TDDE45 - Lecture 2: Design Patterns

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

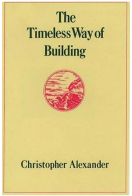
2023-08-30

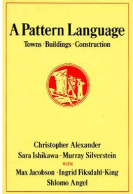
Part I

Intro

Brief History (1)

General concept of *patterns* (253 of them, for architecture in the buildings sense):





Similarities between software design and architecture was noted by Smith 1987.

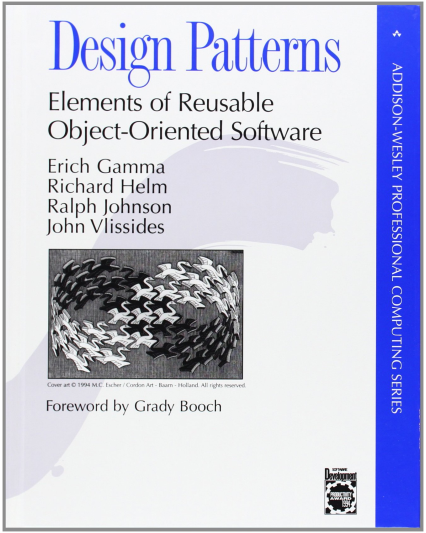
Brief History (2)

By 1995, the Design Patterns book by the Gang of Four was published (Gamma et al. 1995).

Many of the patterns are based on existing idioms in programming languages.

New patterns have been created over the years:

Kind	GoF	Wikipedia
Creational	5	10
Structural	7	12
Behavioral	11	15
Concurrency	0	16



Principles + Problem = Pattern

Principles = SOLID + Some general tips

SOLID

1. Encapsulate what varies (S)
2. Program to an interface, not to an implementation (I, D)
3. Favor Composition over Inheritance (L)
4. Don't call us, we'll call you (O)

Some more tips

5. Depend upon abstractions, not upon concrete classes (see 2).
6. Strive for loosely coupled designs between objects that interact (see 4).
7. Only talk to your friends.
8. Avoid global variables (constants can be fine), static methods (thread-safe code).
9. Simple, readable code is often favorable over strictly adhering to the design principles.

Part II

Some preparation for seminar 1

Full instructions are on the course homepage

One of your tasks is to:

> *Read specifically the Intent, Motivation, Applicability and Structure of 4 design patterns per person in the Gang of Four course book (or the corresponding parts in another source such as Head First Design Patterns).*

Structure of the book

The Gang of Four book is very structured; the following is a summary of section 1.3:

- Pattern name and classification (creational, structural, behavioral; class or object)
- Intent
- Also known as
- Motivation
- Applicability – what poor designs can this pattern solve?
- Structure – graphical representation (using OMT – a predecessor to UML (1997))
- Participants – classes or objects in the design pattern
- Collaborations – related to participants
- Consequences – trade-offs?
- Implementation – pitfalls, hints?
- Sample code (C++ or smalltalk)
- Known uses (from real code; you could of course list Eclipse on every design pattern)
- Related patterns – many patterns do similar things; how do they differ? Which design patterns can you combine with it?

Part III

Some Design Patterns

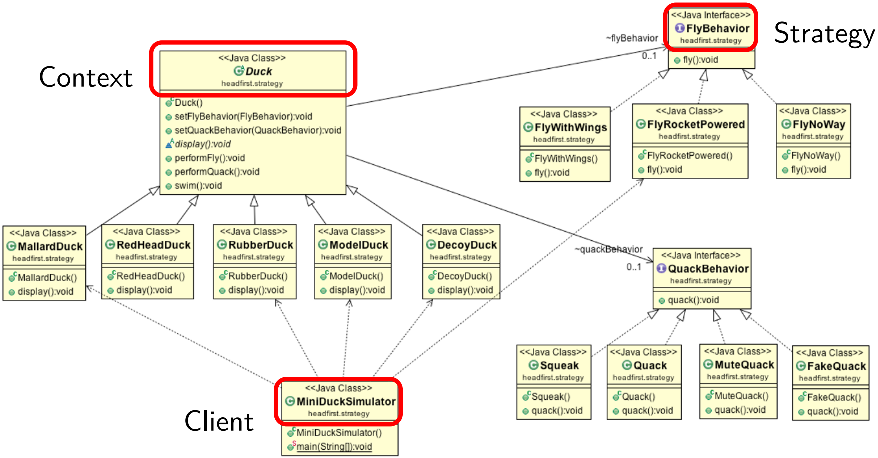
Outline

- Strategy
- Factory Method
- Decorator
- Template Method
- Composite
- Abstract Factory (+ Dependency Injection)
- Singleton (+ example in Ruby)
- Builder
- Adapter
- Bridge
- Observer
- Chain of Responsibility
- Memento
- Command

Part IV

Strategy

Strategy



Context
Strategy
Client
<<Java Class>>
Duck
headfirst.strategy
Duck()
setFlyBehavior(FlyBehavior):void
setQuackBehavior(QuackBehavior):void
display():void
performFly():void
performQuack():void
swim():void
~flyBehavior
0..1
<<Java Interface>>
FlyBehavior
headfirst.strategy
fly():void
<<Java Class>>
FlyWithWings
headfirst.strategy
FlyWithWings()
fly():void
<<Java Class>>
FlyRocketPowered
headfirst.strategy
FlyRocketPowered()
fly():void
<<Java Class>>
FlyNoWay
headfirst.strategy
FlyNoWay()
fly():void
<<Java Class>>
MallardDuck
headfirst.strategy
MallardDuck()
display():void
<<Java Class>>
RedHeadDuck
headfirst.strategy
RedHeadDuck()
display():void
<<Java Class>>
RubberDuck
headfirst.strategy
RubberDuck()
display():void
<<Java Class>>
ModelDuck
headfirst.strategy
ModelDuck()
display():void
<<Java Class>>
DecoyDuck
headfirst.strategy
DecoyDuck()
display():void
~quackBehavior
0..1
<<Java Interface>>
QuackBehavior
headfirst.strategy
quack():void
<<Java Class>>
Squeak
headfirst.strategy
Squeak()
quack():void
<<Java Class>>
Quack
headfirst.strategy
Quack()
quack():void
<<Java Class>>
MuteQuack
headfirst.strategy
MuteQuack()
quack():void
<<Java Class>>
FakeQuack
headfirst.strategy
FakeQuack()
quack():void
<<Java Class>>
MiniDuckSimulator
headfirst.strategy
MiniDuckSimulator()
main(String[]):void

Strategy: Consequences

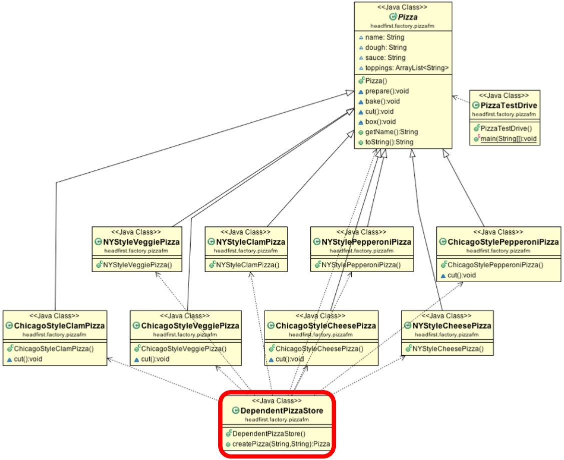
+ Can choose implementation of a strategy at run time
+ Eliminate hardcoded conditionals
+ Avoids excessive subclassing

- Clients must be aware of different strategies
- Communication required between context and strategies
- Potentially many strategy objects created

Part V

Factory Method

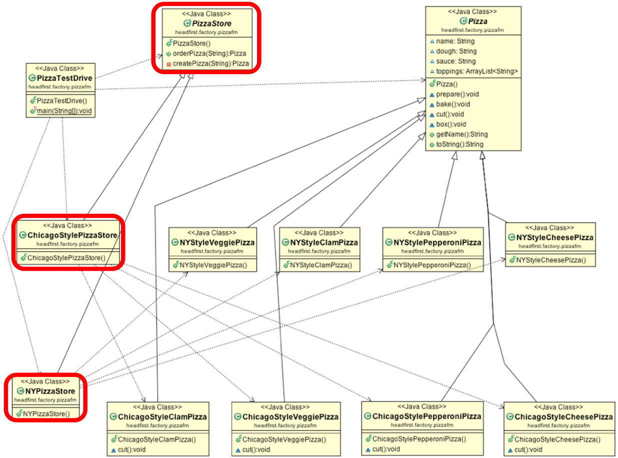
Factory method (before)



Factory method (before)

```
Pizza pizza = null;
if (style.equals("NY")) {
  if (type.equals("cheese")) {
    pizza = new NYStyleCheesePizza();
  } else if (type.equals("veggie")) {
    pizza = new NYStyleVeggiePizza();
  } else if (type.equals("clam")) {
    pizza = new NYStyleClamPizza();
  } else if (type.equals("pepperoni")) {
    pizza = new NYStylePepperoniPizza();
  }
} else if (style.equals("Chicago")) {
  if (type.equals("cheese")) {
    pizza = new ChicagoStyleCheesePizza();
  } else if (type.equals("veggie")) {
    pizza = new ChicagoStyleVeggiePizza();
  } else if (type.equals("clam")) {
    pizza = new ChicagoStyleClamPizza();
  } else if (type.equals("pepperoni")) {
    pizza = new ChicagoStylePepperoniPizza();
  }
} else {
  System.out.println("Error: invalid type of pizza");
  return null;
}
```

Factory method



Factory method

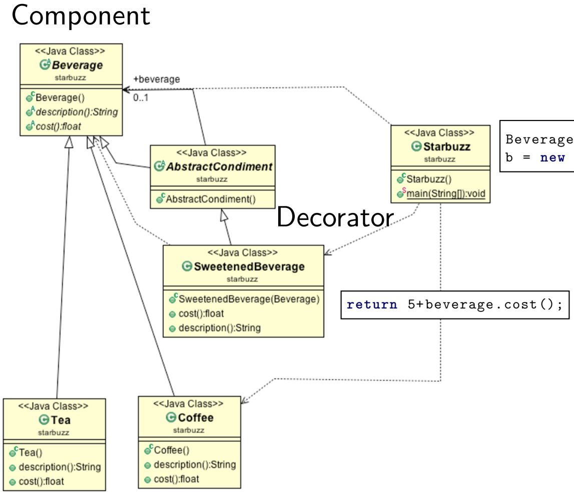
- + Decouples clients from specific dependency classes
- + Eliminates hardcoded conditionals
- + Connects parallel class hierarchies (NY*Pizza, Chicago*Pizza)

- - Requires keeping factory methods in sync with domain classes

Part VI

Decorator

Decorator




```java
Beverage b = new Coffee();
b = new SweetenedBeverage(new SweetenedBeverage(b));
```

Decorator

+ Dynamically adds behavior to specific instances of a class
+ Customizes an abstract class without knowing the implementations

- Decorator objects are not of the same type as the objects it comprises
- May result in many small objects

Part VII

Template Method

Template Method

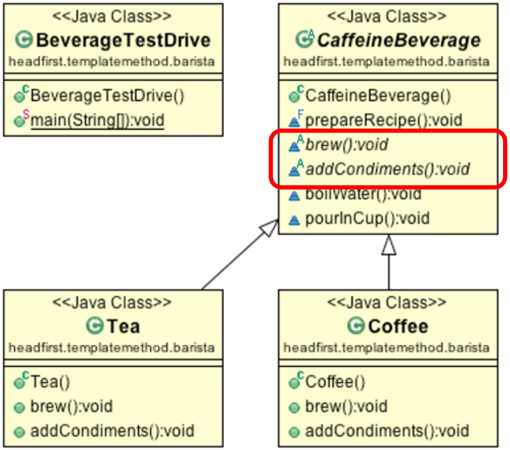
```
class Coffee{
public:
    void prepareRecipe();
    void boilWater();
    void brewCoffeeGrinds();
    void pourInCup();
    void addSugarAndMilk();
};
class Tea{
public:
    void prepareRecipe();
    void boilWater();
    void steepTeaBag();
    void pourInCup();
    void addLemon();
};
```

```
class Beverage {
public:
    void prepareRecipe();
    void boilWater();
    void pourInCup();
    // No brew==steep
    // No addCondiments
};
```

Template method: Consequences

- + Can isolate the extensions possible to an algorithm
- + Isolates clients from algorithm changes

Template method



Default implementations (hooks)

```
public abstract class CaffeineBeverageWithHook {
  void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    if (customerWantsCondiments()) {
      addCondiments();
    }
  }
  boolean customerWantsCondiments() {
    return true;
  }
}

public class CoffeeWithHook extends CaffeineBeverageWithHook {
  // ...
  public boolean customerWantsCondiments() {
    return getUserInput().toLowerCase().startsWith("y");
  }
}
```

Template method?

```java
Duck[] ducks = {
        new Duck("Daffy", 8),
        new Duck("Dewey", 2),
        new Duck("Howard", 7),
        new Duck("Louie", 2),
        new Duck("Donald", 10),
        new Duck("Huey", 2)
 };

Arrays.sort(ducks, new Comparator<Duck>(){

  @Override
  public int compare(Duck arg0, Duck arg1) {
    return new Integer(arg1.weight).compareTo(arg0.
          weight);
  }
});
```

No

```java
public class Duck implements Comparable<Duck> {
  String name;
  int weight;

  public Duck(String name, int weight) {
    this.name = name;
    this.weight = weight;
  }

  public String toString() {
    return MessageFormat.format("{0} weighs {1}",
          name, weight);
  }

  public int compareTo(Duck object) {
    return new Integer(this.weight).compareTo(object
          .weight);
  }
}
```

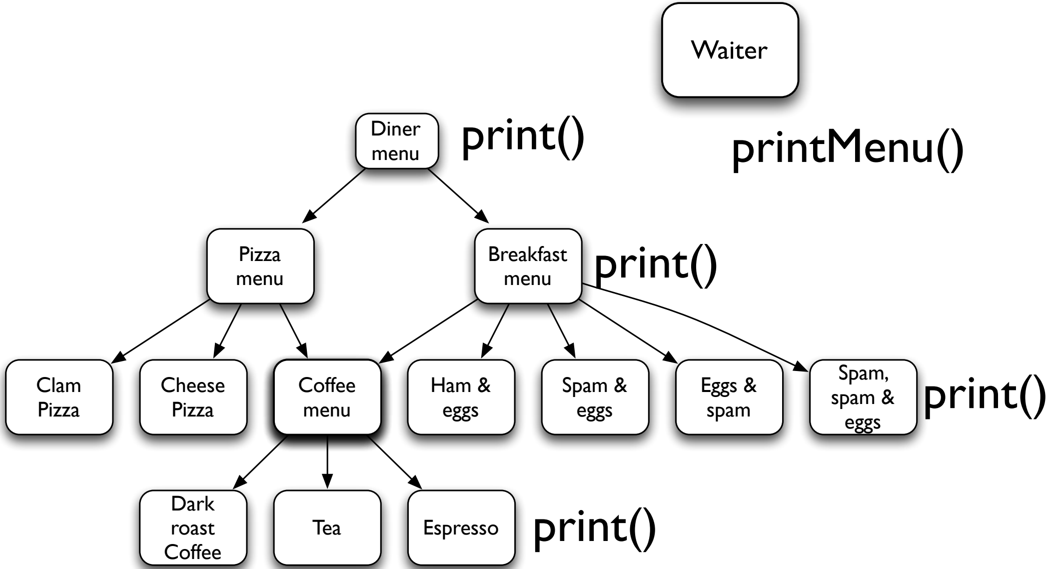
Yes

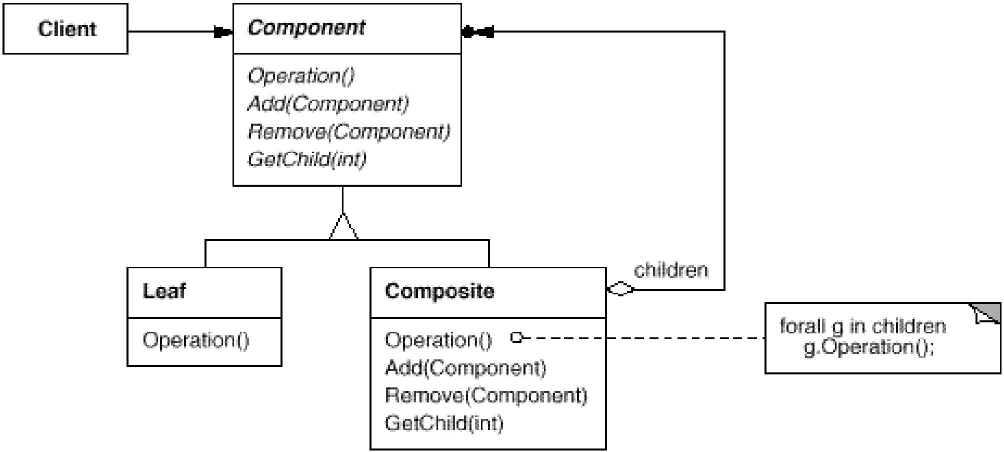
Java 8: With inline lambda (not Template method)

```
Arrays.sort(ducks, (arg0, arg1) -> new Integer(arg1.weight)
    .compareTo(arg0.weight));
```

Part VIII

Composite


Waiter
printMenu()
Diner menu
print()
Pizza menu
Breakfast menu
print()
Clam Pizza
Cheese Pizza
Coffee menu
Ham & eggs
Spam & eggs
Eggs & spam
Spam, spam & eggs
print()
Dark roast Coffee
Tea
Espresso
print()

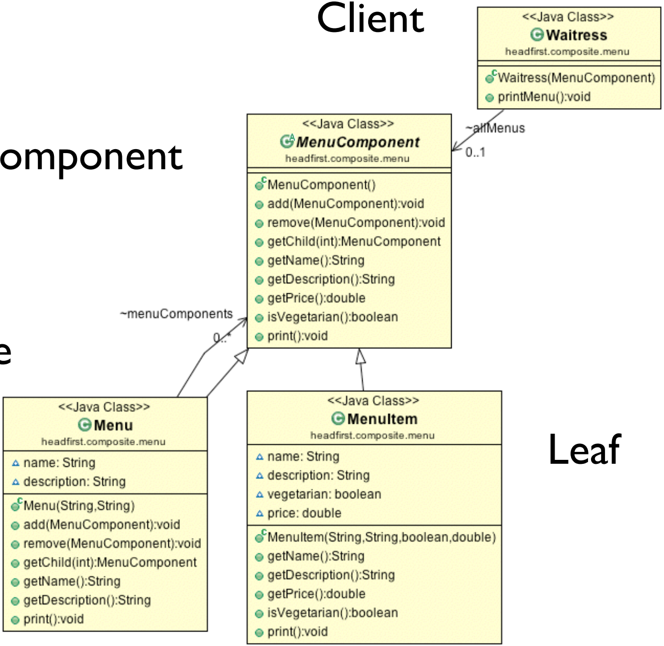

Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Client

Component

Composite

Leaf



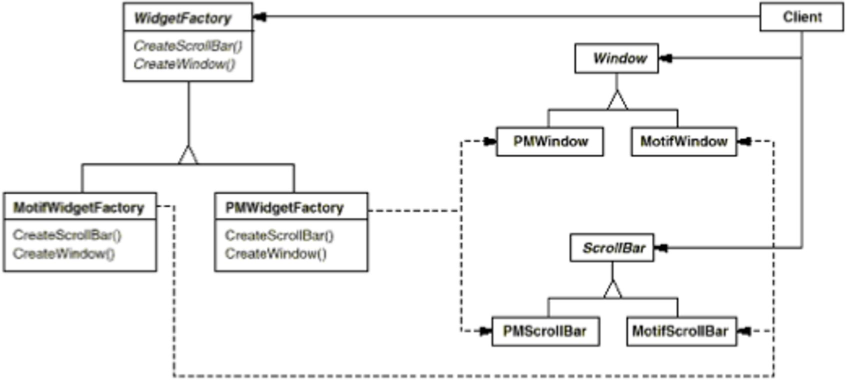
Composite: consequences

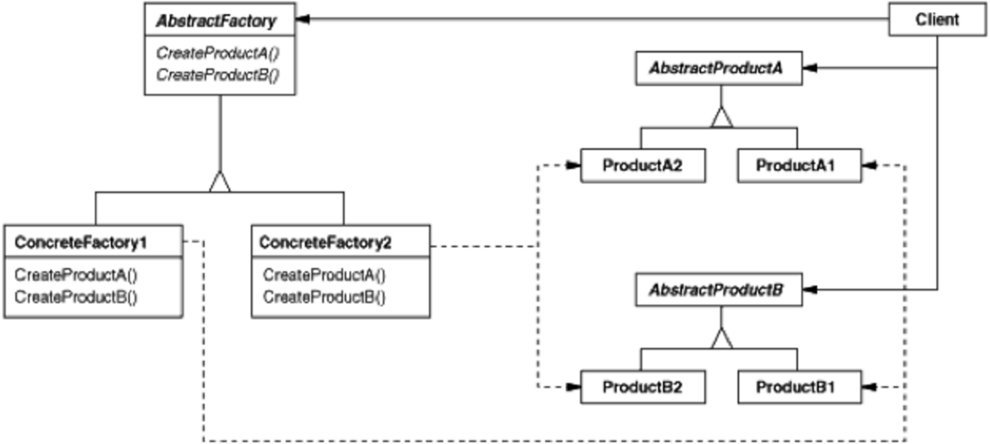
+ Allow us to treat composite objects and individual objects uniformly
+ Allows arbitrarily complex trees

- Creates composite classes that violate the principle of a single responsibility
- The composite cannot rely on components to implement all methods

Part IX

Abstract Factory


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

Ingredients | Pizza Store | Clients

Fresh Clam

Mozzarella Cheese

Thin Crust Dough

NY

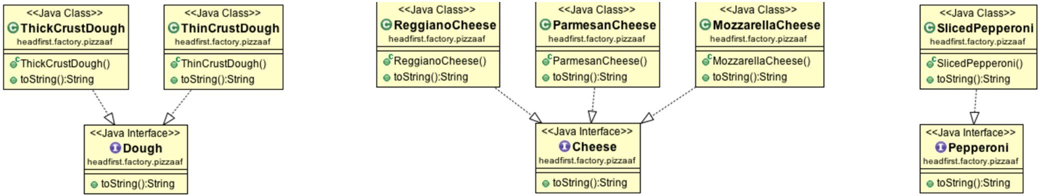
I Want a Cheese Pizza

Frozen Clam

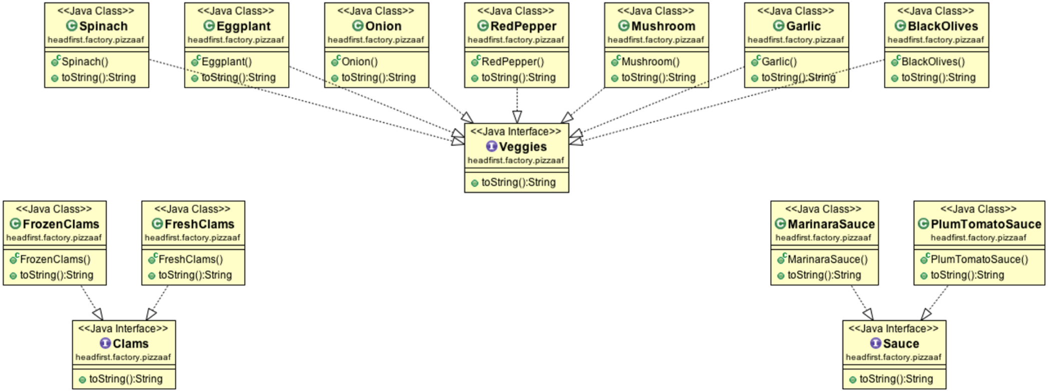
Parmesan Cheese

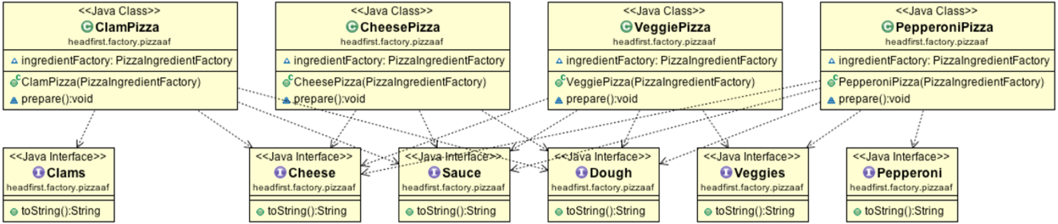
Thick Crust Dough

Chicago

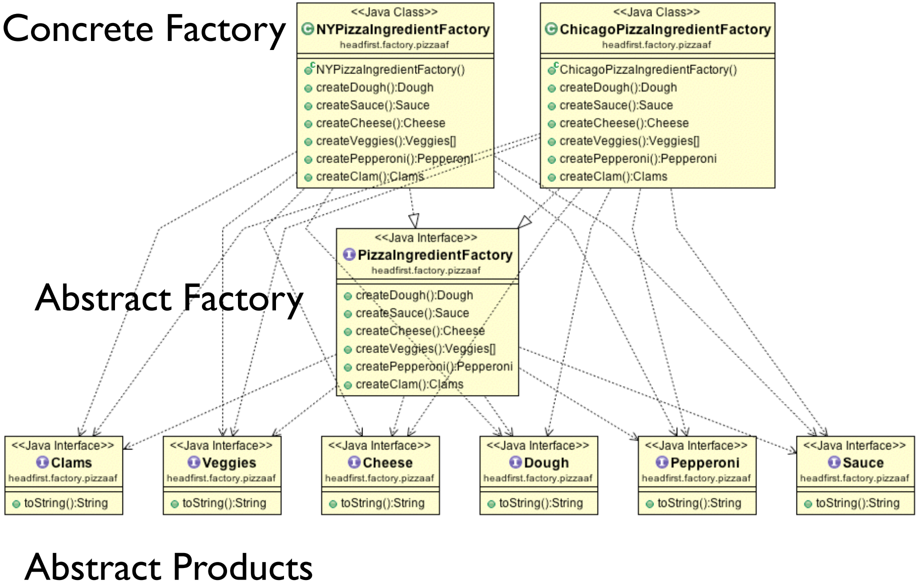


<<Java Class>>
ThickCrustDough
headfirst.factory.pizzaaf
ThickCrustDough()
toString():String
<<Java Class>>
ThinCrustDough
headfirst.factory.pizzaaf
ThinCrustDough()
toString():String
<<Java Interface>>
Dough
headfirst.factory.pizzaaf
toString():String
<<Java Class>>
ReggianoCheese
headfirst.factory.pizzaaf
ReggianoCheese()
toString():String
<<Java Class>>
ParmesanCheese
headfirst.factory.pizzaaf
ParmesanCheese()
toString():String
<<Java Class>>
MozzarellaCheese
headfirst.factory.pizzaaf
MozzarellaCheese()
toString():String
<<Java Interface>>
Cheese
headfirst.factory.pizzaaf
toString():String
<<Java Class>>
SlicedPepperoni
headfirst.factory.pizzaaf
SlicedPepperoni()
toString():String
<<Java Interface>>
Pepperoni
headfirst.factory.pizzaaf
toString():String

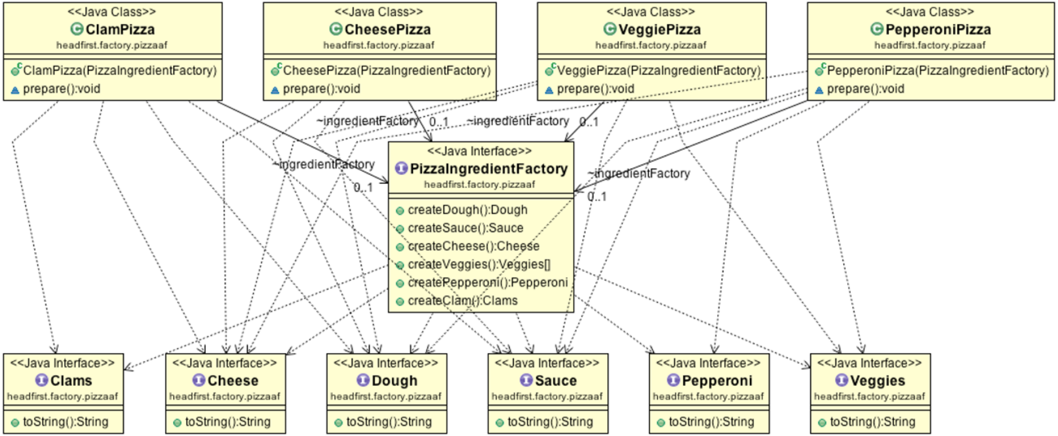

<<Java Class>>
Spinach
headfirst.factory.pizzaaf
Spinach()
toString():String
<<Java Class>>
Eggplant
headfirst.factory.pizzaaf
Eggplant()
toString():String
<<Java Class>>
Onion
headfirst.factory.pizzaaf
Onion()
toString():String
<<Java Class>>
RedPepper
headfirst.factory.pizzaaf
RedPepper()
toString():String
<<Java Class>>
Mushroom
headfirst.factory.pizzaaf
Mushroom()
toString():String
<<Java Class>>
Garlic
headfirst.factory.pizzaaf
Garlic()
toString():String
<<Java Class>>
BlackOlives
headfirst.factory.pizzaaf
BlackOlives()
toString():String
<<Java Interface>>
Veggies
headfirst.factory.pizzaaf
toString():String
<<Java Class>>
FrozenClams
headfirst.factory.pizzaaf
FrozenClams()
toString():String
<<Java Class>>
FreshClams
headfirst.factory.pizzaaf
FreshClams()
toString():String
<<Java Interface>>
Clams
headfirst.factory.pizzaaf
toString():String
<<Java Class>>
MarinaraSauce
headfirst.factory.pizzaaf
MarinaraSauce()
toString():String
<<Java Class>>
PlumTomatoSauce
headfirst.factory.pizzaaf
PlumTomatoSauce()
toString():String
<<Java Interface>>
Sauce
headfirst.factory.pizzaaf
toString():String



<<Java Class>>
ClamPizza
headfirst.factory.pizzaaf
ingredientFactory: PizzaIngredientFactory
ClamPizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
CheesePizza
headfirst.factory.pizzaaf
ingredientFactory: PizzaIngredientFactory
CheesePizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
VeggiePizza
headfirst.factory.pizzaaf
ingredientFactory: PizzaIngredientFactory
VeggiePizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
PepperoniPizza
headfirst.factory.pizzaaf
ingredientFactory: PizzaIngredientFactory
PepperoniPizza(PizzaIngredientFactory)
prepare():void
<<Java Interface>>
Clams
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Cheese
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Sauce
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Dough
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Veggies
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Pepperoni
headfirst.factory.pizzaaf
toString():String


Concrete Factory
<<Java Class>>
NYPizzaIngredientFactory
headfirst.factory.pizzaaf
NYPizzaIngredientFactory()
createDough():Dough
createSauce():Sauce
createCheese():Cheese
createVeggies():Veggies[]
createPepperoni():Pepperoni
createClam():Clams
<<Java Class>>
ChicagoPizzaIngredientFactory
headfirst.factory.pizzaaf
ChicagoPizzaIngredientFactory()
createDough():Dough
createSauce():Sauce
createCheese():Cheese
createVeggies():Veggies[]
createPepperoni():Pepperoni
createClam():Clams
Abstract Factory
<<Java Interface>>
PizzaIngredientFactory
headfirst.factory.pizzaaf
createDough():Dough
createSauce():Sauce
createCheese():Cheese
createVeggies():Veggies[]
createPepperoni():Pepperoni
createClam():Clams
<<Java Interface>>
Clams
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Veggies
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Cheese
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Dough
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Pepperoni
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Sauce
headfirst.factory.pizzaaf
toString():String
Abstract Products

Clients


<<Java Class>>
ClamPizza
headfirst.factory.pizzaaf
ClamPizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
CheesePizza
headfirst.factory.pizzaaf
CheesePizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
VeggiePizza
headfirst.factory.pizzaaf
VeggiePizza(PizzaIngredientFactory)
prepare():void
<<Java Class>>
PepperoniPizza
headfirst.factory.pizzaaf
PepperoniPizza(PizzaIngredientFactory)
prepare():void
~ingredientFactory 0..1
~ingredientFactory 0..1
~ingredientFactory 0..1
~ingredientFactory 0..1
<<Java Interface>>
PizzaIngredientFactory
headfirst.factory.pizzaaf
createDough():Dough
createSauce():Sauce
createCheese():Cheese
createVeggies():Veggies[]
createPepperoni():Pepperoni
createClam():Clams
<<Java Interface>>
Clams
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Cheese
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Dough
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Sauce
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Pepperoni
headfirst.factory.pizzaaf
toString():String
<<Java Interface>>
Veggies
headfirst.factory.pizzaaf
toString():String

Abstract factory: consequences

- + Isolates clients from concrete dependencies
- + Makes interchanging families of products easier

Strategy – behavioural

- When related classes only differ in behaviour
- You need different variants of an algorithm
- An algorithm uses data the clients don't need to know
- A class uses conditionals for selecting behavior

Abstract factory – creational

- A system should be independent of how its products are created
- A system should be configured with one of multiple families of products
- You want to provide a class library of products, and only expose their interfaces

Design principles - Abstract Factory

- Encapsulate what varies
- Program to an interface, not to an implementation
- Favor composition over inheritance
- Classes should be open for extension but closed for modification
- Don't call us, we'll call you

Part X

Dependency Injection

Dependency injection: How?

1. Declare dependencies as constructor arguments of interface types
2. Register classes (components) in an Inversion-of-Control Container
3. Resolve the top-level object from an interface through the Container

1. Dependencies

```
namespace DITest {
  public class FancyClamPizza: IClamPizza {
    private IClam clam;
    private ICheese cheese;
    public FancyClamPizza (IClam clam, ICheese cheese) {
      this.clam = clam;
      this.cheese = cheese;
    }
    public String ClamType() {
      return String.Format("fancy {0}",clam);
    }
    public String Describe() {
      return String.Format("fancy clam pizza with {0} and
         {1}",ClamType(), cheese);
    }
}
```

2. Registration

```
namespace DITest{
public class IoCInstaller: IWindsorInstaller {
  public void Install(IWindsorContainer container, IConfigurationStore
      store) {
    container.Register(Classes
      .FromThisAssembly()
      .InNamespace("DITest.NYStyle")
      .WithServiceAllInterfaces());
    container.Register(Classes
      .FromThisAssembly()
      .AllowMultipleMatches()
      .InSameNamespaceAs<IoCInstaller>()
      .WithServiceAllInterfaces());
  }
}
}
```

Castle Windsor, `http://www.castleproject.org`

3. Resolution

```
var container = new WindsorContainer();
// adds and configures all components using
   WindsorInstallers from executing assembly
container.Install(FromAssembly.This());

// instantiate and configure root component and all its
   dependencies and their dependencies and...
var p = container.Resolve<ICheesePizza>();
Console.WriteLine(p.Describe());

// clean up, application exits
container.Dispose();
```

Part XI

Singleton

What about static methods?

```
public class Singleton {
  private static Singleton instance = new Singleton();
  private String name;
  public String getName() {
    return name;
  }
  public static void someOtherMethod(){
    System.out.println("Hi there!");
  }
  private Singleton() {
    try {
      // Very expensive job indeed
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    name = Math.random() > 0.5 ? "Jonas" : "Anders";
  }
}
```

Our app takes forever to load if the Singleton class is part of it.

```java
// Thread that does not use the Singleton object
Thread t1 = new Thread(new StaticMethodInvocation());
// Thread that uses the Singleton object
Thread t2 = new Thread(new SingletonLookup());
t0 = System.nanoTime();
t1.start();
t2.start();
try {
  t1.join();
  t2.join();
} catch (InterruptedException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}
```

```
someOtherMethod invoked
Singleton name: Anders
Singleton lookup took 1 003 348 000 ns
Static method invocation took 1 002 463 000 ns
```

How about now?

```
private static Singleton instance;

public static Singleton getInstance() {
  if (instance == null) {
    instance = new Singleton();
  }
  return instance;
}
```

- How about now?

```
someOtherMethod invoked
Singleton name: Anders
Static method invocation took 899 000 ns
Singleton lookup took 1 003 348 000 ns
```

What about threads?

```
private Singleton() {
  try {
    // Very expensive job indeed
    Thread.sleep(100);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  name = Math.random() > 0.5 ? "Jonas" : "Anders";
}

private static final class SingletonLookup implements Runnable {
  @Override
  public void run() {
    System.out.println(MessageFormat.format("Singleton name: {0}
        ",
        Singleton.getInstance().getName()));
  }
}
```

```java
public static void main(String[] args) {
  Thread t1 = new Thread(new SingletonLookup());
  Thread t2 = new Thread(new SingletonLookup());
  t0 = System.nanoTime();
  t1.start(); t2.start();
  try {
    t1.join();
    t2.join();
  } catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
  System.out.println("Singleton name after our threads have run:
      "+Singleton.getInstance().getName());
}
```

```
Singleton name: Jonas
Singleton name after our threads have run: Anders Oops!
```

```java
public static synchronized Singleton getInstance() {
  if (instance == null) {
    instance = new Singleton();
  }
  return instance;
}
```

```
Singleton name: Anders
Singleton name: Anders
Singleton lookup took 1 003 340 000 ns
Singleton lookup took 1 003 286 000 ns
Singleton name after our threads have run: Anders
```

Woohoo!

Singleton: consequences

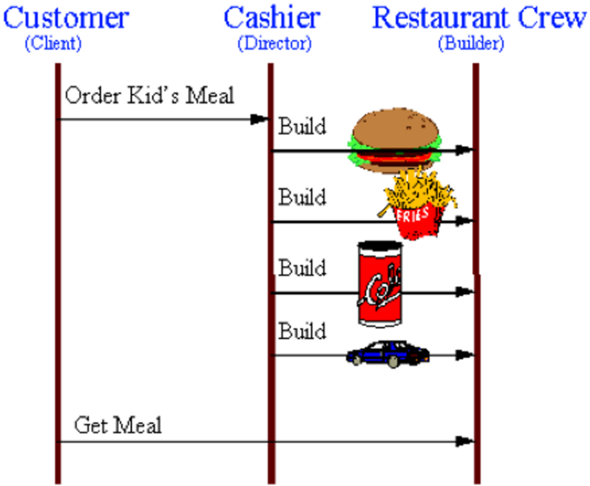
- Violates several design principles!
+ Ensures single objects per class
  - Saves memory
  - Ensures consistency

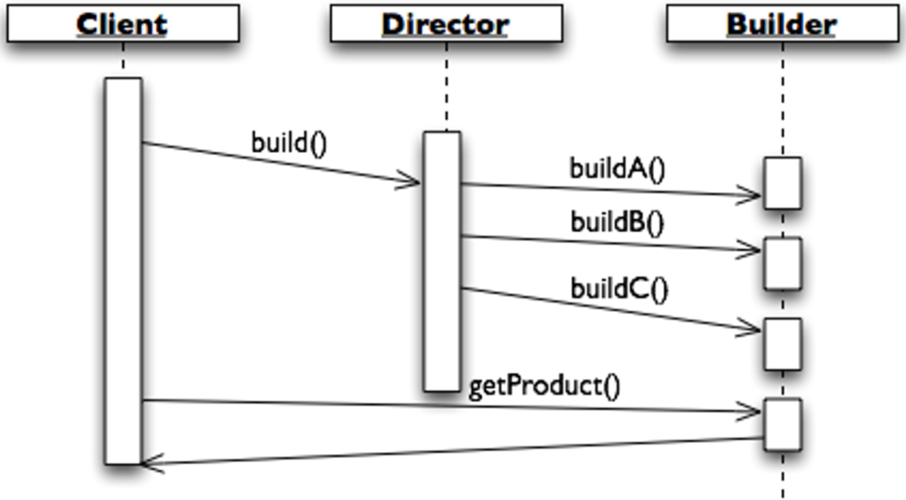
Singleton considered dangerous

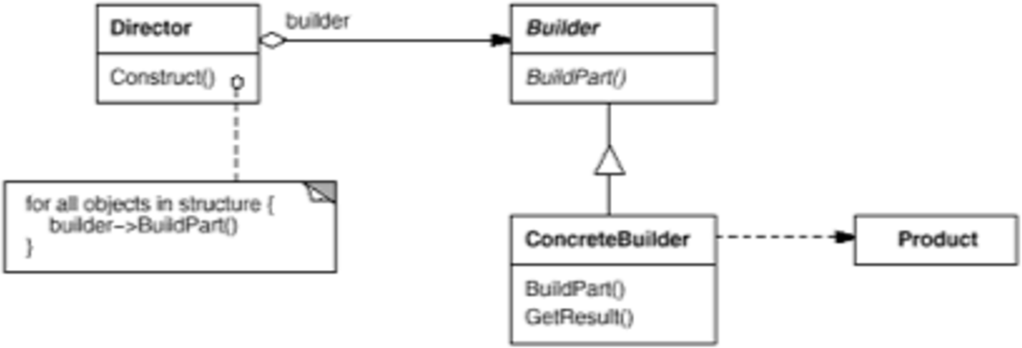
- Encapsulate what varies
- Program to an interface, not to an implementation
- Favor composition over inheritance
- Classes should be open for extension but closed for modification
- Don't call us, we'll call you
- Depend on abstractions, do not depend on concrete classes
- Classes should only have one reason to change
- Strive for loosely-coupled design

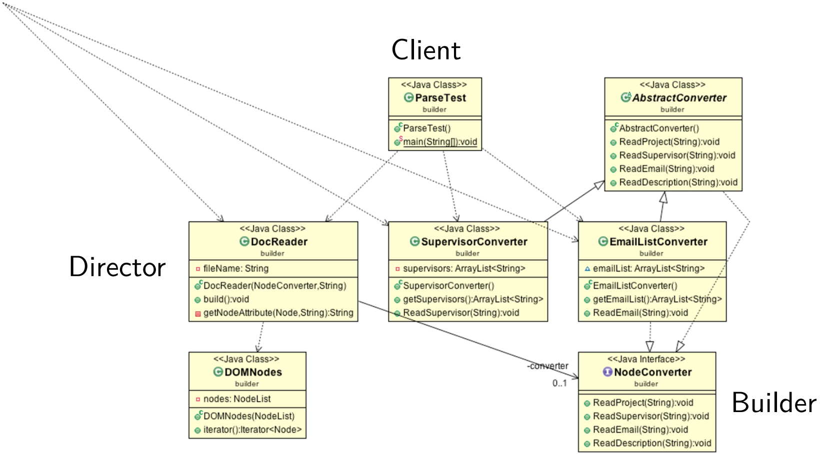
Part XII

Builder


Customer
(Client)
Cashier
(Director)
Restaurant Crew
(Builder)
Order Kid's Meal
Build
Build
FRIES
Build
Build
Get Meal


Client
Director
Builder
build()
buildA()
buildB()
buildC()
getProduct()


Director
Construct()
builder
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product


Client
Director
Builder
<<Java Class>>
ParseTest
builder
ParseTest()
main(String[]):void
<<Java Class>>
AbstractConverter
builder
AbstractConverter()
ReadProject(String):void
ReadSupervisor(String):void
ReadEmail(String):void
ReadDescription(String):void
<<Java Class>>
DocReader
builder
fileName: String
DocReader(NodeConverter,String)
build():void
getNodeAttribute(Node,String):String
<<Java Class>>
SupervisorConverter
builder
supervisors: ArrayList<String>
SupervisorConverter()
getSupervisors():ArrayList<String>
ReadSupervisor(String):void
<<Java Class>>
EmailListConverter
builder
emailList: ArrayList<String>
EmailListConverter()
getEmailList():ArrayList<String>
ReadEmail(String):void
<<Java Class>>
DOMNodes
builder
nodes: NodeList
DOMNodes(NodeList)
iterator():Iterator<Node>
-converter
0..1
<<Java Interface>>
NodeConverter
builder
ReadProject(String):void
ReadSupervisor(String):void
ReadEmail(String):void
ReadDescription(String):void

Abstract Factory

Client receives a Factory
Client requests a product from Factory ⇒ Client receives an abstract product

Builder

Client initializes Director with Builder
Client asks Director to build
Client requests product from Builder ⇒ Client receives a builder-specific product

Builder: consequences

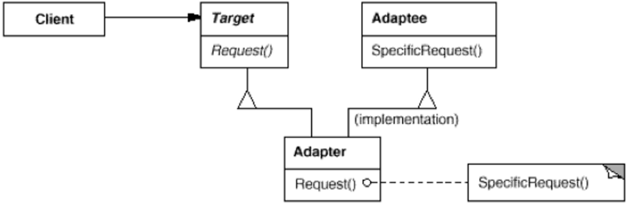
+ Can control the way objects are created
+ Can produce different products using the same Director

- Not necessarily a common interface for products
- Clients must know how to initialize builders and retrieve products

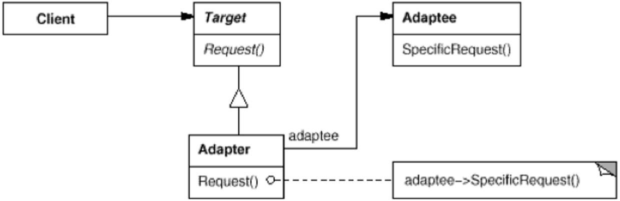
Part XIII

Adapter

Adapter

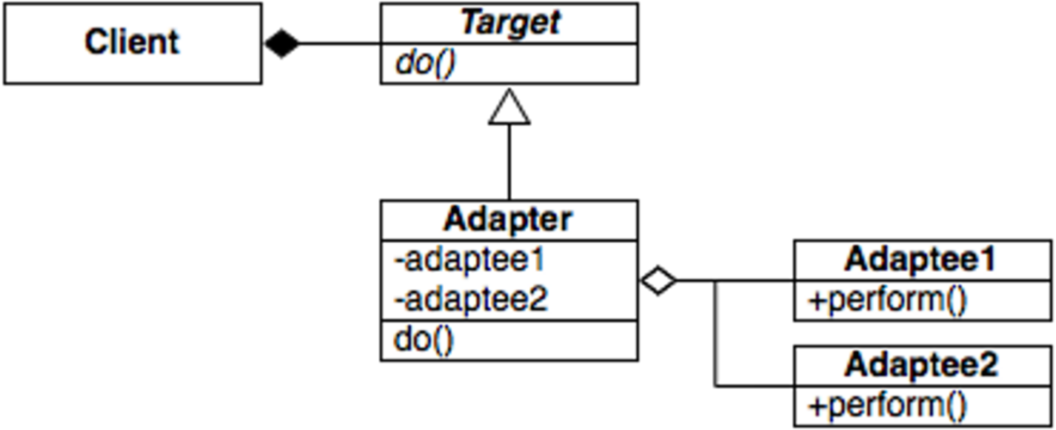


Class Adapter

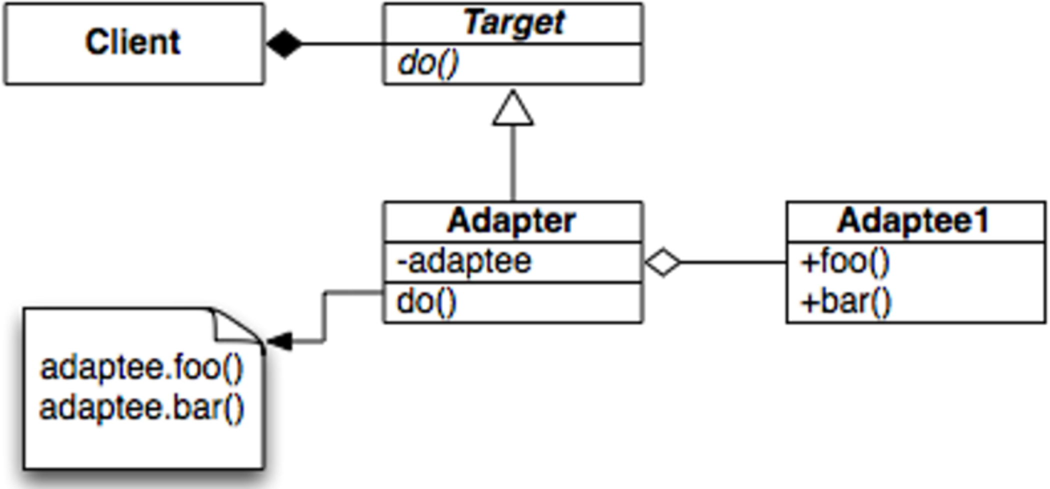


Object Adapter

Multiple back-end objects



Multiple back-end methods



```java
public interface Duck {
  public void quack();
  public void fly();
}

public class TurkeyAdapter implements
    Duck {
  Turkey turkey;

  public TurkeyAdapter(Turkey turkey) {
    this.turkey = turkey;
  }

  public void quack() {
    turkey.gobble();
  }

  public void fly() {
    for(int i=0; i < 5; i++) {
      turkey.fly();
    }
  }
}
```

```java
public interface Turkey {
  public void gobble();
  public void fly();
}

public class DuckAdapter implements
    Turkey {
  Duck duck;
  Random rand;

  public DuckAdapter(Duck duck) {
    this.duck = duck;
    rand = new Random();
  }

  public void gobble() {
    duck.quack();
  }

  public void fly() {
    if (rand.nextInt(5) == 0) {
      duck.fly();
    }
  }
}
```

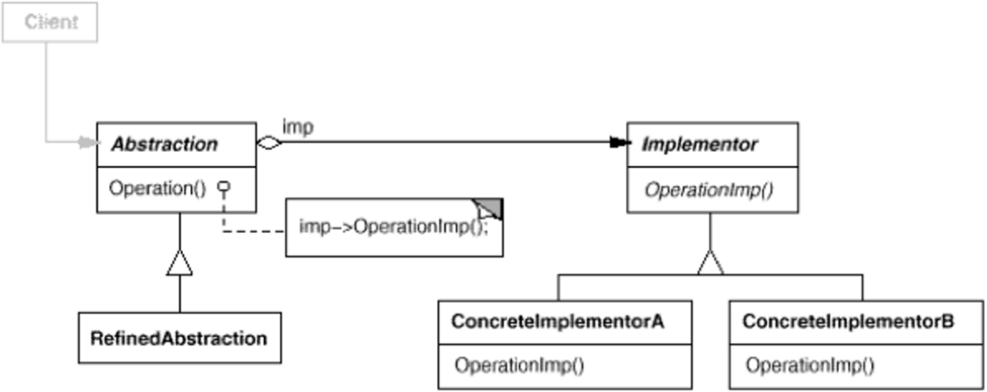
Adapter: consequences

+ Isolates interface changes to the adapter class
- Class adapters require target interfaces or multiple inheritance in the language

Part XIV

Bridge

Abstraction == That which we (should) care about



	Bridge	Strategy
Intent	Decouple two class hierarchies (abstraction/implementation)	Allow for exchangeable algorithms
Collaborations	The Bridge forwards requests to the Implementor	The Context and Strategy collaborate, passing data between them

	Bridge	Adapter
Intent	Decouple two class hierarchies (abstraction/implementation)	Convert an existing class to fit a new interface
Applicability	In a new system	In an existing system

Design principles - Bridge

- Encapsulate what varies
- Program to an interface, not to an implementation
- Favor composition over inheritance
- Classes should be open for extension but closed for modification
- Don't call us, we'll call you
- Depend on abstractions, do not depend on concrete classes
- Classes should only have one reason to change

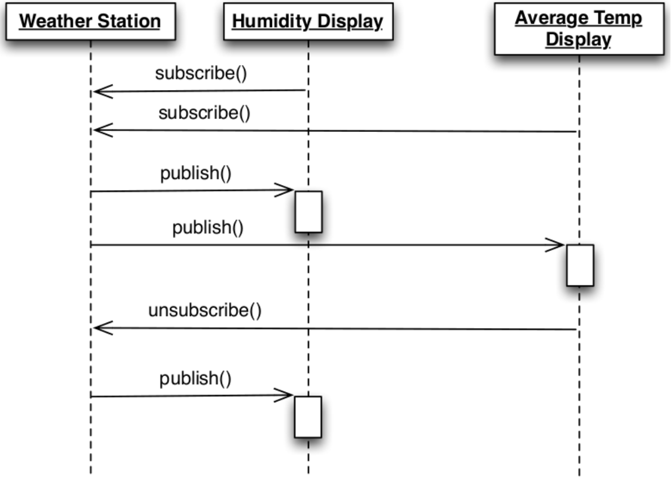
Bridge: consequences

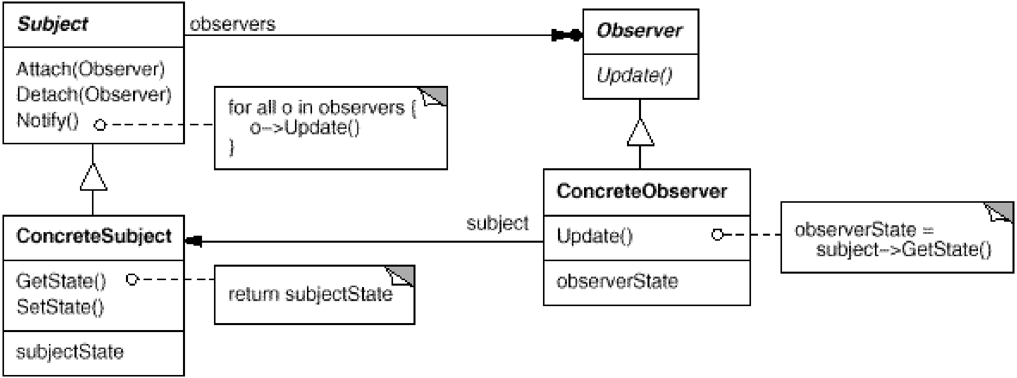
+ Lets two class hierarchies with common superclasses vary independently
- If some implementation classes do not support an abstract concept, the abstraction breaks

Part XV

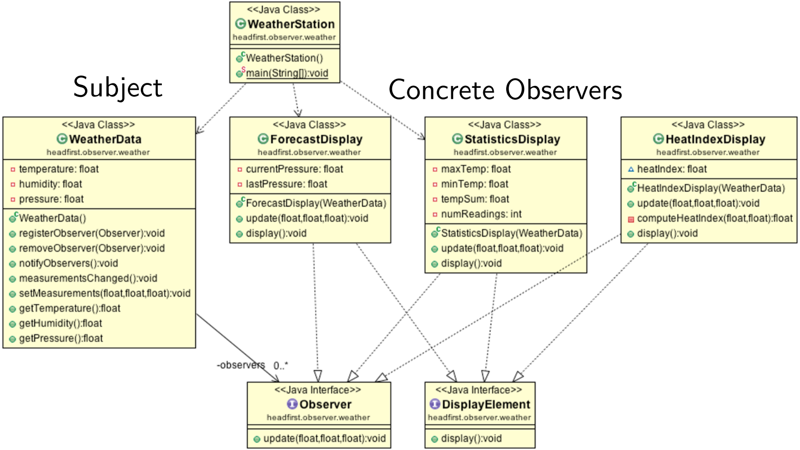
Observer

RSS feeds


Weather Station
Humidity Display
Average Temp Display
subscribe()
subscribe()
publish()
publish()
unsubscribe()
publish()



Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState


Subject
Concrete Observers
<<Java Class>>
WeatherStation
headfirst.observer.weather
WeatherStation()
main(String[]):void
<<Java Class>>
WeatherData
headfirst.observer.weather
temperature: float
humidity: float
pressure: float
WeatherData()
registerObserver(Observer):void
removeObserver(Observer):void
notifyObservers():void
measurementsChanged():void
setMeasurements(float,float,float):void
getTemperature():float
getHumidity():float
getPressure():float
<<Java Class>>
ForecastDisplay
headfirst.observer.weather
currentPressure: float
lastPressure: float
ForecastDisplay(WeatherData)
update(float,float,float):void
display():void
<<Java Class>>
StatisticsDisplay
headfirst.observer.weather
maxTemp: float
minTemp: float
tempSum: float
numReadings: int
StatisticsDisplay(WeatherData)
update(float,float,float):void
display():void
<<Java Class>>
HeatIndexDisplay
headfirst.observer.weather
heatIndex: float
HeatIndexDisplay(WeatherData)
update(float,float,float):void
computeHeatIndex(float,float):float
display():void
-observers 0..*
<<Java Interface>>
Observer
headfirst.observer.weather
update(float,float,float):void
<<Java Interface>>
DisplayElement
headfirst.observer.weather
display():void

Mediator vs Observer

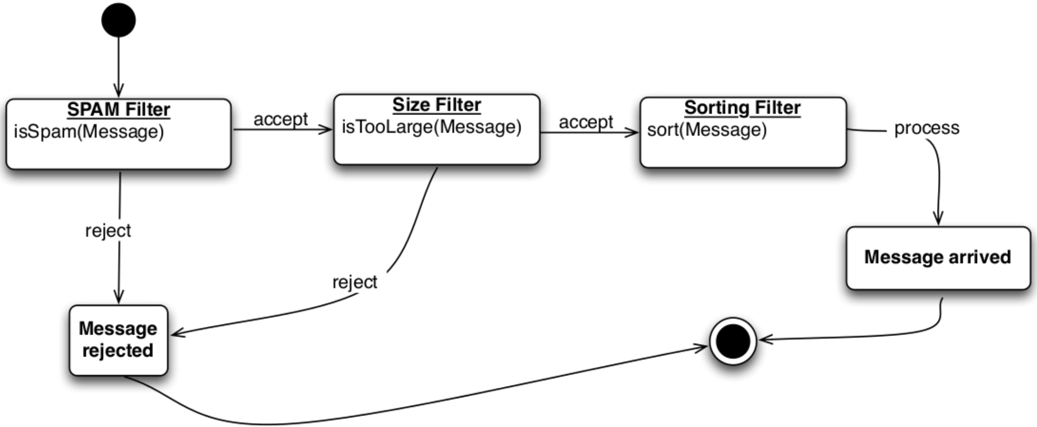
An Observer lets one object (or event) talk to a set of objects.
A Mediator lets objects talk to each other through the Mediator.

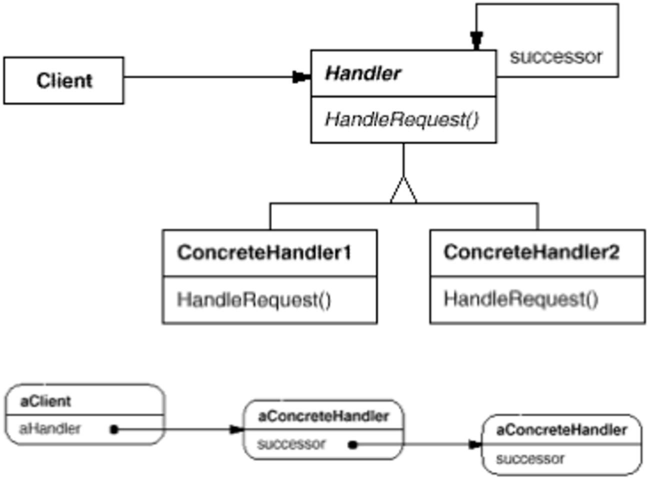
Design principles - Observer

- Encapsulate what varies
- Program to an interface, not to an implementation
- Favor composition over inheritance
- Classes should be open for extension but closed for modification
- Don't call us, we'll call you
- Depend on abstractions, do not depend on concrete classes
- Classes should only have one reason to change
- Strive for loosely-coupled design

Part XVI

Chain of Responsibility


SPAM Filter
isSpam(Message)
accept
Size Filter
isTooLarge(Message)
accept
Sorting Filter
sort(Message)
process
reject
reject
Message arrived
Message rejected


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()
aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

Examples

- Logging
- Input management in GUI:s

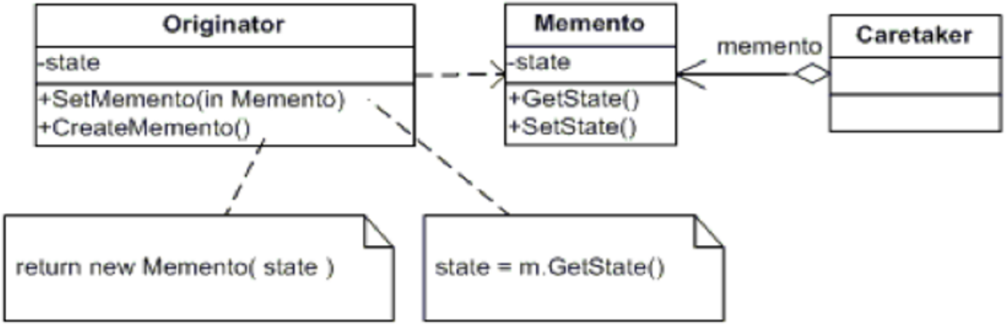
Chain of Responsibility: consequences

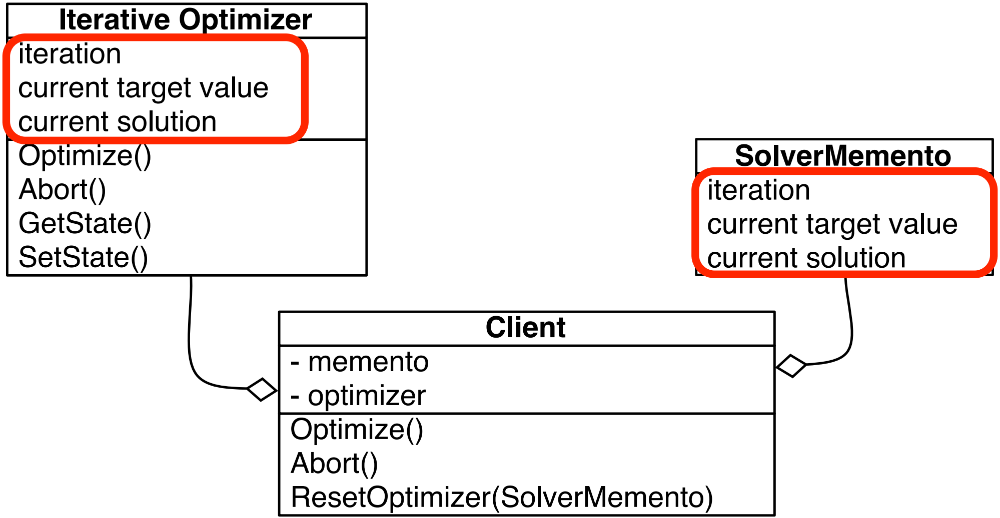
+ Provides the Observer with more control over invocation of targets
- A handler does not know if it will receive a message, depending on the behavior of other handlers in the chain

Part XVII

Memento

Memento


Originator
-state
+SetMemento(in Memento)
+CreateMemento()
Memento
-state
+GetState()
+SetState()
memento
Caretaker
return new Memento(state)
state = m.GetState()


Iterative Optimizer
iteration
current target value
current solution
Optimize()
Abort()
GetState()
SetState()
SolverMemento
iteration
current target value
current solution
Client
- memento
- optimizer
Optimize()
Abort()
ResetOptimizer(SolverMemento)

Mementos in GUIs - Undo/Redo

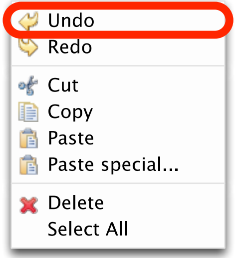
1

User	Name	Age
gi	Ginnie	24

2

User	Name	Age
gi	Ginnie	24
jo	Johnny	37

3



4

User	Name	Age
gi	Ginnie	24

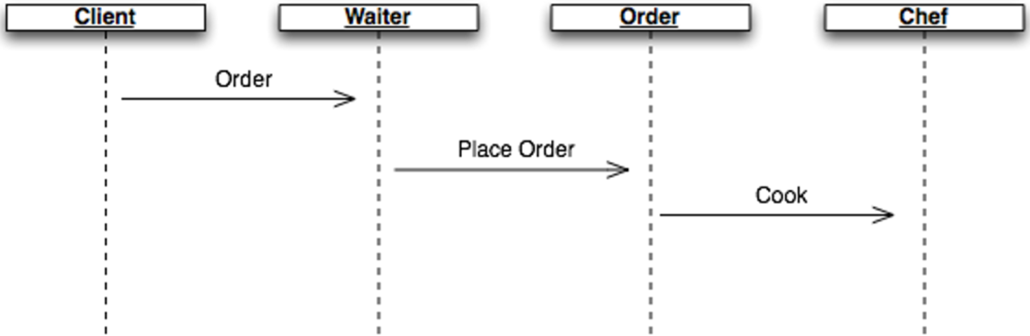
Memento: consequences

- + Can externalize object state for later restoration within the lifetime of the object
- + Encapsulates access to the objects' inner state
- - Depending on implementation, access to private fields requires memento classes as inner/friend classes to each domain class

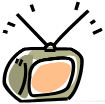
Part XVIII

Command

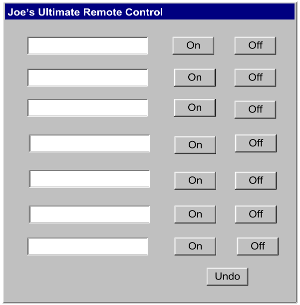
Command



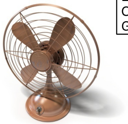
Remote control

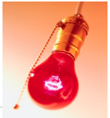


TV
- TurnOn()
- TurnOff()
- SetChannel()

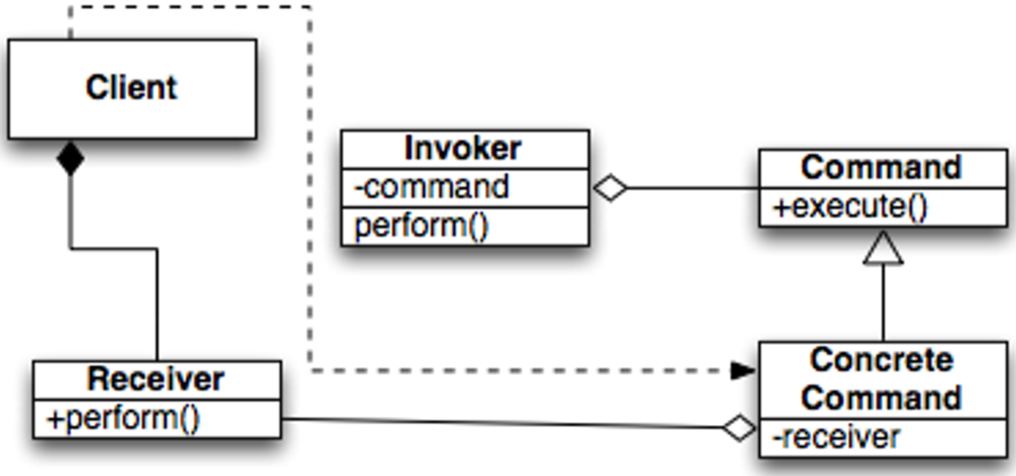


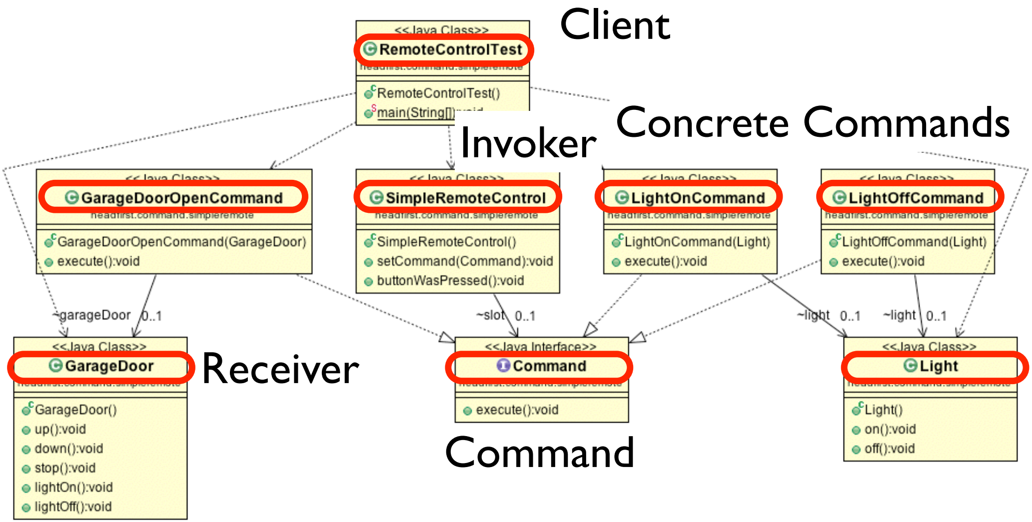
Fan
- High()
- Low()
- Off()
- GetSpeed()





Lamp
- On()
- Off()
- Dim()


Client
Invoker
-command
perform()
Command
+execute()
Receiver
+perform()
Concrete Command
-receiver



Client
<<Java Class>>
RemoteControlTest
RemoteControlTest()
main(String[]):void
Invoker
Concrete Commands
<<Java Class>>
GarageDoorOpenCommand
headfirst.command.simpleremote
GarageDoorOpenCommand(GarageDoor)
execute():void
<<Java Class>>
SimpleRemoteControl
headfirst.command.simpleremote
SimpleRemoteControl()
setCommand(Command):void
buttonWasPressed():void
<<Java Class>>
LightOnCommand
headfirst.command.simpleremote
LightOnCommand(Light)
execute():void
<<Java Class>>
LightOffCommand
headfirst.command.simpleremote
LightOffCommand(Light)
execute():void
~garageDoor 0..1
~slot 0..1
~light 0..1
~light 0..1
<<Java Class>>
GarageDoor
GarageDoor()
up():void
down():void
stop():void
lightOn():void
lightOff():void
Receiver
<<Java Interface>>
Command
execute():void
Command
<<Java Class>>
Light
Light()
on():void
off():void

Command: consequences

- \+ Allows extensions of commands
- \+ Decouples the execution from the specification of the command
- \- Bad design if not needed!
- \- May be confusing if it removes the receiver from responsibilities

Part XIX

Finishing up

Template method, strategy, or factory?

- When is the algorithm chosen?

Compile-time
: Template

Run-time
: Strategy, Factory

Template
: Often has several methods in the class, all implemented by the pattern

Strategy
: Usually only has one method in the class (execute or similar)

Factory
: Is a creational pattern (returns an object)

Coursework

- Intro seminar
- Using design patterns lab (implement design in skeleton) + seminar (finished with the lab and share solution 24h in advance)
- Design principles and Reading design patterns (next week)

References

Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

Reid Smith. "Panel on Design Methodology". In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA '87. Orlando, Florida, USA: ACM, 1987, pp. 91–95. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62151.