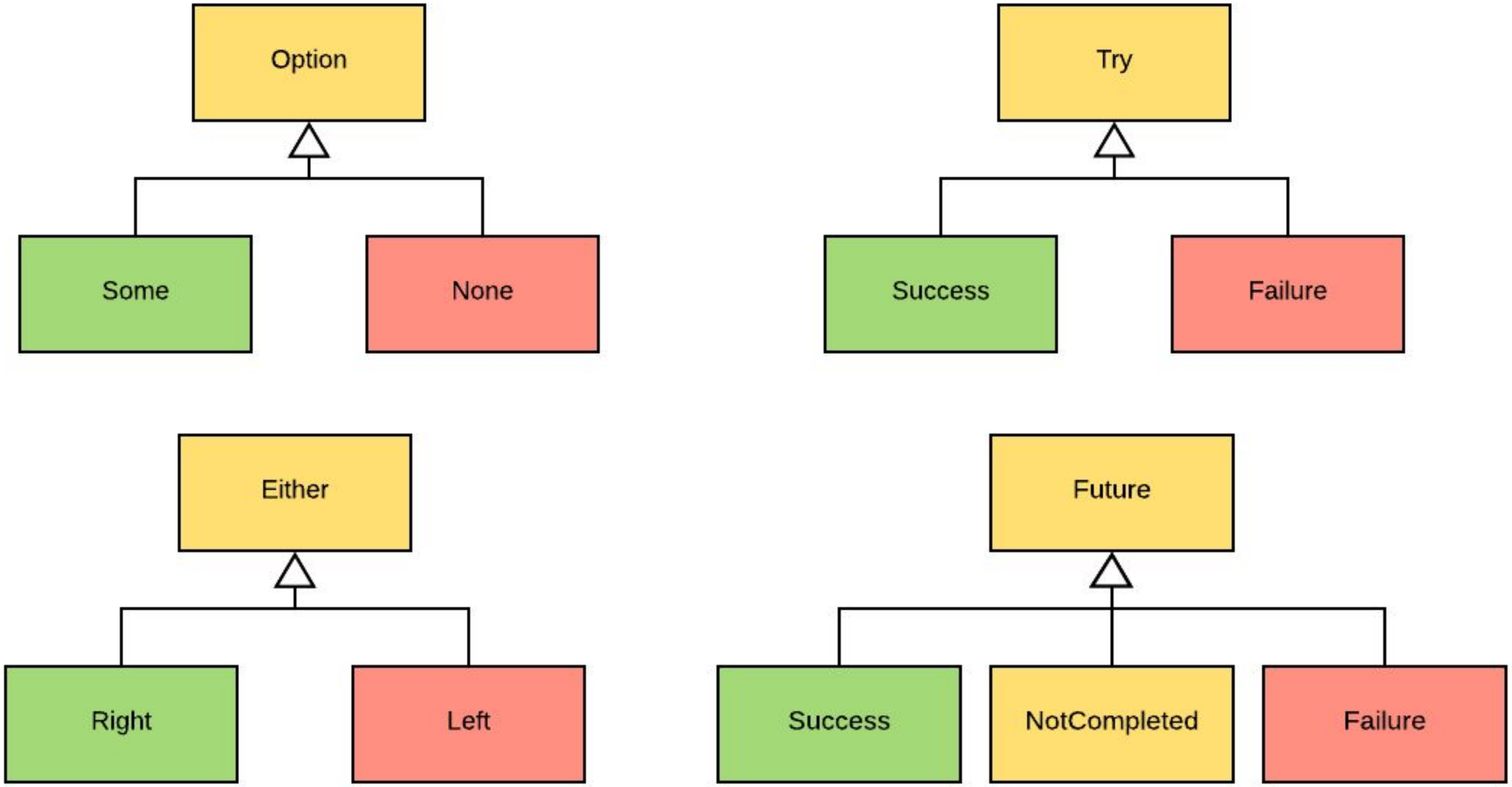
The State Pattern applied in Monads

Take a look again at the structure of the State Pattern.

Monads use this structure and typically distinguish a “good” and a “bad” state.



Option, Try, Future and Either



An Example using Bottles

```
class Bottle {
 var empty=false
def consume = {
 println(" consuming... ")
 empty = true
 this
}
override def toString= if (empty) "b" else "B"
}

class Pack(val bottles:List[Bottle]) {
 def map(f:Bottle => Bottle) = bottles.map(bottle => f(bottle))
 override def toString="UUUU"
}

class Crate(val packs:List[Pack]) {
 def map(f:Pack => Pack) = packs.map(pack => f(pack))
 def flatMap(f:Pack => List[Bottle]) = packs.flatMap(pack => f(pack))
 override def toString="L_____J"
}
```

The Good Case

```scala
val pack1 = new Pack(List(new Bottle, new Bottle, new Bottle, new Bottle))

val pack2 = new Pack(List(new Bottle, new Bottle, new Bottle, new Bottle))

val crate1 = new Crate(List(pack1, pack2))

def consumeAll(crate:Crate) = {
 var packs = crate.packs
 packs.foreach {pack =>
  var bottles = pack.bottles
  bottles.foreach { bottle =>
   bottle.consume
  }
 }
}

consumeAll(crate1)
```

The Bad Case

Hochschule Konstanz
Fakultät Informatik

```scala
val pack3 = new Pack(List(new Bottle, null, new Bottle, new Bottle))
val pack4 = new Pack(List(new Bottle, new Bottle, null, new Bottle))
val crate2 = new Crate(List(pack3, pack4))
def consumeAllAssumeNull(crate:Crate) = {
 if (crate != null) {
  var packs = crate2.packs
  if (packs != null) {
    packs.foreach { pack =>
     if (pack != null) {
      var bottles = pack.bottles
      if (bottles != null) {
       bottles.foreach { bottle =>
        if (bottle != null) bottle.consume
        else println ("Found a null for bottle")
       }
      } else println ("Found a null for bottles")
     } else println ("Found a null for pack")
    }
  } else println ("Found a null for packs")
 }  else println ("Found a null for crate")
}
consumeAllAssumeNull(crate2)
```

Bottle using Monads

```
class PackT[T](val bottles:List[T]) {
 def map(f:T => T) = bottles.map(bottle => f(bottle))
 override def toString="UUUU"
}

class CrateT[T](val packs:List[T]) {
 def map(f:T => T) = packs.map(pack => f(pack))
 def flatMap(f:T => List[Bottle]) = packs.flatMap(pack => f(pack))
 override def toString="L_____J"
}
// Option already exists. This is a sketch of an implementation.
trait Option[Bottle] {
def map(f:Bottle => Bottle):Option[Bottle]
}
case class Some[Bottle](val b:Bottle) extends Option[Bottle] {
def map(f:Bottle => Bottle) = new Some(f(b))
}
case class None[Bottle]() extends Option[Bottle] {
def map(f:Bottle => Bottle) = new None
}
```

Using For to unpack the Monad

```
val maybeBottle:Option[Bottle] = Some(new Bottle)
val pack5= new PackT( List(Some(new Bottle), None, Some(new Bottle), Some(new
Bottle)))
val pack6= new PackT( List(Some(new Bottle), Some(new Bottle), None, Some(new
Bottle)))
val crate3 = new CrateT(List(Some(pack5),Some(pack6)))


def consumeAllAssumeNoneWithFor(pack:PackT[Option[Bottle]]) = {
 for (
    bottle <- pack.bottles) yield bottle match {
  case Some(b) => b.consume
  case None => println("Found None")
 }
}
consumeAllAssumeNoneWithFor(pack6)
```

Task 8: Integrate Undo, Make use of Try and Option

Implement an Undo mechanism using the Command Pattern.

Avoid the use of null and Null. Use Option instead. Also avoid using try-catch, use the Try-Monad instead.