Esercizi sui design patterns

Prof. Paolo Ciancarini
Corso di Ingegneria del Software
CdL Informatica
Università di Bologna

Agenda

Questa è una raccolta di esercizi sui design patterns, parte originali parte presi da varie fonti (vedi riferimenti in fondo)

- Pattern GRASP (di Larman)
- Patterns GoF

Riconoscere il problema e applicare il pattern

Pensaci bene!

Sui GRASP

- Quale pattern aiuta ad assegnare una specifica responsabilità ad una classe particolare tra le molte di un dominio?

a) Coesione

b) [✓ Information Expert]

c) Accoppiamento

d) Controller

e) Creator

Sui GRASP

“Se un programma riceve eventi da più fonti esterne, aggiungere una classe che disaccoppia le fonti degli eventi dagli oggetti che gestiscono gli eventi stessi”. Questo è il pattern

a) Coesione

b) Accoppiamento

c) Creator

d) Information Expert

e) [✓ Controller]

Sui GRASP

Quale coppia di pattern è utile per guidare la progettazione di un’applicazione software?

a) Alta coesione e accoppiamento

[✓ b) Alta coesione e basso accoppiamento]

c) Bassa coesione e alto accoppiamento

d) Bassa coesione e accoppiamento

Sui GRASP

Quale pattern dice che “ciascun oggetto dovrebbe assumere meno che può sulla struttura o proprietà dei suoi componenti” ?

a) Coesione

b) Accoppiamento

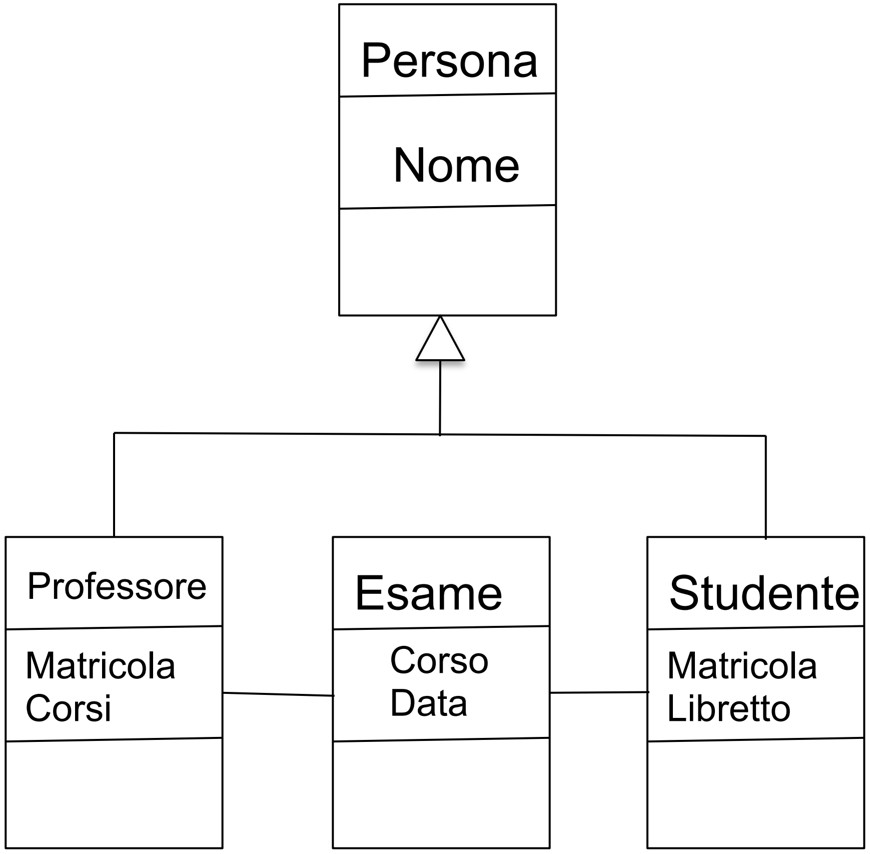
c) Information Expert

[✓ d) Legge di Demetra]

Assegna la responsabilità

Assegnare le responsabilità:

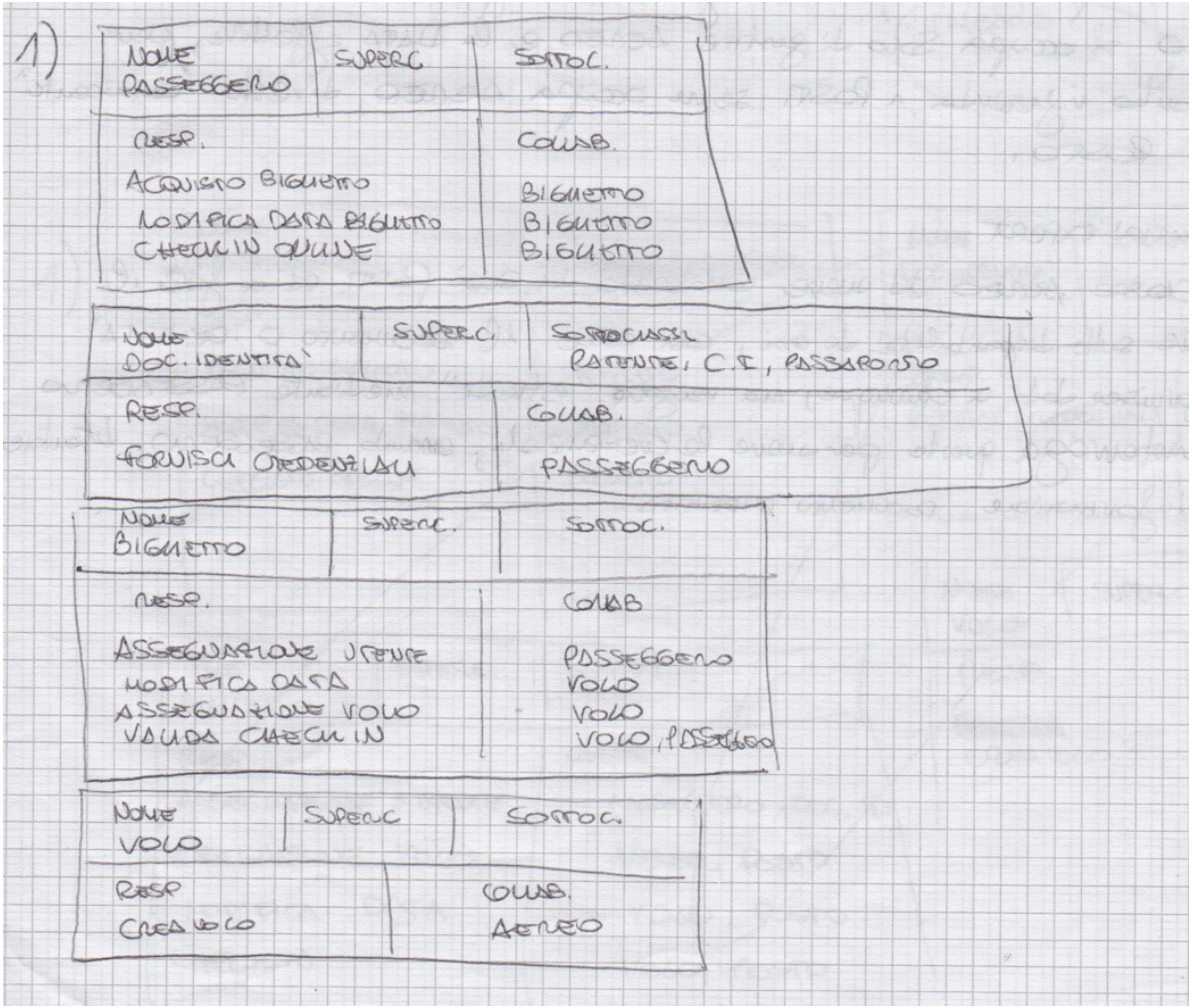
- •Crea_esame
- •Iscrizione_esame
- •Registra_voto
- •Chiedi_matricola



Compito

1. Descrivere un diagramma dei casi d’uso di un sistema di biglietteria su sito Web per viaggi low cost in aereo. Evidenziare almeno le operazioni di i) acquisto biglietto sola andata o andata e ritorno (con o senza registrazione sul sito); ii) modifica data biglietto precedentemente acquistato, a pagamento se manca meno di un mese alla partenza oppure gratis in caso contrario; iii) check-in on line con opzioni a) di scelta del posto a pagamento b) imbarco con priorità (5 punti).
2. Descrivere col metodo CRC l’analisi del sistema di biglietteria descritto nell’esercizio 1, assumendo almeno le classi Passeggero, Documento di Identità, Biglietto, Aereo, Posto. Definire le opportune relazioni ed eventuali classi aggiuntive (4 punti).
3. Descrivere mediante un diagramma delle classi un dominio coerente con l’analisi CRC dell’esercizio precedente (3 punti; 5 punti in più se in questo diagramma mostrate e commentate l’uso di almeno un pattern GRASP).

CRC 1



1)

NOME PASSEGGERO	SUPERC.	SOTTOC.

RESP.	COLLAB.
ACQUISTO BIGLIETTO	BIGLIETTO
MODIFICA DATA BIGLIETTO	BIGLIETTO
CHECKIN ONLINE	BIGLIETTO

NOME DOC. IDENTITA'	SUPERC.	SOTTOCLASSI PATENTE, C.I., PASSAPORTO

RESP.	COLLAB.
FORNISCI CREDENZIALI	PASSEGGERO

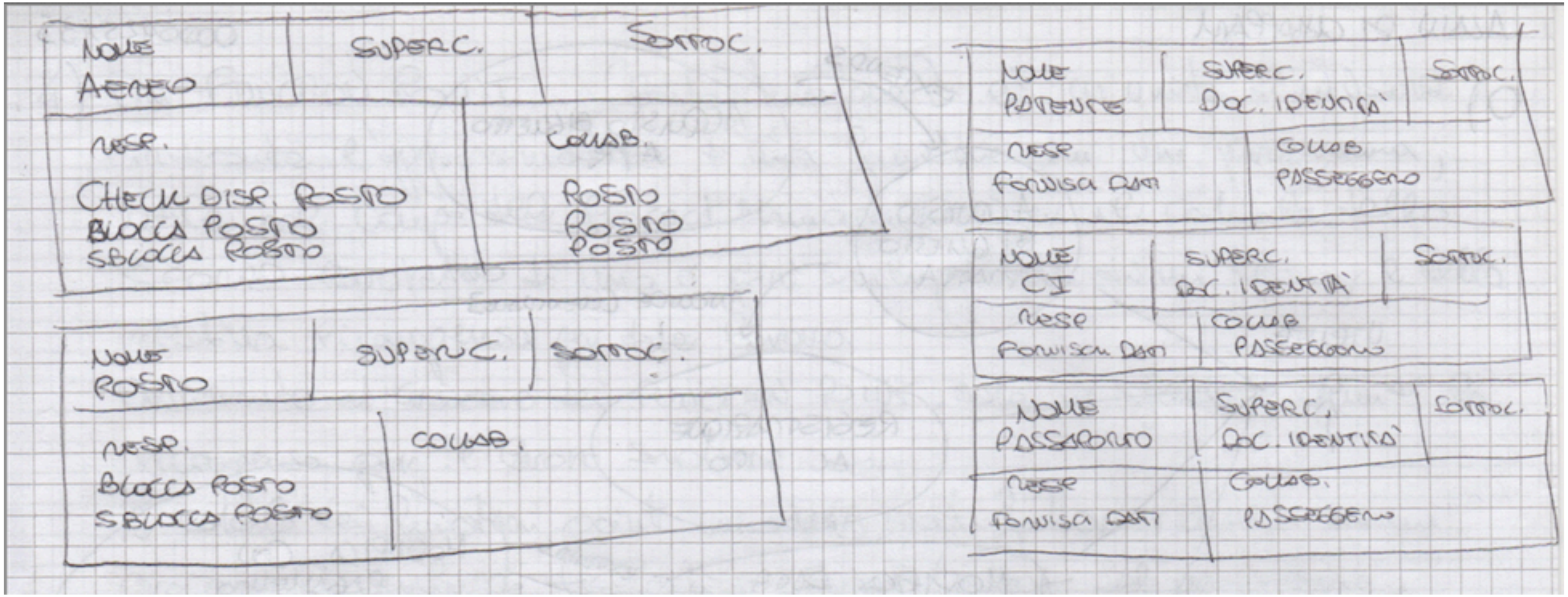
NOME BIGLIETTO	SUPERC.	SOTTOC.

RESP.	COLLAB
ASSEGNAZIONE UTENTE	PASSEGGERO
MODIFICA DATA	VOLO
ASSEGNAZIONE VOLO	VOLO
VALIDA CHECKIN	VOLO, PASSEGGERO

NOME VOLO	SUPERC	SOTTOC.

RESP	COLLAB.
CREA VOLO	AEREO

CRC 2



NOME AEREO	SUPERC.	SOTTOC.
RESP. CHECK DISP. POSTO BLOCCA POSTO SBLOCCA POSTO	COLLAB. POSTO POSTO POSTO	

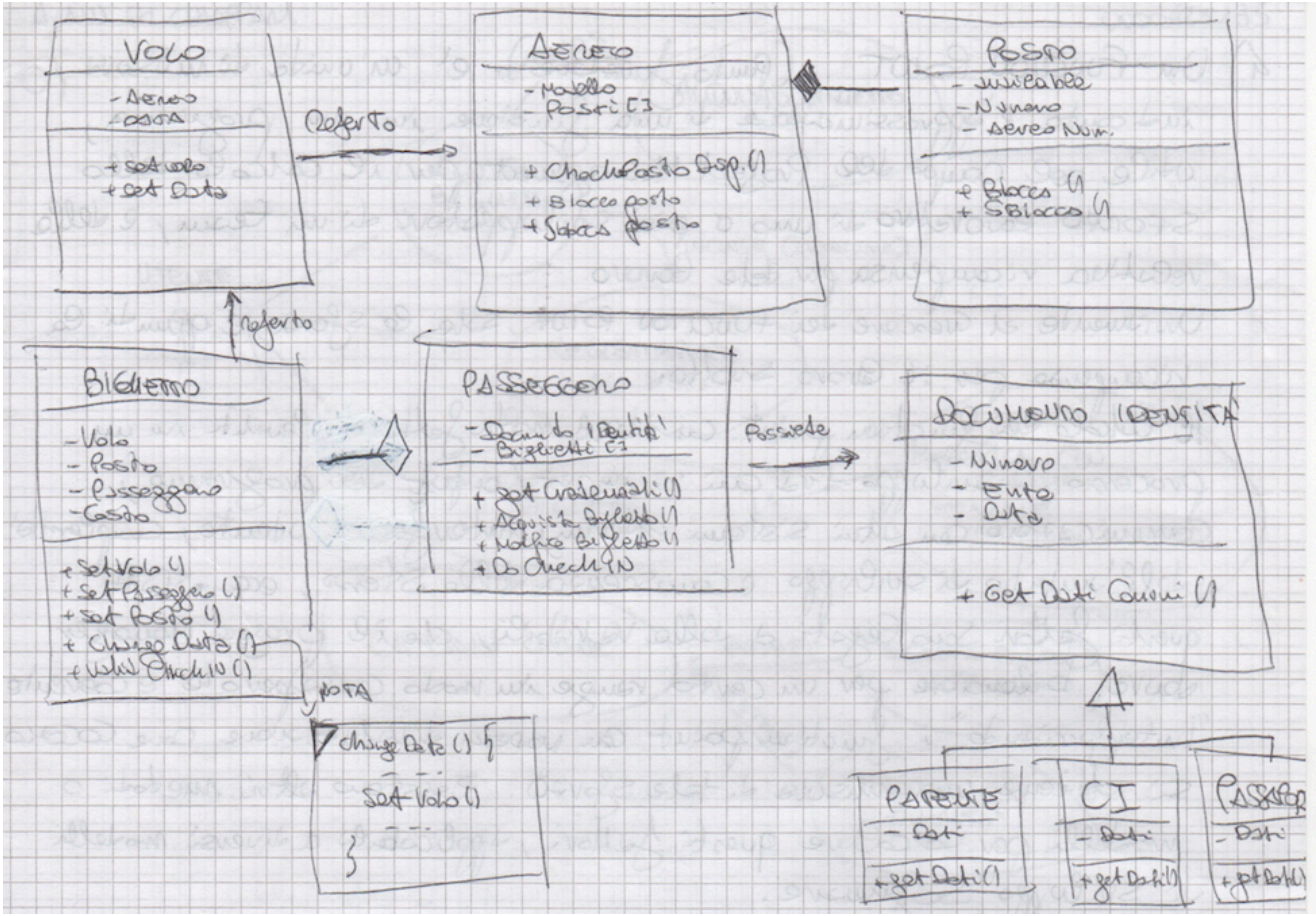
NOME POSTO	SUPERC.	SOTTOC.
RESP. BLOCCA POSTO SBLOCCA POSTO	COLLAB.	

NOME PATENTE	SUPERC. DOC. IDENTITÀ	SOTTOC.
RESP. FORNISCI DATI	COLLAB PASSEGGERO	

NOME CI	SUPERC. DOC. IDENTITÀ	SOTTOC.
RESP FORNISCI DATI	COLLAB PASSEGGERO	

NOME PASSAPORTO	SUPERC. DOC. IDENTITÀ	SOTTOC.
RESP FORNISCI DATI	COLLAB. PASSEGGERO	

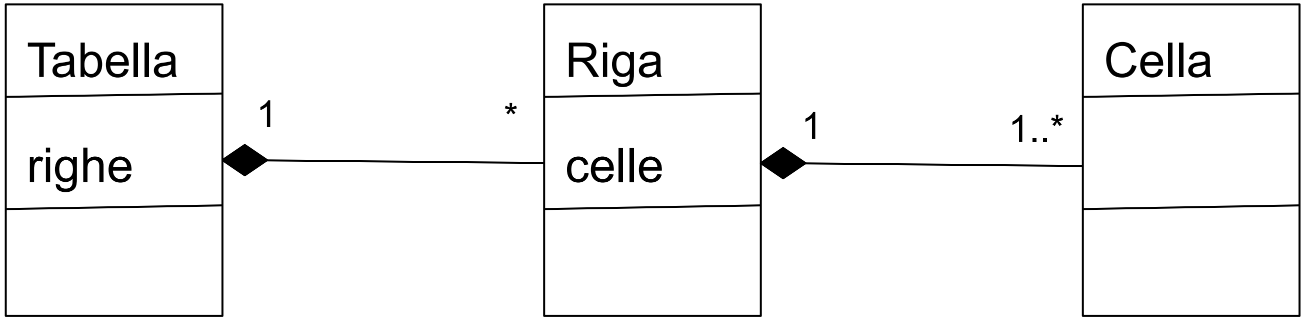
Diagramma di classe



Assegna la responsabilità

Data la seguente rappresentazione UML di una tabella (es. HTML), a chi assegnare la responsabilità di creare una riga? E una cella? Quale pattern si applica?

Come cambierebbero le responsabilità se la Tabella fosse composta solo da celle, non organizzate in righe (cioè se la classe Riga non esiste)?



Assegna la responsabilità

Si consideri il seguente dominio:

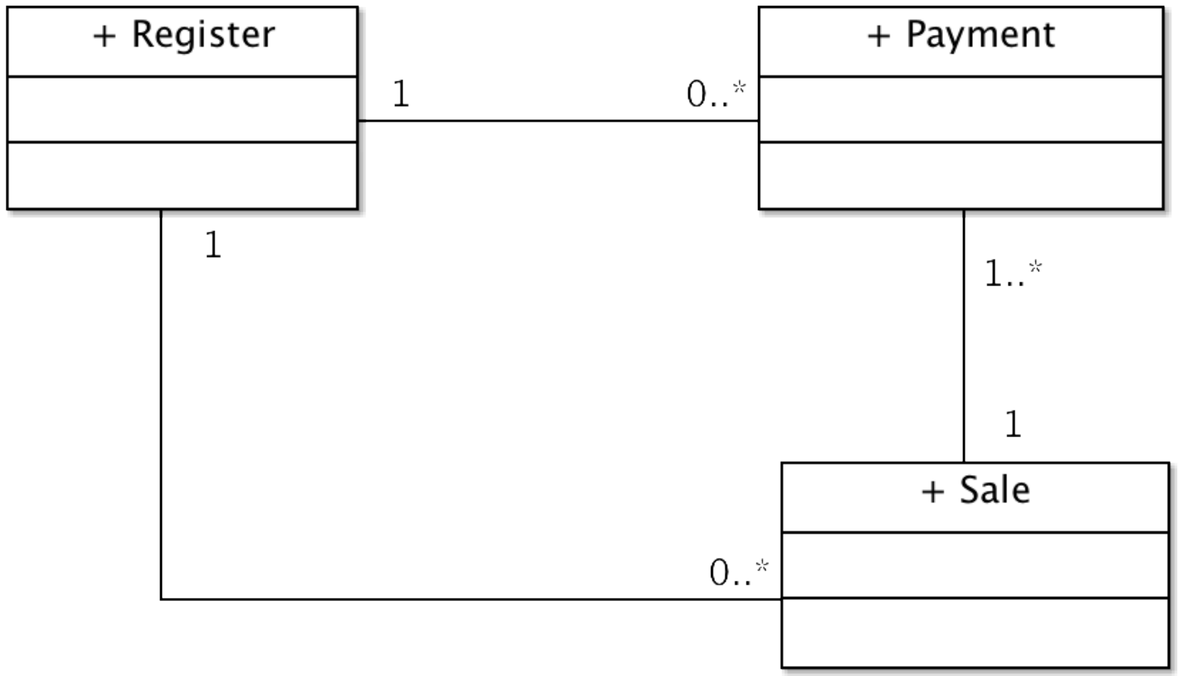
- Un Registro (Register) tiene traccia dei Pagamenti (Payment)
- Ogni Vendita (Sale) è associata a un insieme di Pagamenti

Modellare il dominio con un diagramma delle classi

A chi assegnare la responsabilità di creare un’istanza di Pagamento?

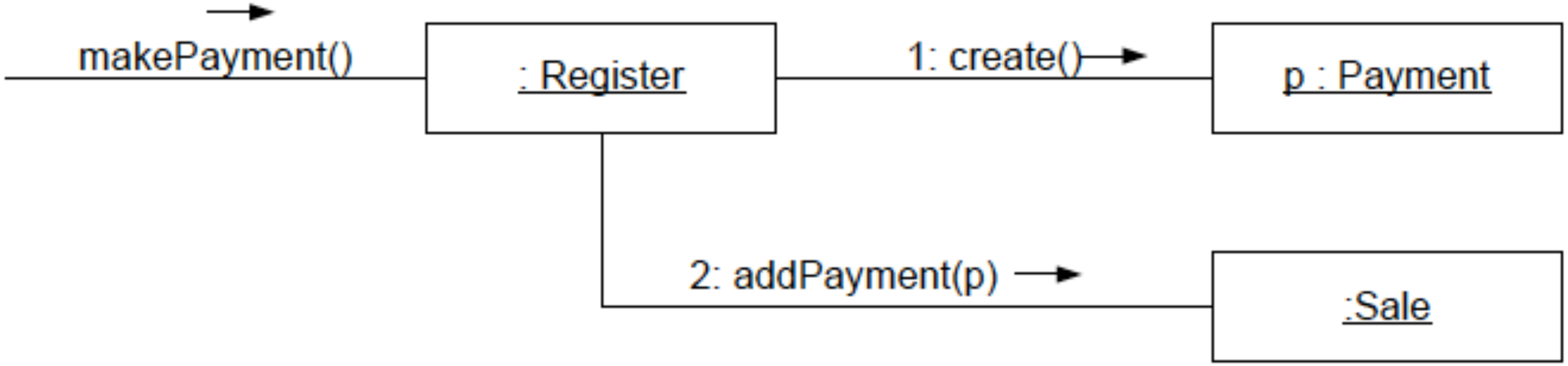
Disegnare un diagramma di comunicazione che descrive la soluzione

Diagramma delle classi



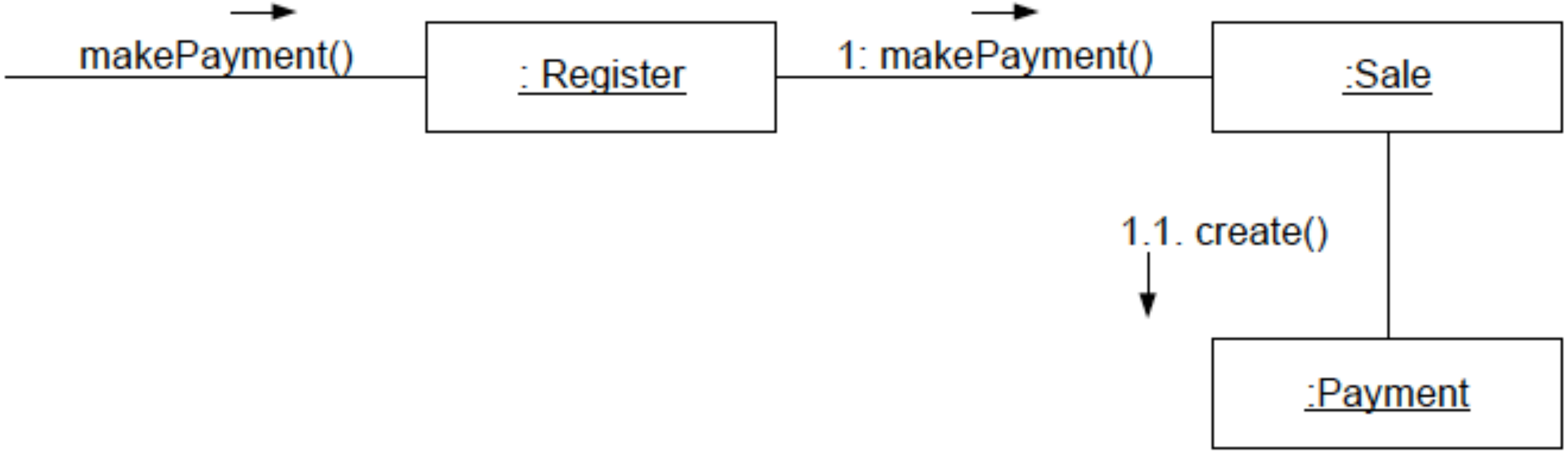
Soluzione a

- Problema?



Soluzione b

- Contraddice il pattern Creator?



Assegna la responsabilità

- Una parete, che contiene porte e finestre, deve essere dipinta con una vernice. Ogni barattolo contiene una data quantità di vernice, che permette di dipingere una data superficie.

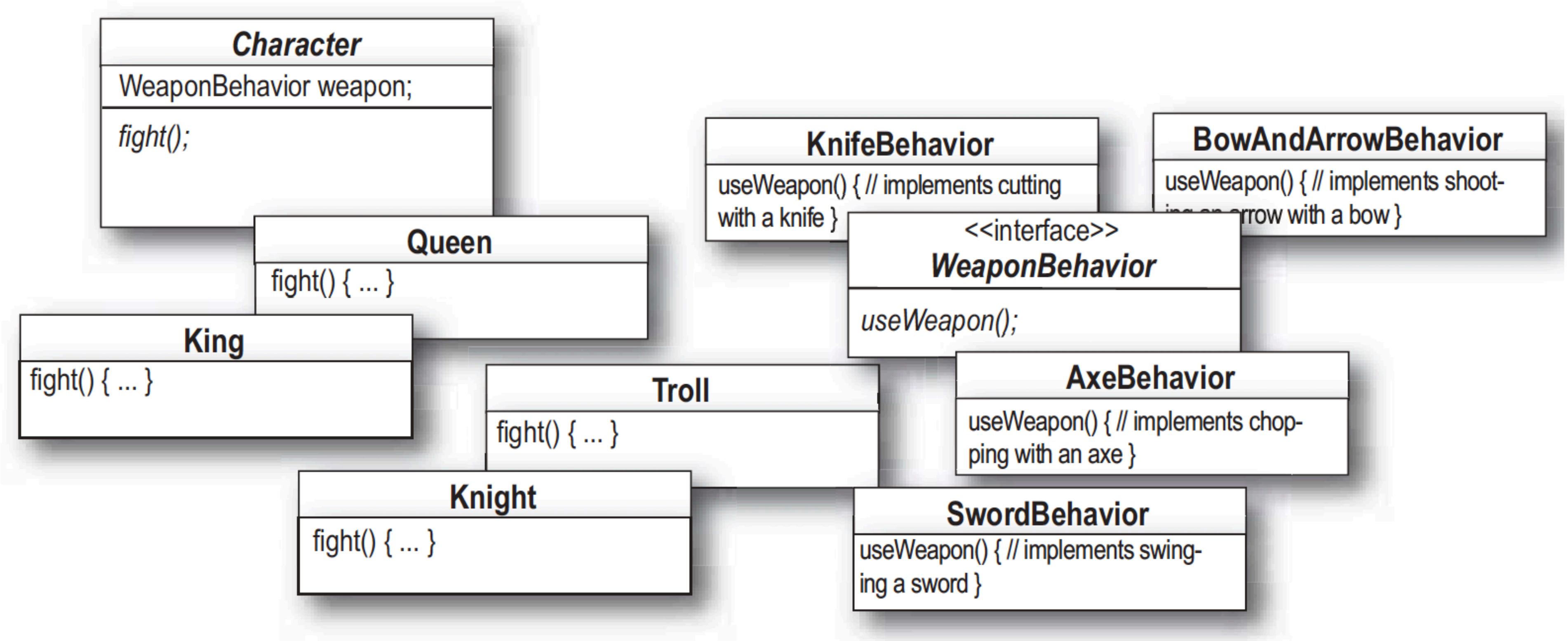
Modellare il problema

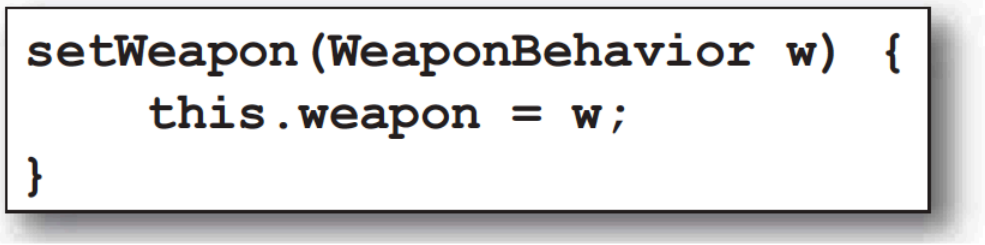
Rispondere alle domande:

- A chi assegnare la responsabilità di calcolare la quantità di vernice necessaria per una data superficie?
- A chi assegnare la responsabilità di calcolare la quantità di vernice necessaria per dipingere una parete?

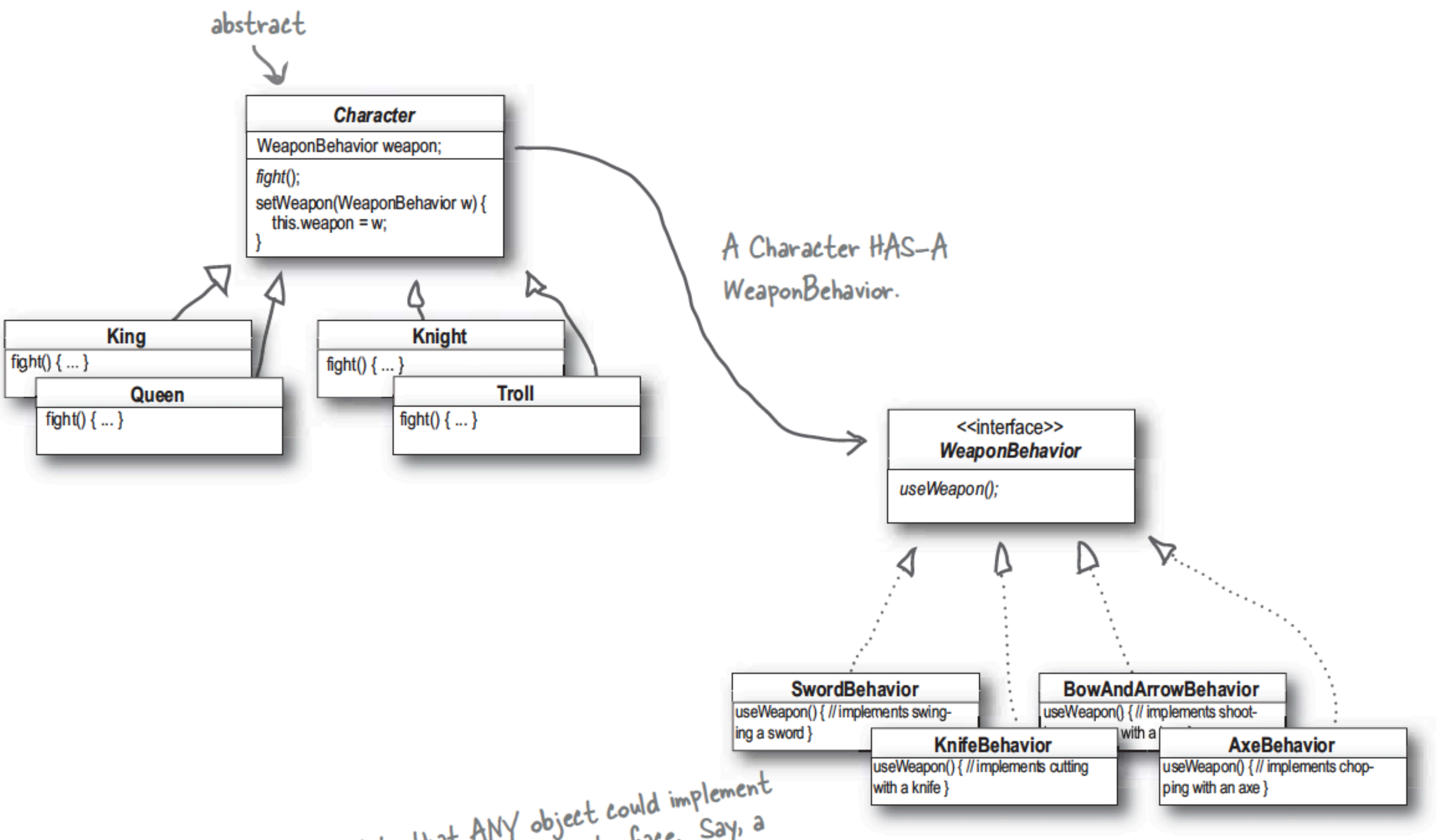
Attribuire una responsabilità

- Disegnare le frecce tra le classi
- Attribuire la responsabilità setWeapon()





```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

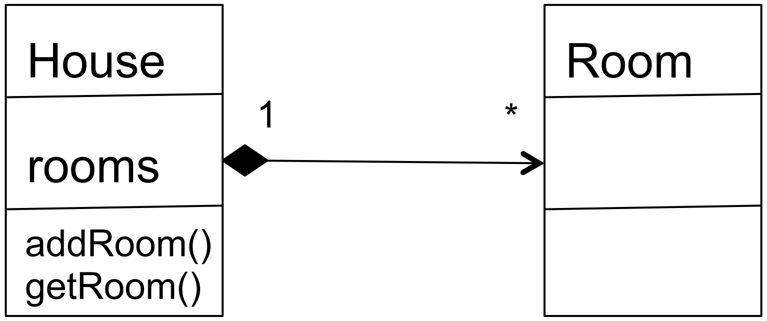


abstract

A Character HAS-A WeaponBehavior.

Note that ANY object could implement the WeaponBehavior interface. Say, a paperclip, a tube of toothpaste or a mutated sea bass.

Quale preferire?


House

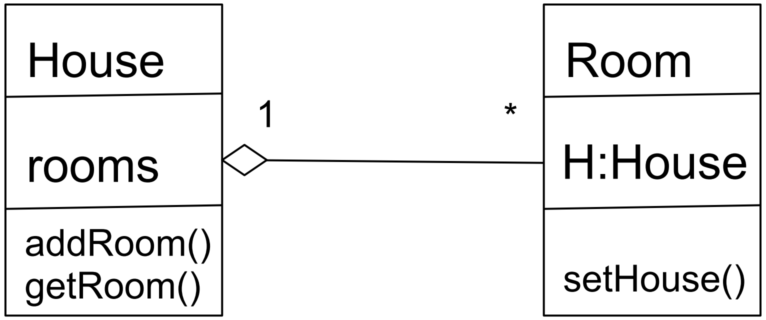
rooms

addRoom()
getRoom()

1

*

Room


House

rooms

addRoom()
getRoom()

1

*

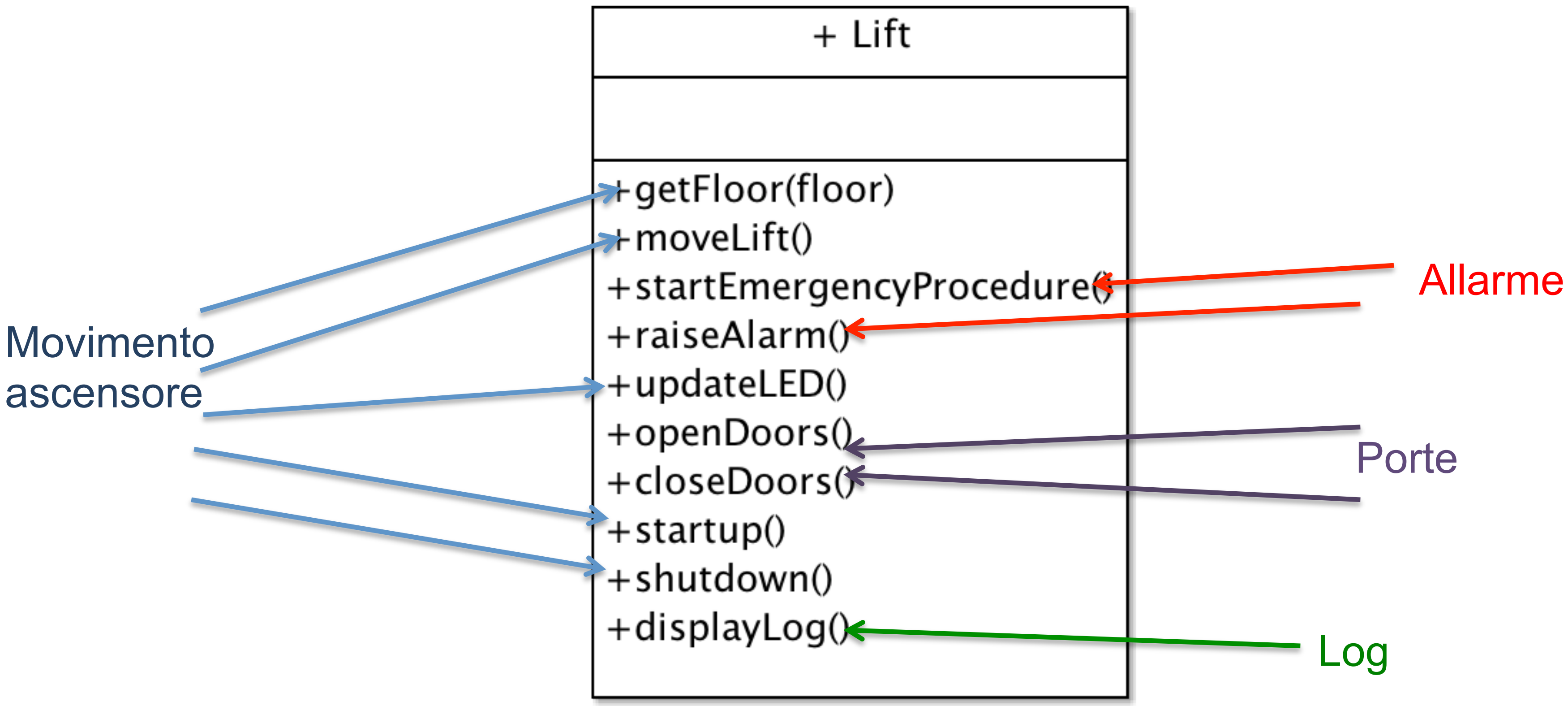
Room

H:House

setHouse()

Esercizio

- Si può progettare meglio questa classe, che modella un ascensore?



+ Lift

+getFloor(floor)
+moveLift()
+startEmergencyProcedure()
+raiseAlarm()
+updateLED()
+openDoors()
+closeDoors()
+startup()
+shutdown()
+displayLog()

Movimento ascensore

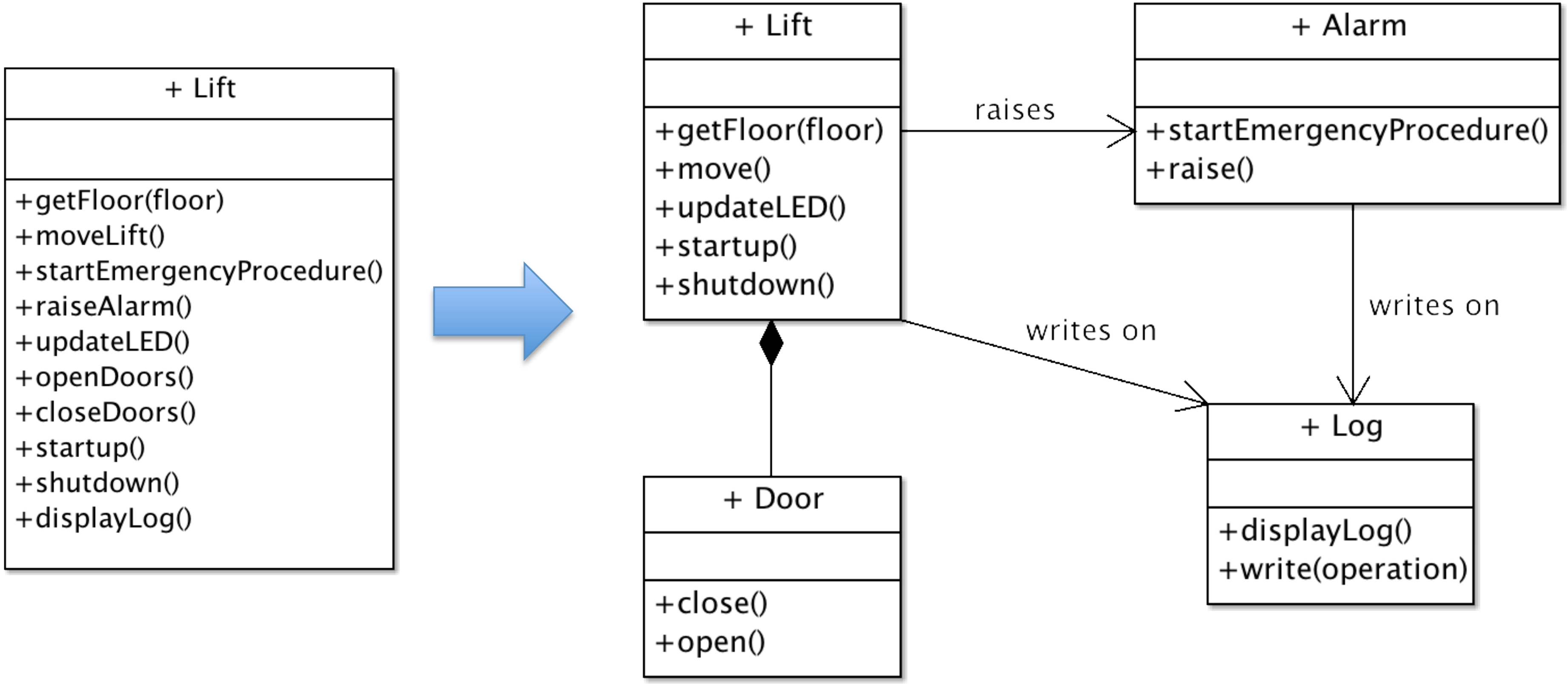
Allarme

Porte

Log

Possibile soluzione

- Coesione?
- Ulteriore refactoring?



+ Lift

+getFloor(floor)
+moveLift()
+startEmergencyProcedure()
+raiseAlarm()
+updateLED()
+openDoors()
+closeDoors()
+startup()
+shutdown()
+displayLog()

+ Lift

+getFloor(floor)
+move()
+updateLED()
+startup()
+shutdown()

raises

+ Alarm

+startEmergencyProcedure()
+raise()

writes on

writes on

+ Door

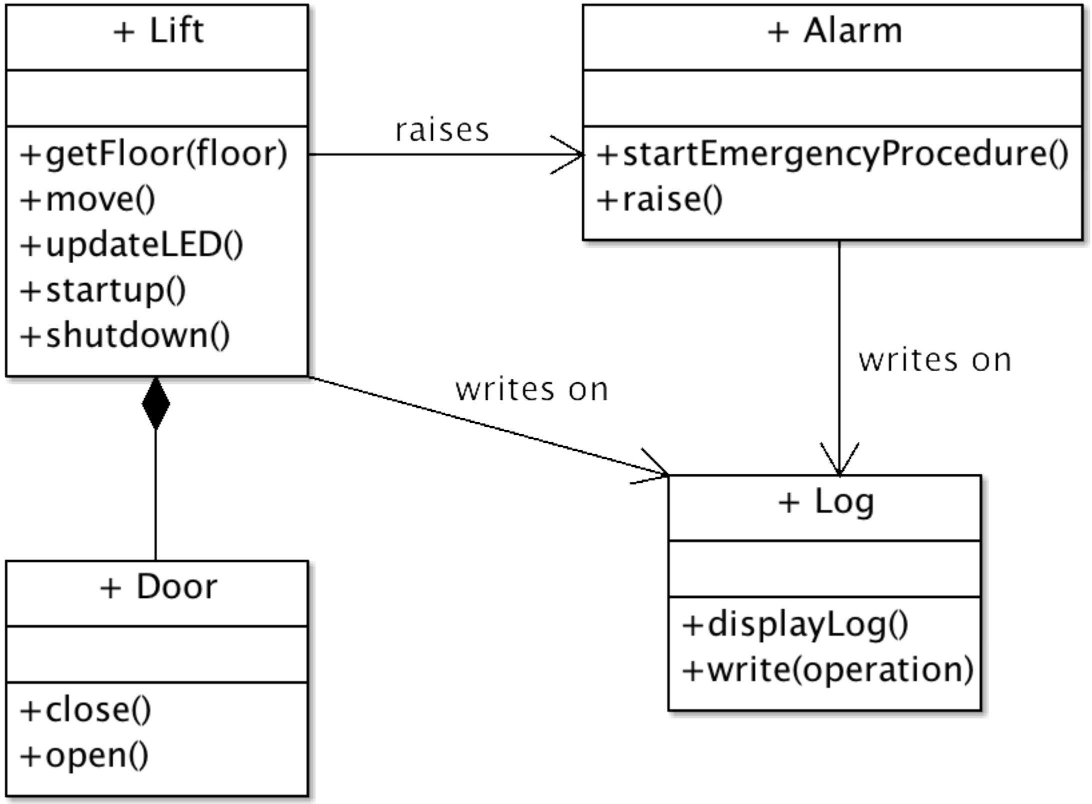
+close()
+open()

+ Log

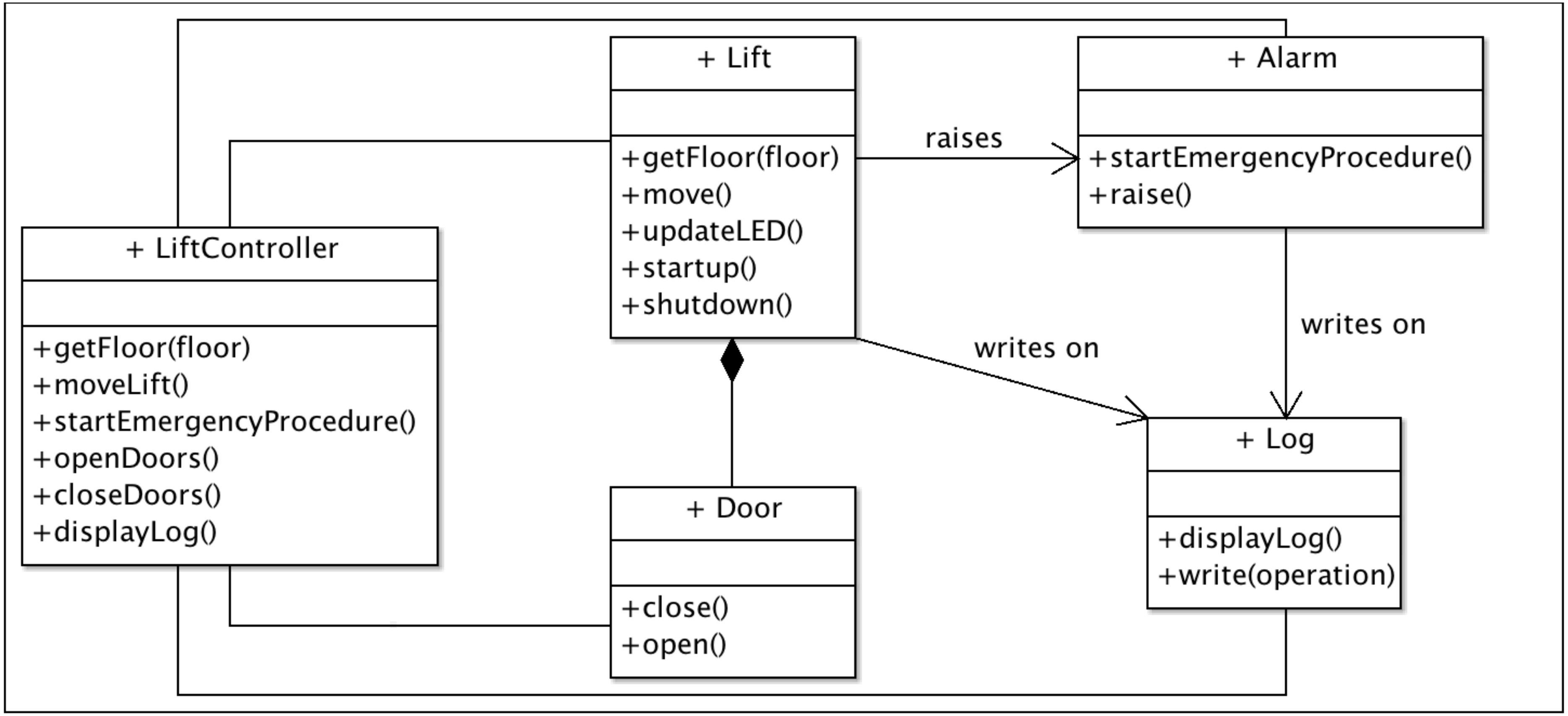
+displayLog()
+write(operation)

Interazione con il sistema

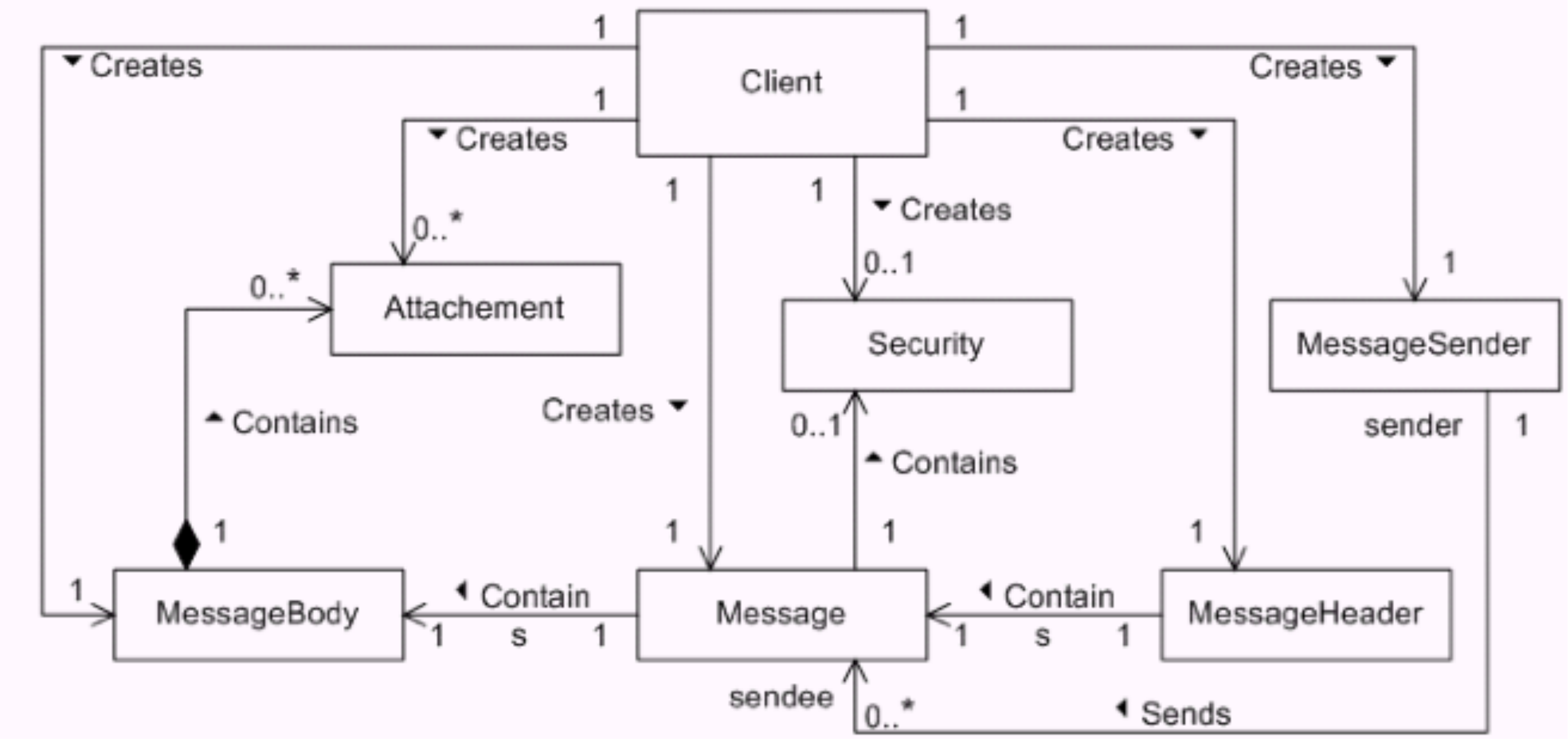
- Quali metodi rispondono ad eventi sollevati dall'utente?



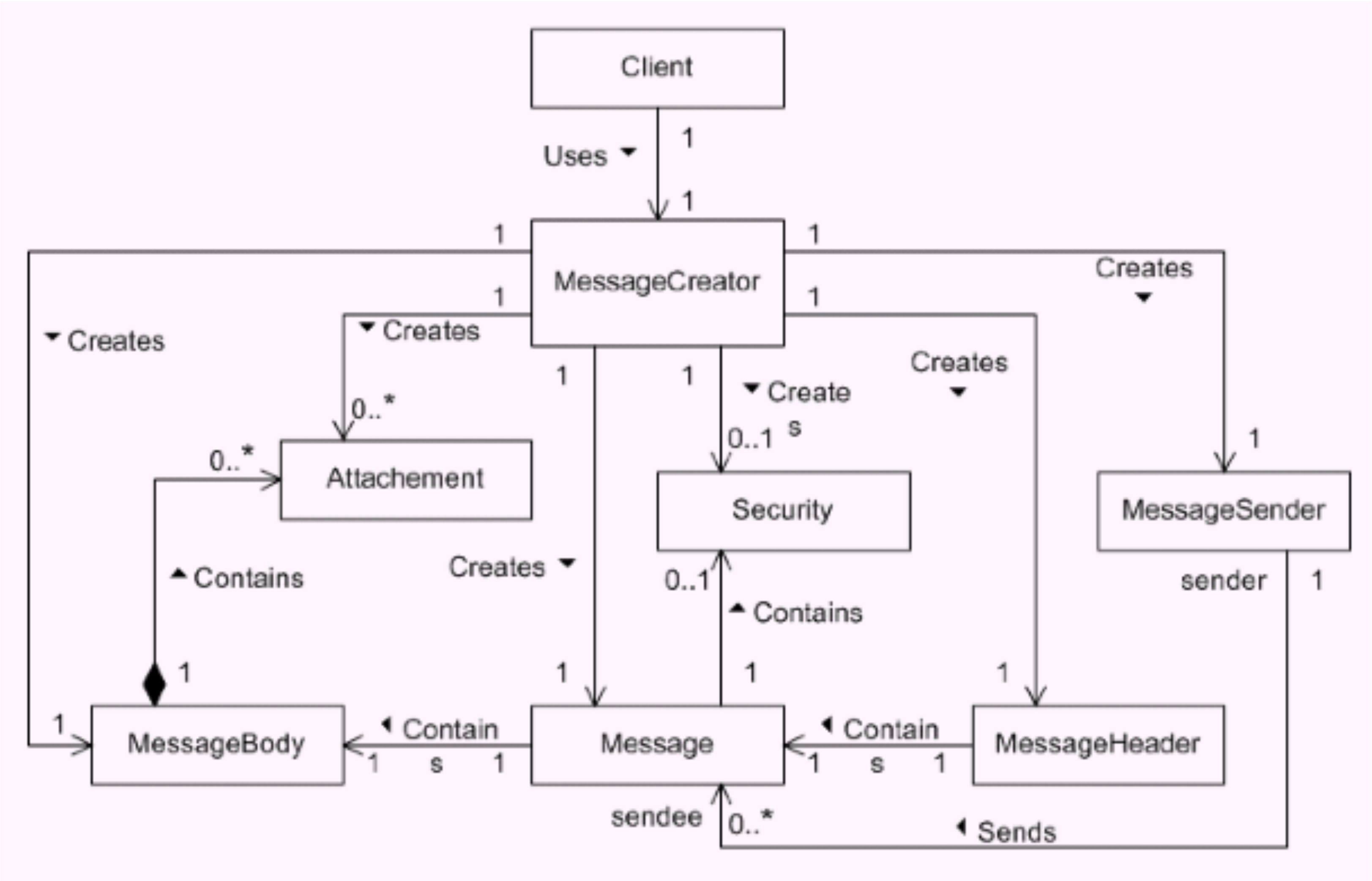
Controller per Ascensore



Controller?


Client
Attachement
Security
MessageSender
MessageBody
Message
MessageHeader
Creates
Creates
Creates
Creates
Creates
Creates
Contains
Contains
Contain s
Contain s
Sends
sender
sendee
1
1
1
1
1
1
0..*
0..*
0..1
0..1
0..*
1

GRASP Controller (GoF Façade)



Pattern GoF creazionali

Nascondono i costruttori delle classi e mettono dei metodi al loro posto creando un'interfaccia: in questo modo si possono utilizzare oggetti senza sapere come sono implementati

- Factory method
- Abstract factory
- Builder
- Prototype
- Singleton

Sul pattern Singleton

Quali frasi sono vere per il pattern Singleton?

1.Permette solamente una istanza di una classe

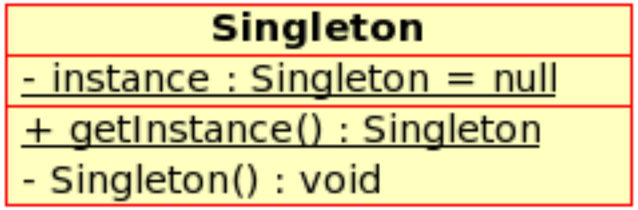
2.Il sistema che include un’istanza Singleton usa un singolo punto di accesso per l’istanza

[✓ 3.Entrambi 1 e 2]

4.Nessuna delle precedenti

Come garantire che un oggetto sia unico in un sistema OO?

```
Public class Singleton{
    Private static Singleton uniqueInstance;
    //other useful instance variables here

    Private Singleton() {}

    Public static Singleton getInstance() {
        if (uniqueInstance == null{
            uniqueInstance= new Singleton();
    }
    Return uniqueInstance,
    }
    //other useful methods here
    }
```



Singleton

- instance : Singleton = null

+ getInstance() : Singleton

- Singleton() : void

Sui pattern GoF

Quale pattern aiuta ad assegnare la responsabilità di creare oggetti la cui logica di creazione è complessa, e inoltre esistono diverse rappresentazioni per oggetti in costruzione?

a. Factory method

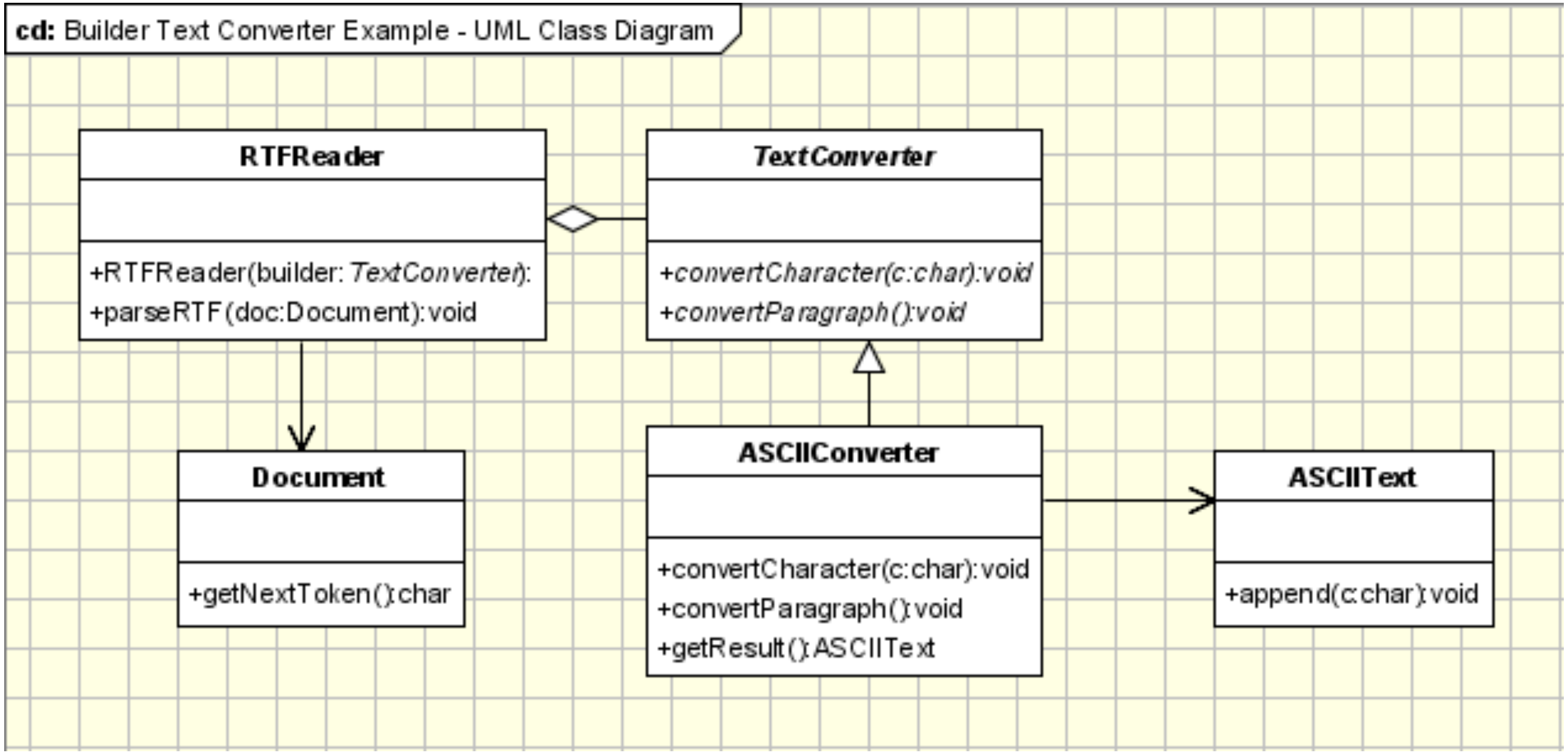
b. Abstract factory

[✓ c. Builder]

d. Prototype

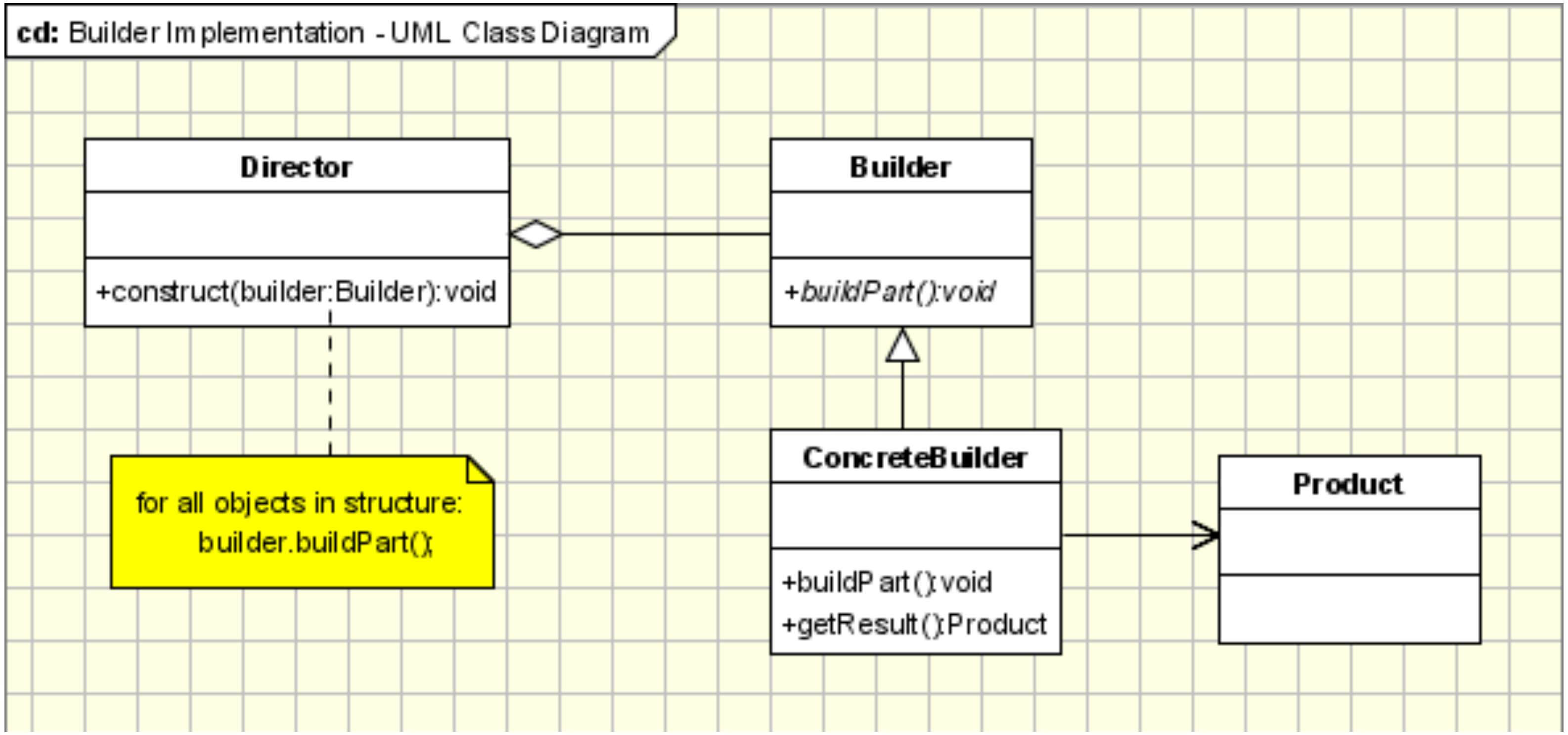
e. Singleton

Identifica il pattern


cd: Builder Text Converter Example - UML Class Diagram
RTFReader
+RTFReader(builder: TextConverter):
+parseRTF(doc:Document):void
TextConverter
+convertCharacter(c:char):void
+convertParagraph():void
Document
+getNextToken():char
ASCIIConverter
+convertCharacter(c:char):void
+convertParagraph():void
+getResult():ASCIIText
ASCIIText
+append(c:char):void

The Client needs to convert a document from RTF format to ASCII format. Therefore, it calls a method createASCIIText that takes as a parameter the document that will be converted. This method calls the ConcreteBuilder, ASCIIConverter, that extends the Builder, TextConverter, and overrides its two methods for converting characters and paragraphs, and also the Director, RTFReader, that parses the document and calls the builder’s methods depending on the type of token encountered. The product, the ASCIIText, is built step by step, by appending converted characters.

Pattern Builder



The client, that may be either another object or the actual client that calls the main() method of the application, initiates the Builder and Director classes. The Builder represents the complex object that needs to be built in terms of simpler objects and types. The constructor in the Director class receives a Builder object as a parameter from the Client and is responsible for calling the appropriate methods of the Builder class. In order to provide the Client with an interface for all concrete Builders, the Builder class should be an abstract one. This way you can add new types of complex objects by only defining the structure and reusing the logic for the actual construction process. The Client is the only one that needs to know about the new types, the Director needing to know which methods of the Builder to call.

Builder vs Abstract Factory

- The Builder design pattern is similar to the Abstract Factory pattern
- In the case of the Abstract Factory, the client uses the factory’s methods to create its own objects
- In the Builder’s case, the Builder class is instructed on how to create the object and then it is asked for it, but the way that the class is put together is up to the Builder class

Pattern GoF strutturali

I pattern strutturali consentono di riutilizzare un oggetto esistente fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze

- Adapter (class, object)
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Sui pattern GoF

Quale pattern GoF definisce una struttura ricorsiva?

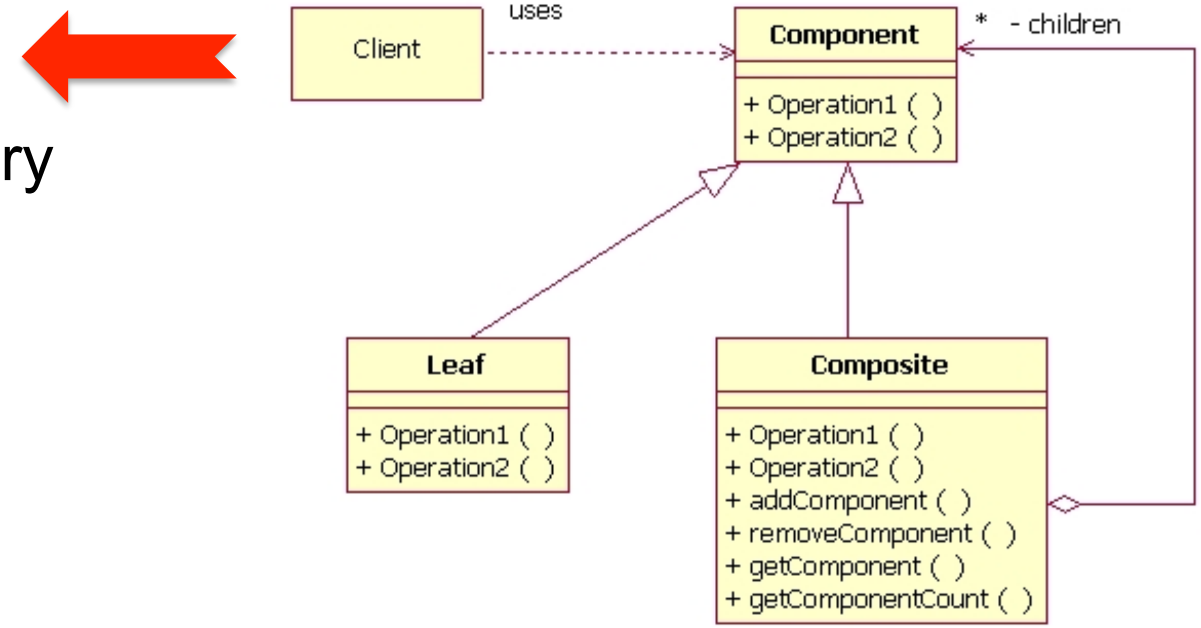
a) Bridge

b) Composite

c) Abstract factory

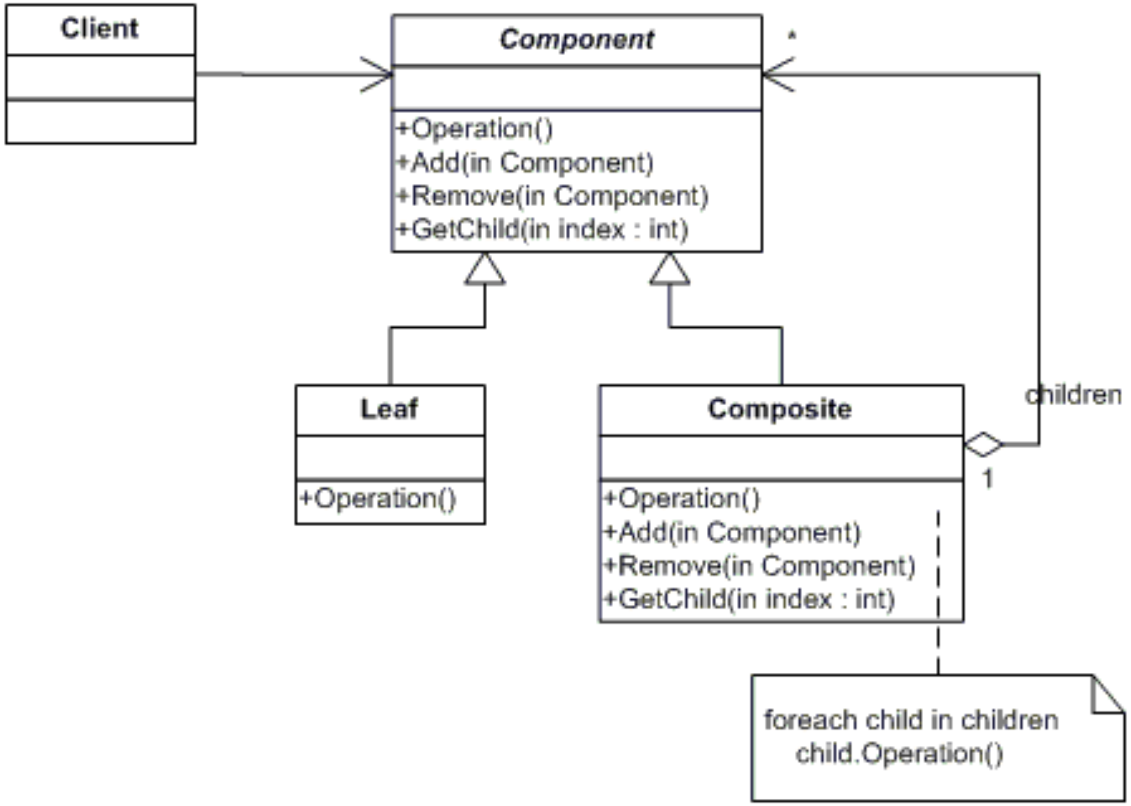
d) Strategy

e) Decorator



Composite Pattern

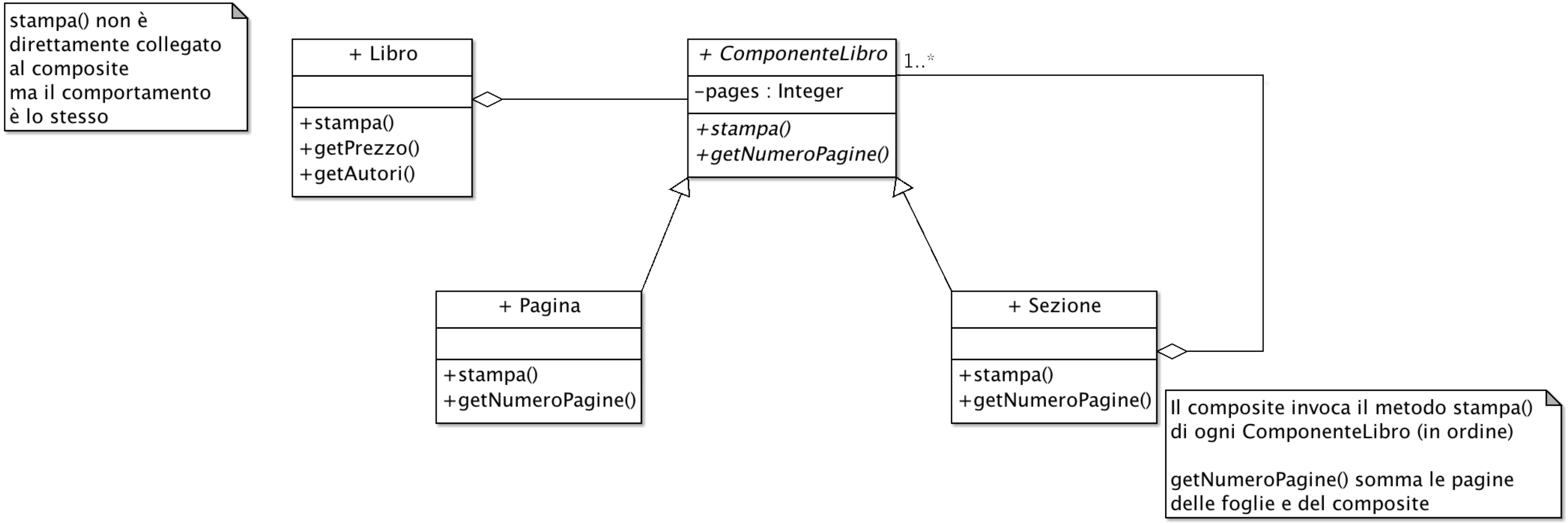
- *Problema*: creare una gerarchia di oggetti (elementari o contenitori) in cui il client “usa” allo stesso modo sia gli oggetti elementari che i contenitori



Esercizio: Libro

- Modellare questo dominio:
  – Un libro è composto da pagine, eventualmente organizzate in sezioni. Ogni sezione può contenere sezioni (una o più) e pagine semplici.
  – E’ possibile stampare una pagina singola, una sezione o l’intero libro.

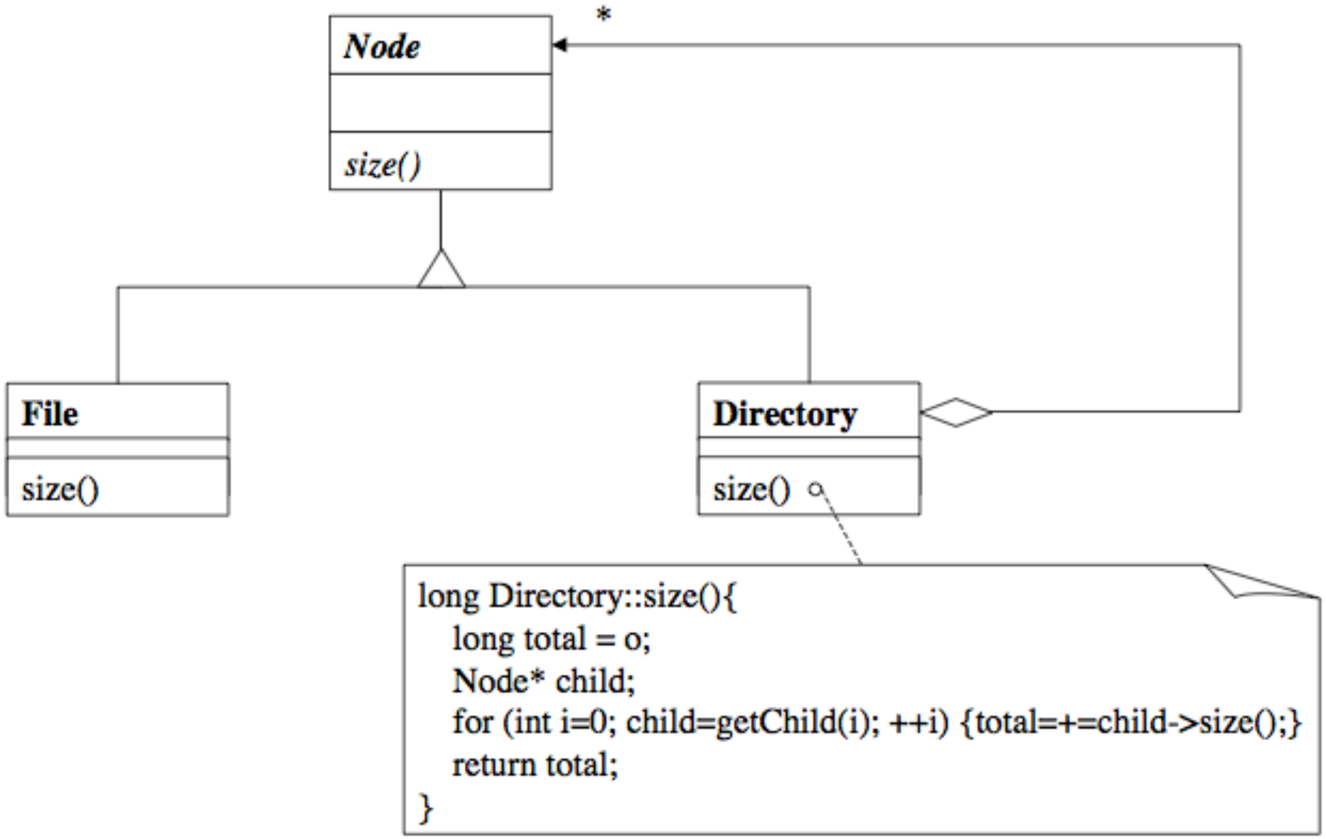
Soluzione


stampa() non è direttamente collegato al composite ma il comportamento è lo stesso
+ Libro
+stampa()
+getPrezzo()
+getAutori()
+ ComponenteLibro
-pages : Integer
+stampa()
+getNumeroPagine()
1..*
+ Pagina
+stampa()
+getNumeroPagine()
+ Sezione
+stampa()
+getNumeroPagine()
Il composite invoca il metodo stampa() di ogni ComponenteLibro (in ordine)
getNumeroPagine() somma le pagine delle foglie e del composite

Esercizio: file system

- Come organizzare un diagramma delle classi (e relativo codice) per modellare un file-system, in cui è possibile conoscere le dimensioni di ogni file e/o directory?

File system



Node

size()

*

File

size()

Directory

size()

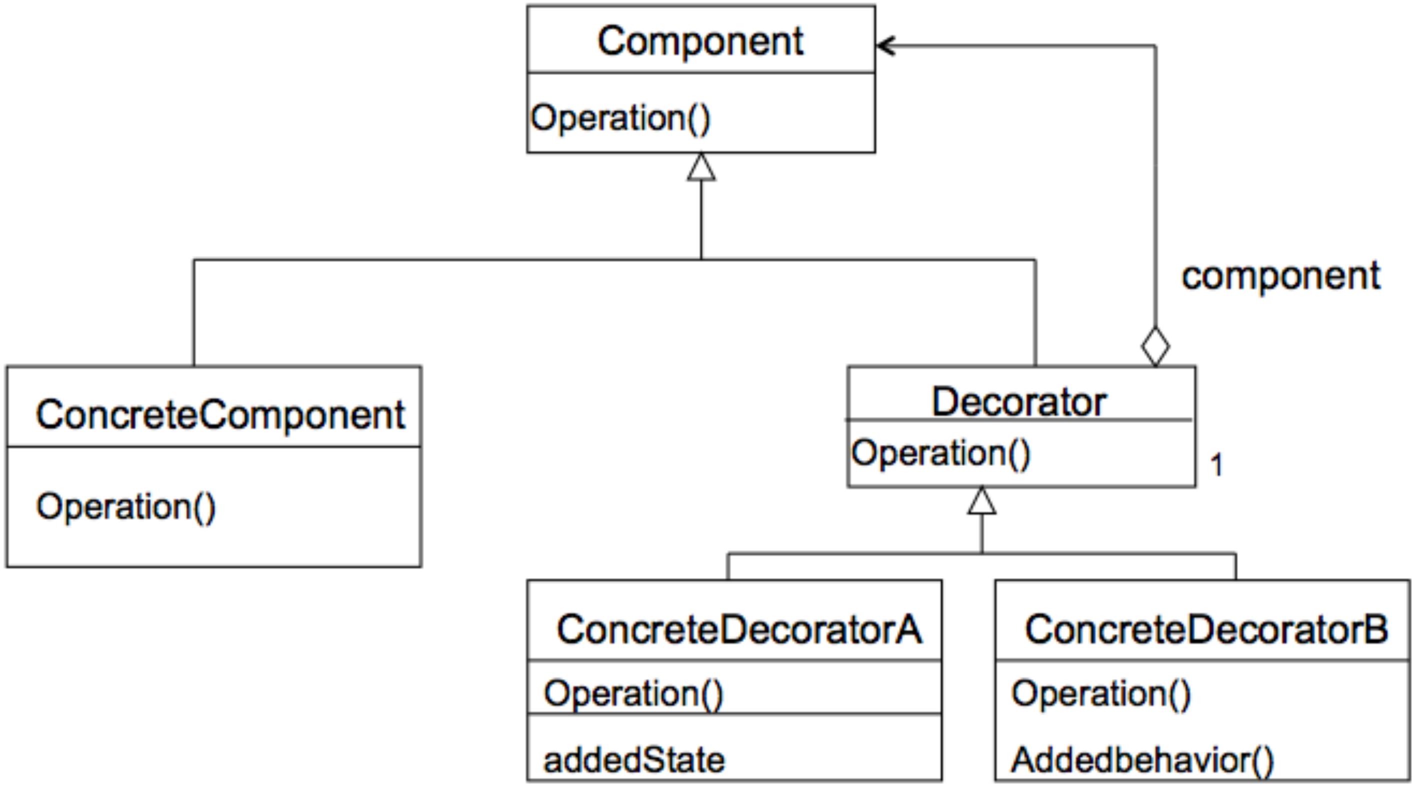
```
long Directory::size(){
   long total = o;
   Node* child;
   for (int i=0; child=getChild(i); ++i) {total=+=child->size();}
   return total;
}
```

Esercizio

- Disegnare un diagramma UML che modella il seguente dominio:
  - una azienda è costituita da Employee che afferiscono a diversi uffici
  - un Engineer è un tipo di Employee
  - un Engineer può assumere l'incarico di capoufficio (AdministrativeManager) o di capoprogetto (ProjectManager)
  - un Engineer può essere capo ufficio ed anche capo progetto di più progetti
- Quale design pattern?

Decorator Pattern

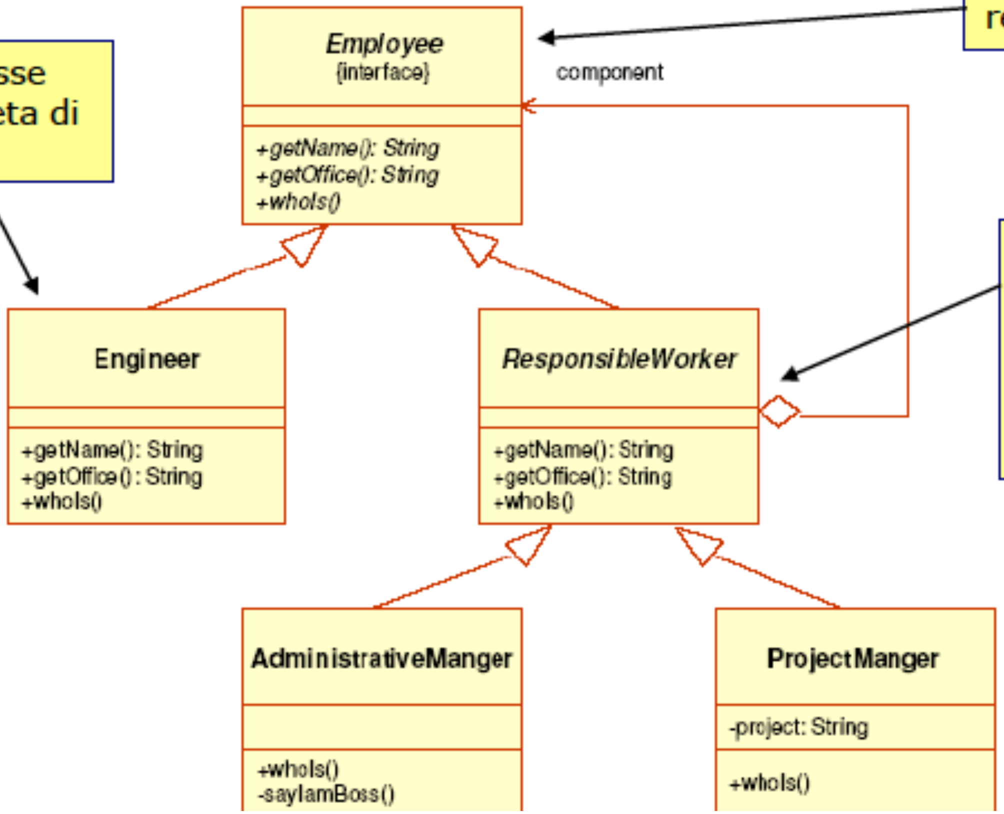
- *Problema*: aggiungere un comportamento ad un oggetto dinamicamente (a run-time)



Soluzione: usare Decorator

La classe concreta di base

L'interfaccia comune serve a garantire la relazione di aggregazione

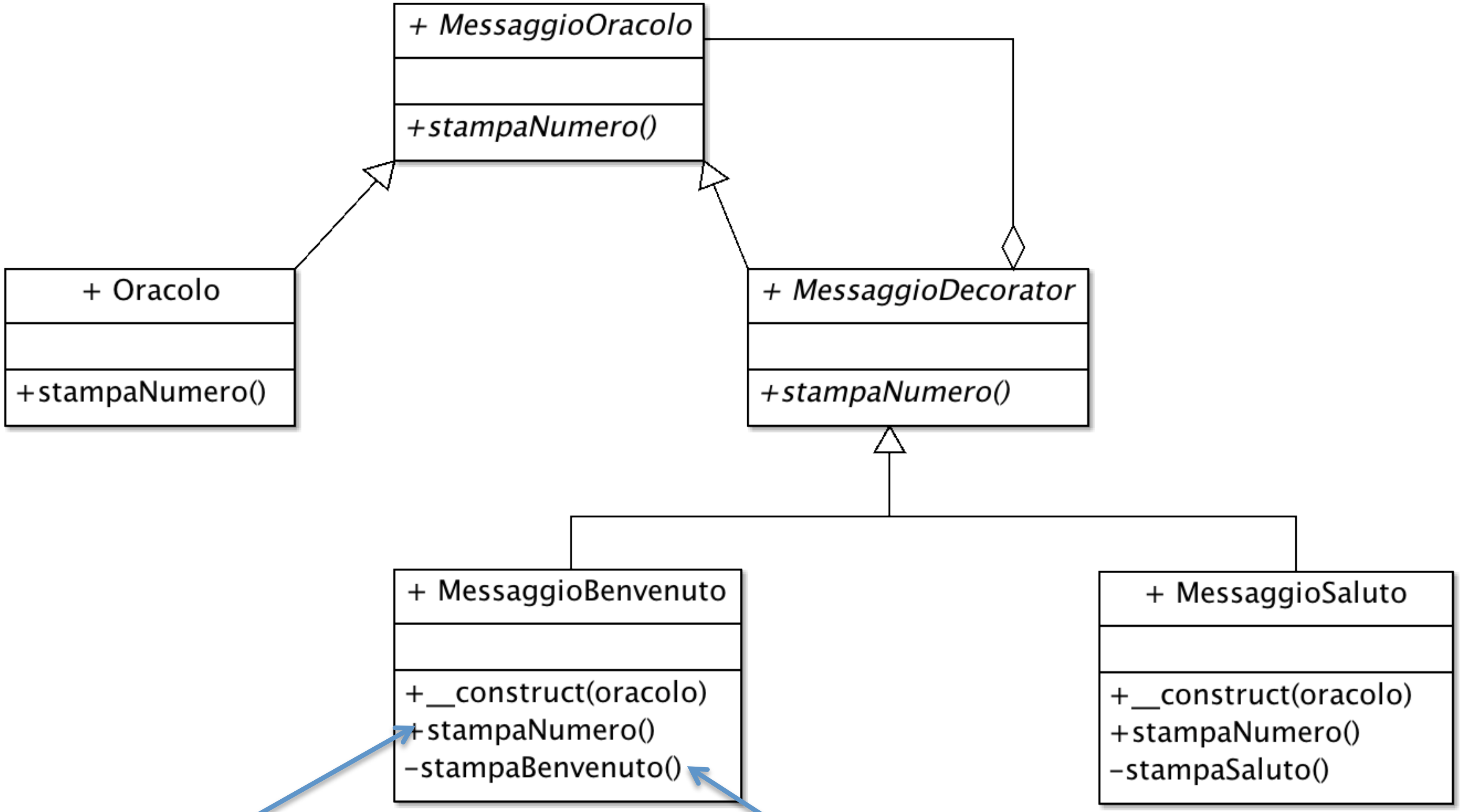


- Le classi di tipo ResponsibleWorker (AdministrativeManager e ProjectManager) sono i Decorator
- Engineer può essere "decorato" con diverse responsabilità ovvero con diverse classi di tipo ResponsibleWorker

Esercizio

- Una classe *Oracolo* esporta un metodo per restituire una numero casuale (*stampaNumero*).
- Estendere la classe per permettere di:
  - stampare un messaggio di benvenuto prima di cercare il numero
  - stampare un messaggio di saluto alla fine
  - stampare entrambi i messaggi precedenti, anche in ordine diverso

Oracolo



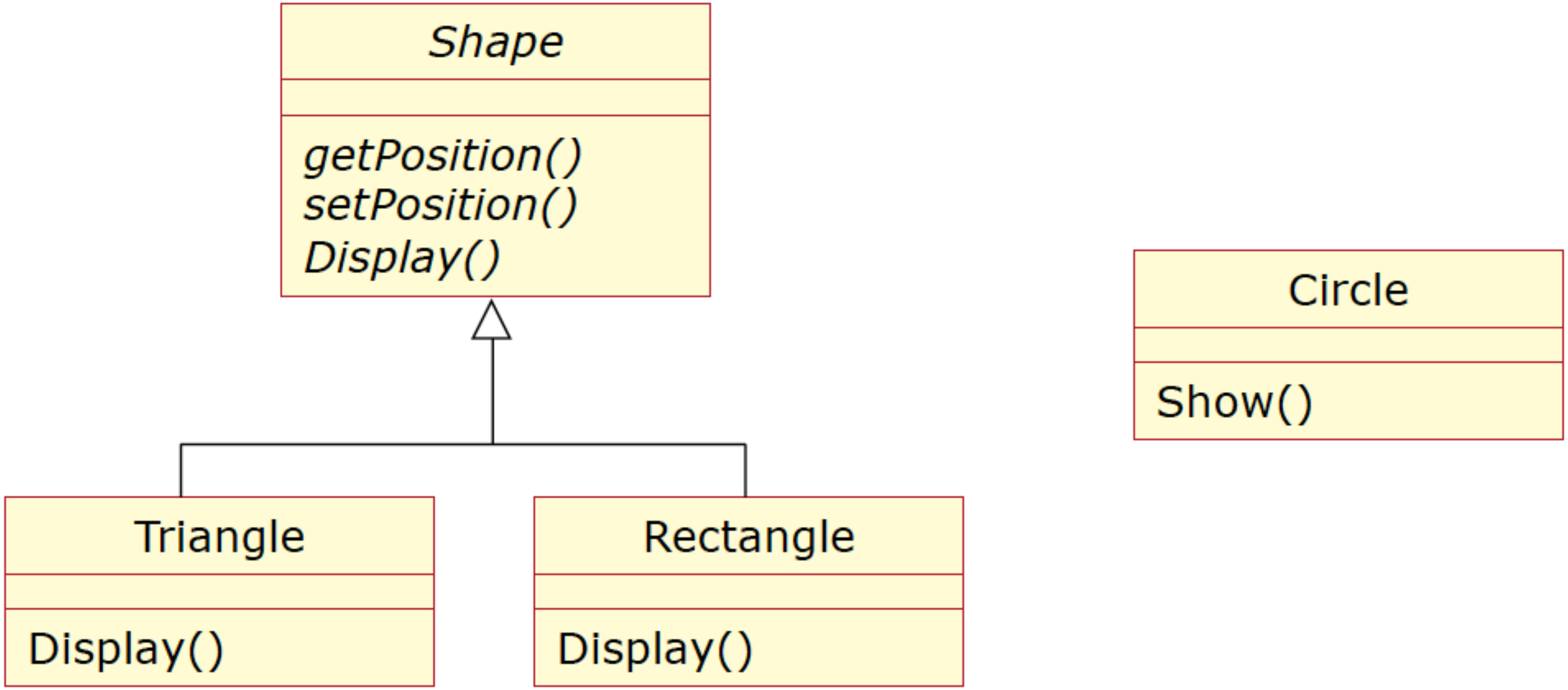
```
$this->stampaBenvenuto;
$oracolo->stampaNumero();
```

```
echo “welcome”;
```

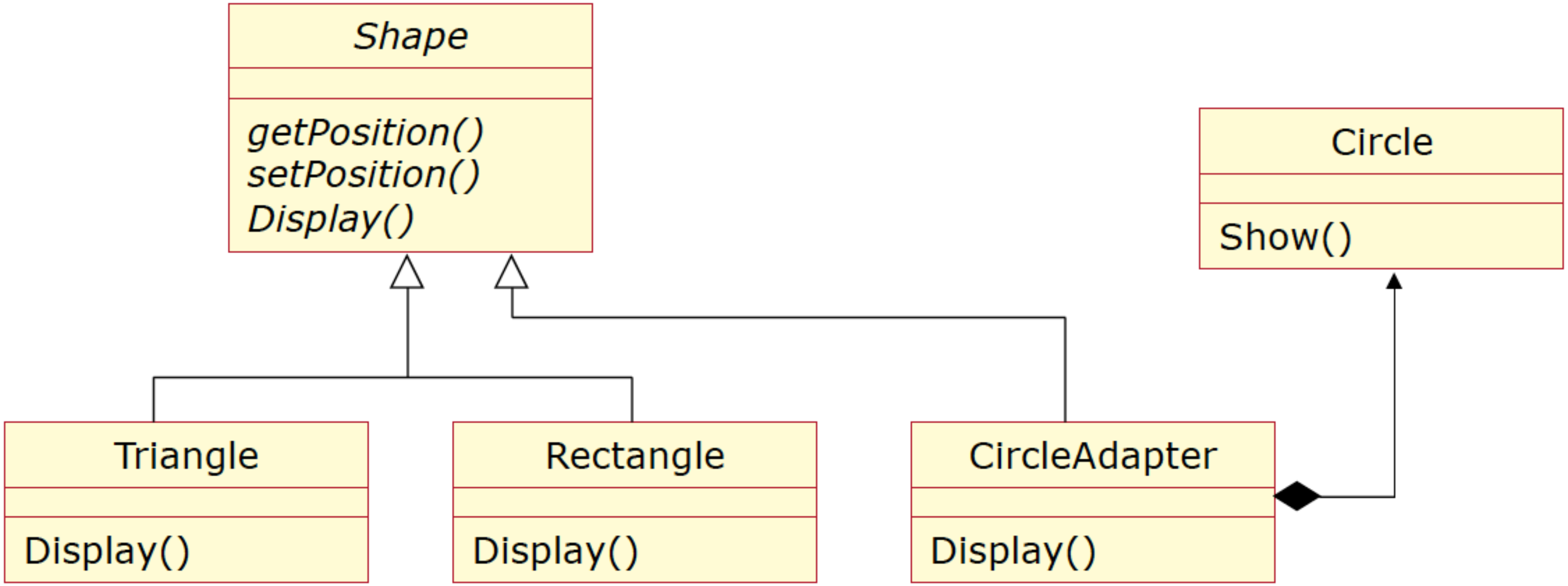
Run-time

```
oracolo = new Oracolo();
oracolo.stampaNumero();         // stampa 327189

welcome = new MessaggioBenvenuto(oracolo);
bye = new MessaggioSaluto(oracolo);

welcome.stampaNumero();
   // stampa “welcome 790789”

bye.stampaNumero();
   // stampa “33909 bye”

all = new MessaggioSaluto(welcome);
all.stampaNumero();
   // stampa “welcome 4446 bye”

crazy = new MessaggioBenvenuto(new MessaggioSaluto(oracolo))
```

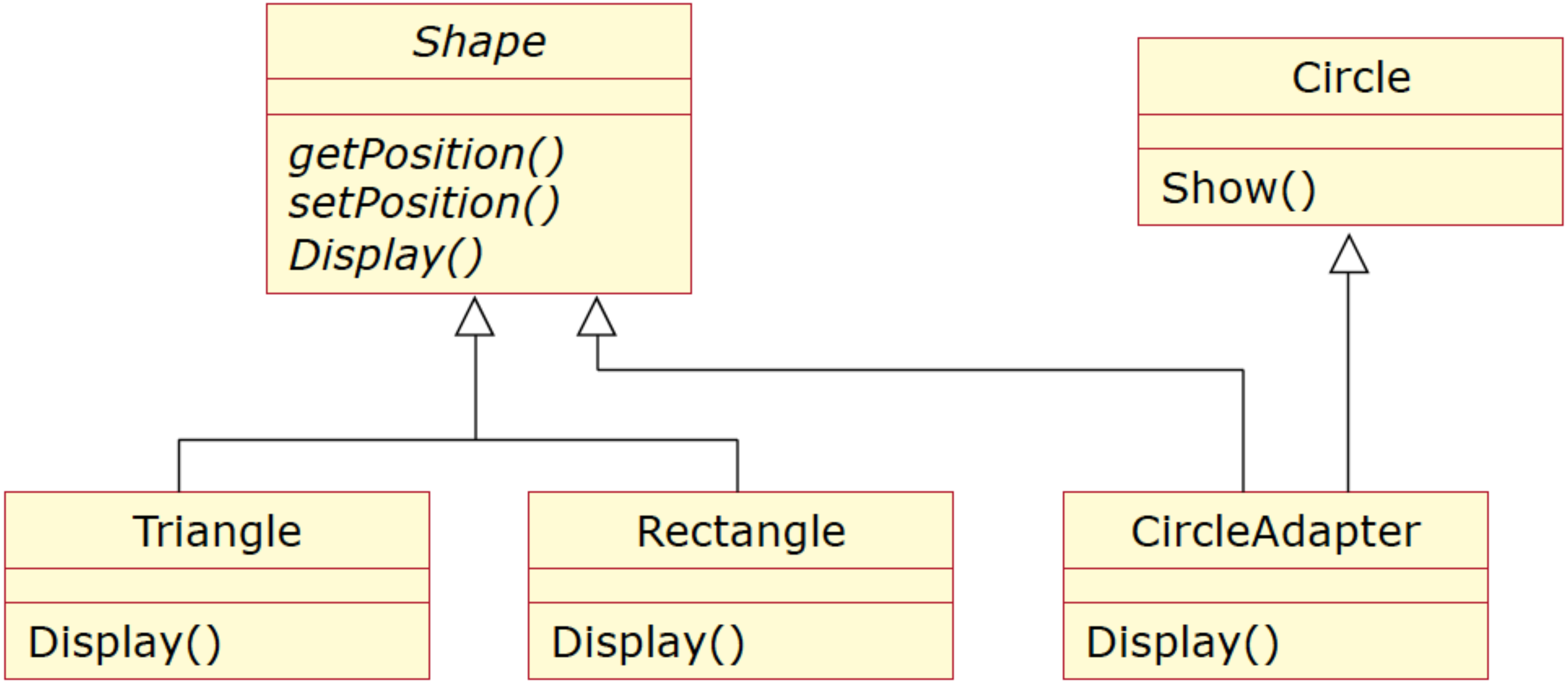
Problema


Shape
getPosition()
setPosition()
Display()
Triangle
Display()
Rectangle
Display()
Circle
Show()

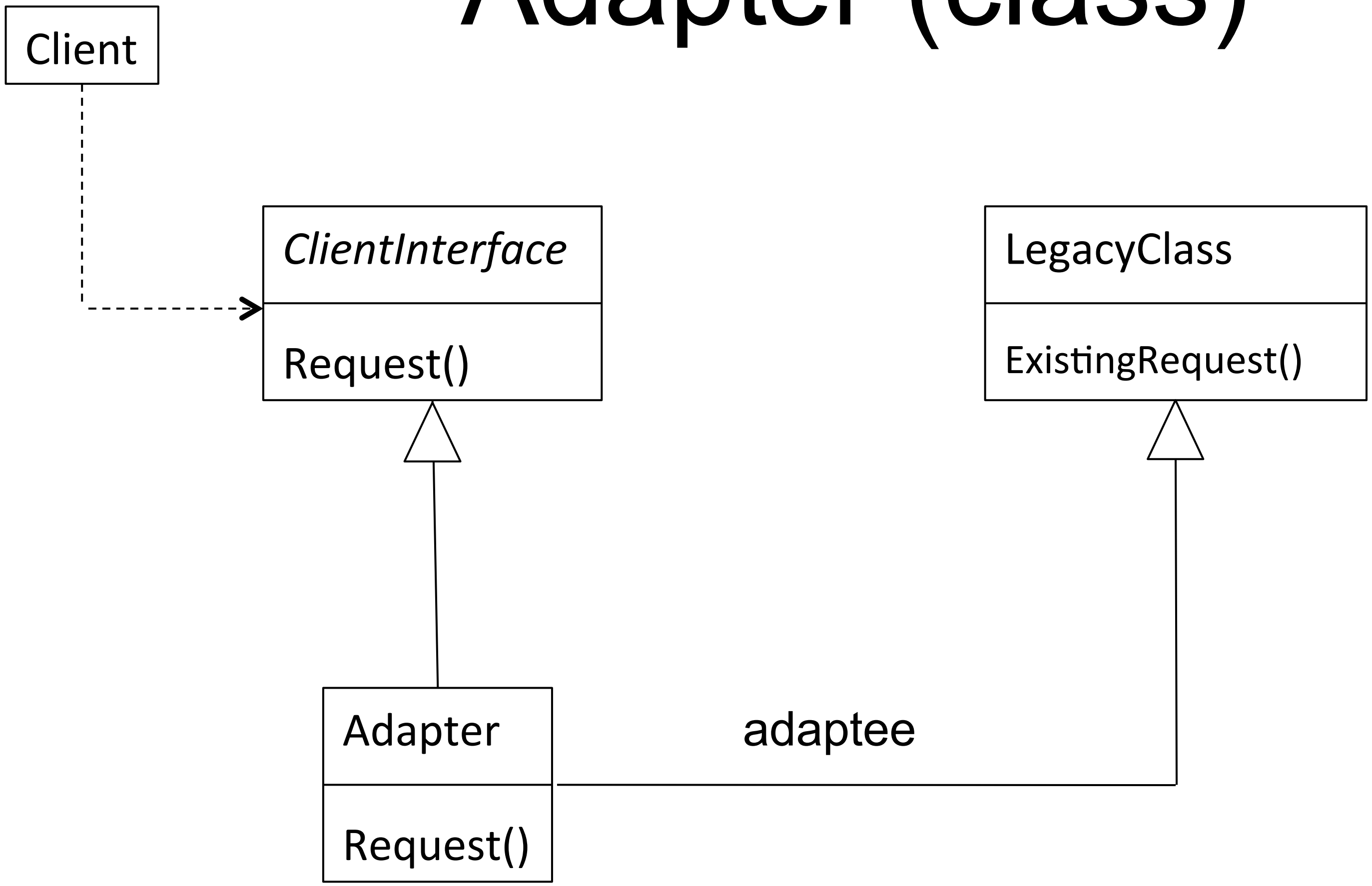
Soluzione 1: Adapter object



Soluzione 2: Adapter class



Adapter (class)


Client

ClientInterface

Request()

LegacyClass

ExistingRequest()

Adapter

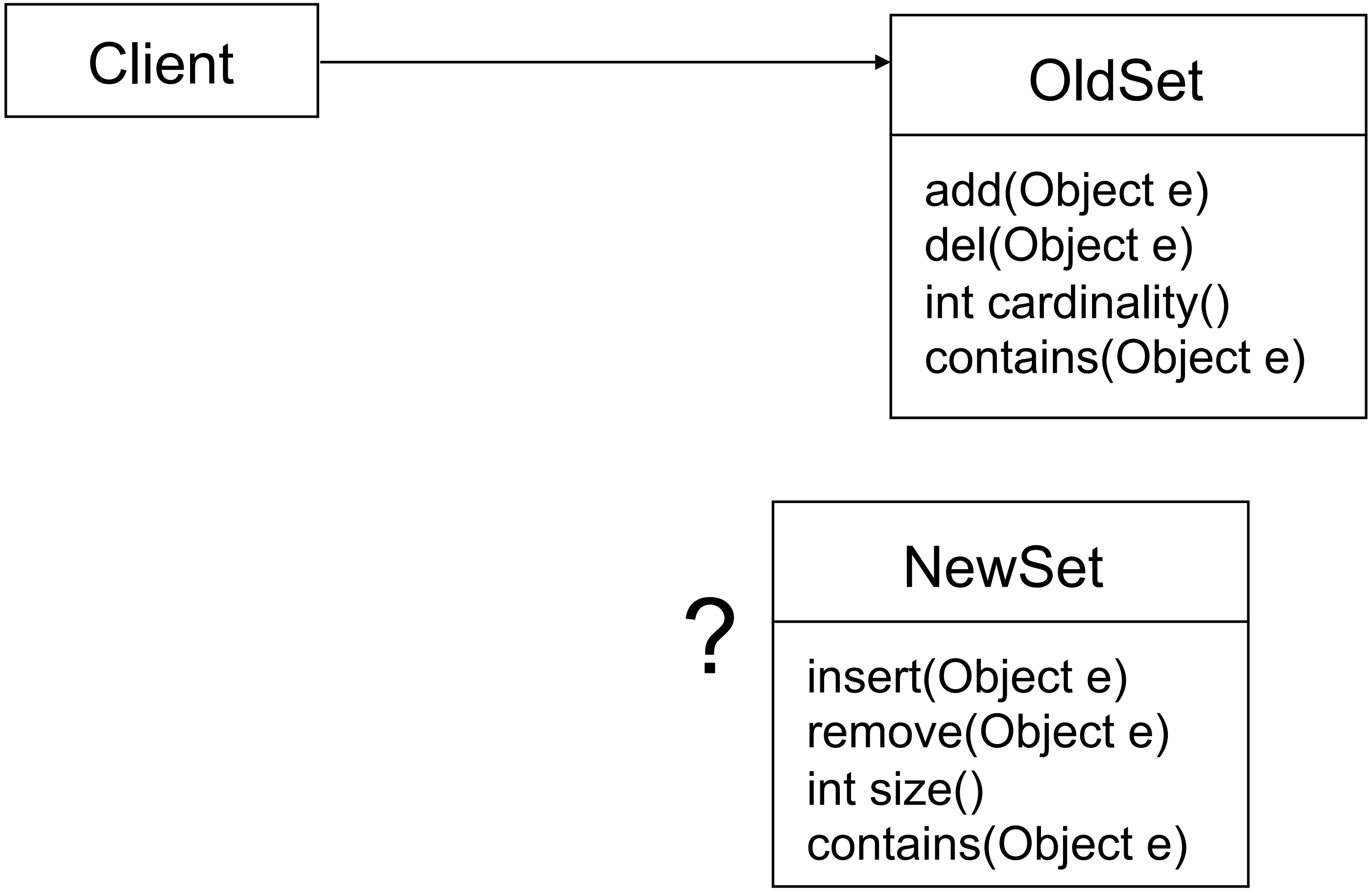
Request()

adaptee

Esempio: insiemi

- There are many ways to implement a set
- Assume that:
  - Existing systems based on sets from a local library.
  - The local version is inadequate (e.g., poor performance)
- We acquire a better set class, BUT:
  - The new set has a different interface
  - And we have no access to the source code!
- Solution: A class or object set adapter:
  - Same interface as existing system's expect.
  - Simply translates to the new set's interface.

Class diagram

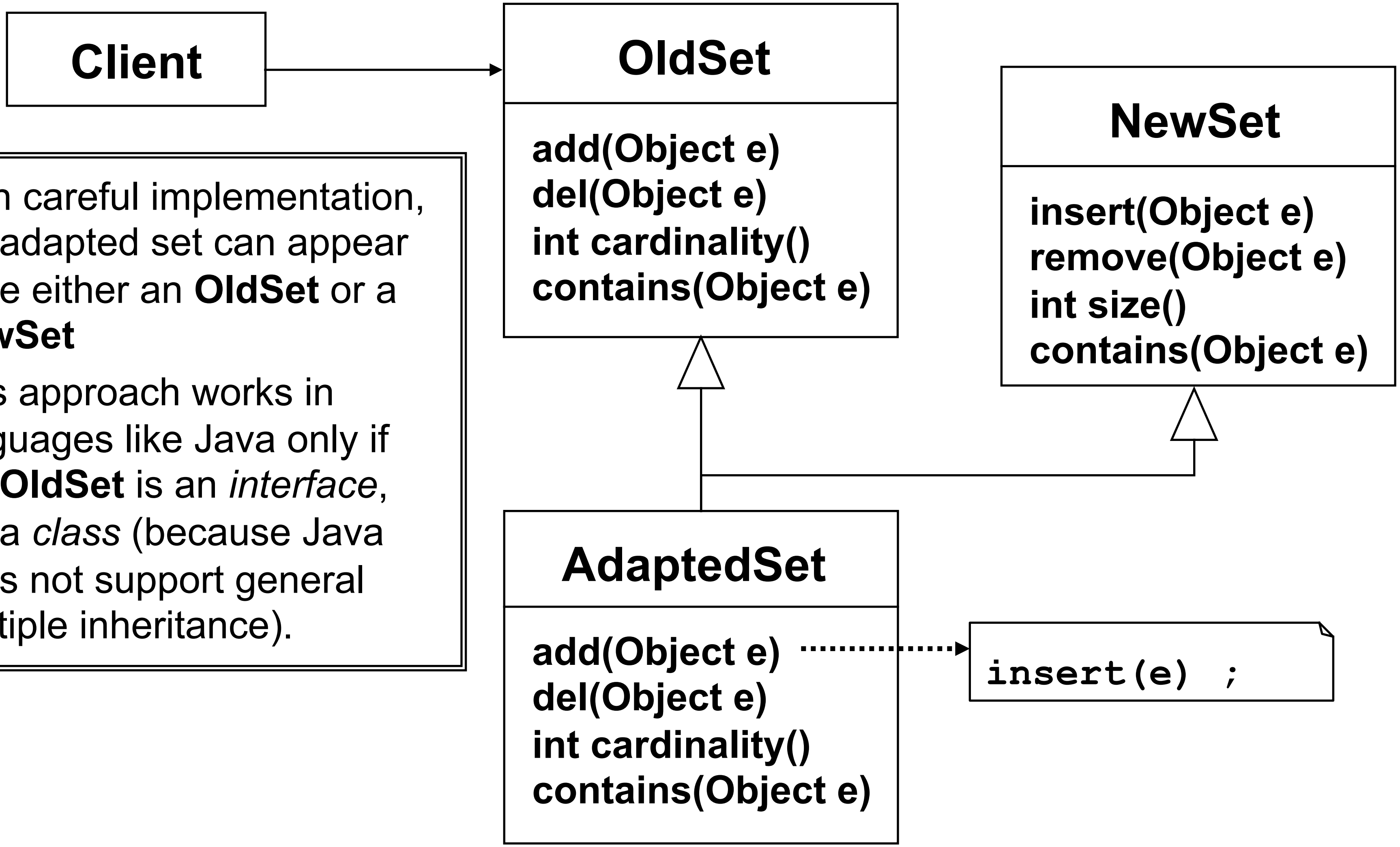


Adapter description

Intent

- Convert the interface of a class to an interface expected by the users of the class.
- Allows classes to work together even though they expect incompatible interfaces.

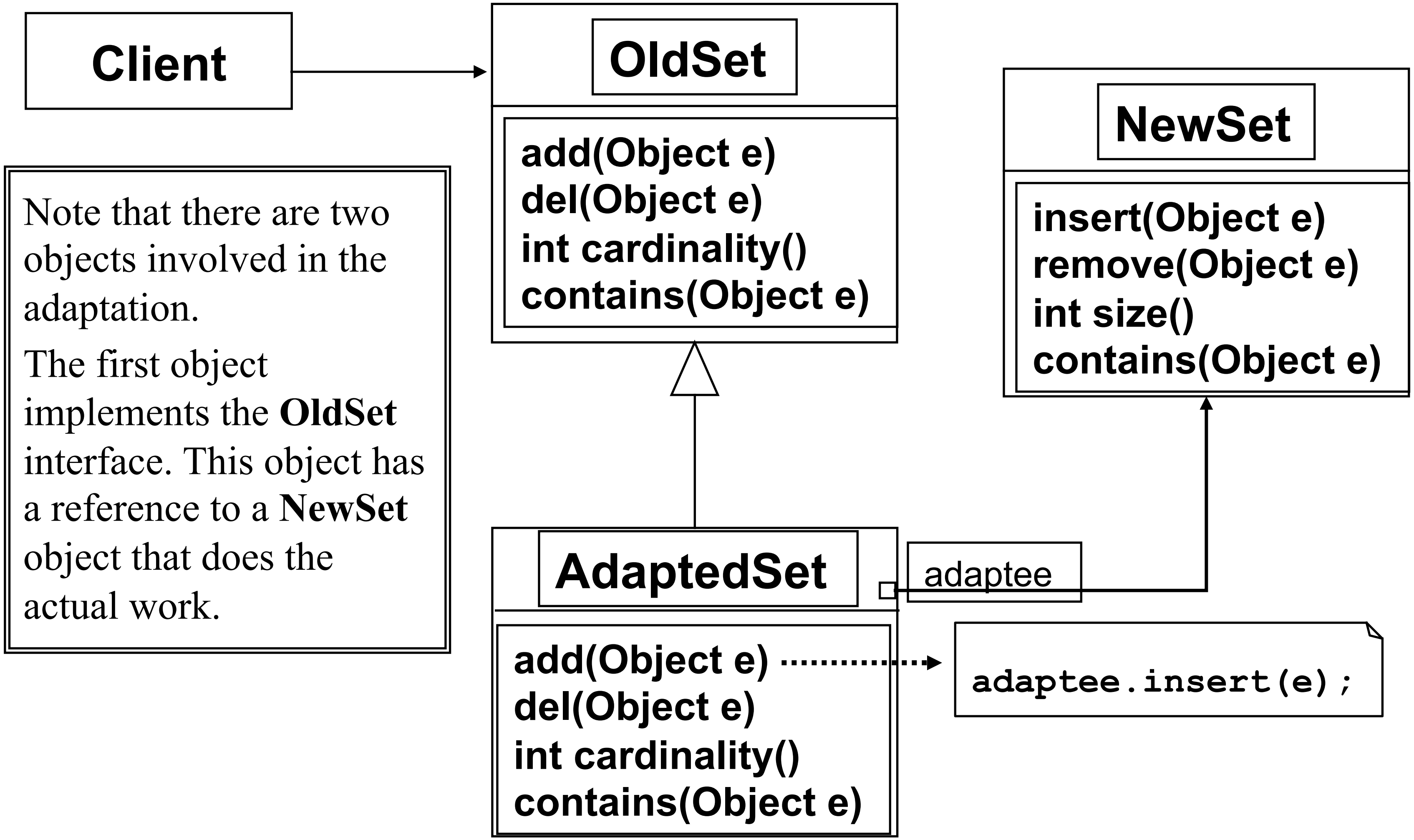
Class Adaptation (via Inheritance)



With careful implementation, the adapted set can appear to be either an **OldSet** or a **NewSet**

This approach works in languages like Java only if the **OldSet** is an *interface*, not a *class* (because Java does not support general multiple inheritance).

Object Adaptation (via Delegation)



Client

OldSet

add(Object e)
del(Object e)
int cardinality()
contains(Object e)

NewSet

insert(Object e)
remove(Object e)
int size()
contains(Object e)

AdaptedSet

add(Object e)
del(Object e)
int cardinality()
contains(Object e)

adaptee

```
adaptee.insert(e);
```

Note that there are two objects involved in the adaptation.

The first object implements the **OldSet** interface. This object has a reference to a **NewSet** object that does the actual work.

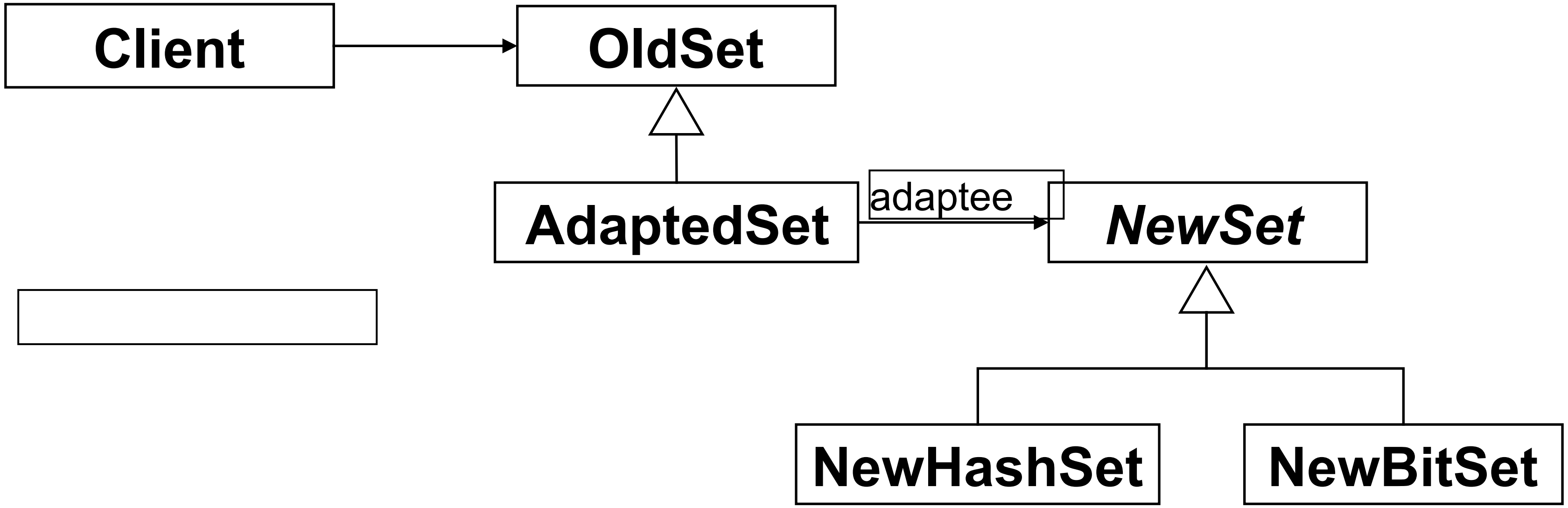
Variant: Adapt Multiple Versions of NewSet

(Object only) Several subclasses to adapt:

Too expensive to adapt each subclass.

Create single adapter to superclass interface.

Configure the **AdaptedSet** with the specific **NewSet** at run-time.



Consequences - Class Adapters

- Creates concrete adapter for a *specific* Adaptee (e.g., **NewSet**)
- Cannot adapt a class and *all* its subclasses
- Only one object is created
  - The object has two faces (or identities).
  - But there is no need for indirection.
- Can override Adaptee (e.g., **NewSet**) behavior, as Adapter is a subclass of Adaptee

Consequences - Object Adapters

- Single Adapter class handles many Adaptees
  - Any class that has the specified *Adaptee* interface (e.g., all **NewSets**).
  - Can adapt the Adaptee class and all its subclasses.
- Hard to override Adaptee behavior
  - Because the Adapter *uses* but does not *inherit from* the Adaptee interface.
  - Overriding means:
    - Subclassing Adaptee to modify behavior.
    - Adapting the subclass.
    - Often not be worth the effort.and adapt this.

Other Issues

- How much adapting does adapter do?
  - Simple forwarding of requests (renaming)?
  - Different set of operations & semantics?
  - At what point do the Adaptee and Adapter interfaces diverge so much that “adaption” is no longer the correct term?

Implementation

- C++ Class Adapters

  `public` inheritance from *Target* class.

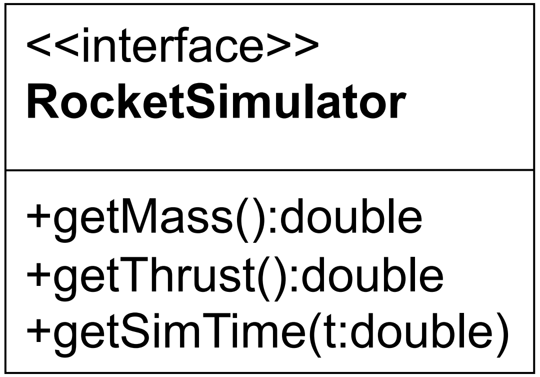
  `private` inheritance from *Adaptee* class.

  => *Adapter* of type *Target* but not *Adaptee.*

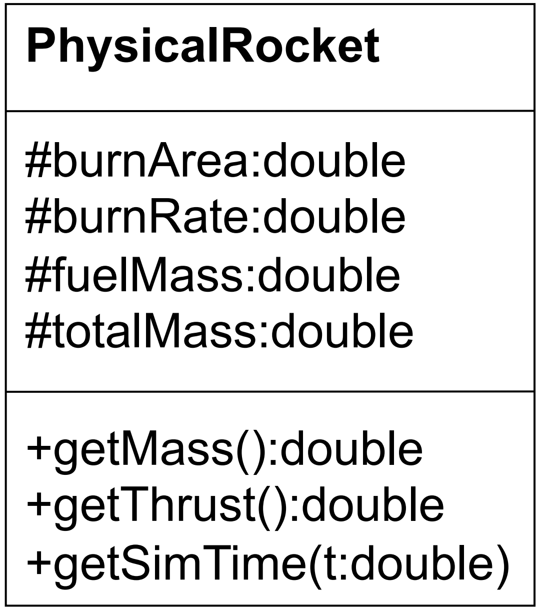
- Adapting to Java interfaces
  - Similar to class adaptation via multiple inheritance.
  - Lighter weight than class adapting.
  - No carrying of useless superclass baggage.

Adattare

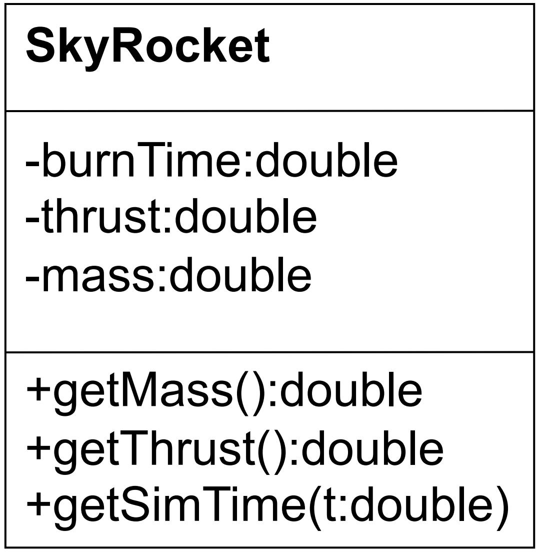
- Definire una classe che adatti PhysicalRocket a RocketSimulator



```
<<interface>>
RocketSimulator
---------------------------
+getMass():double
+getThrust():double
+getSimTime(t:double)
```



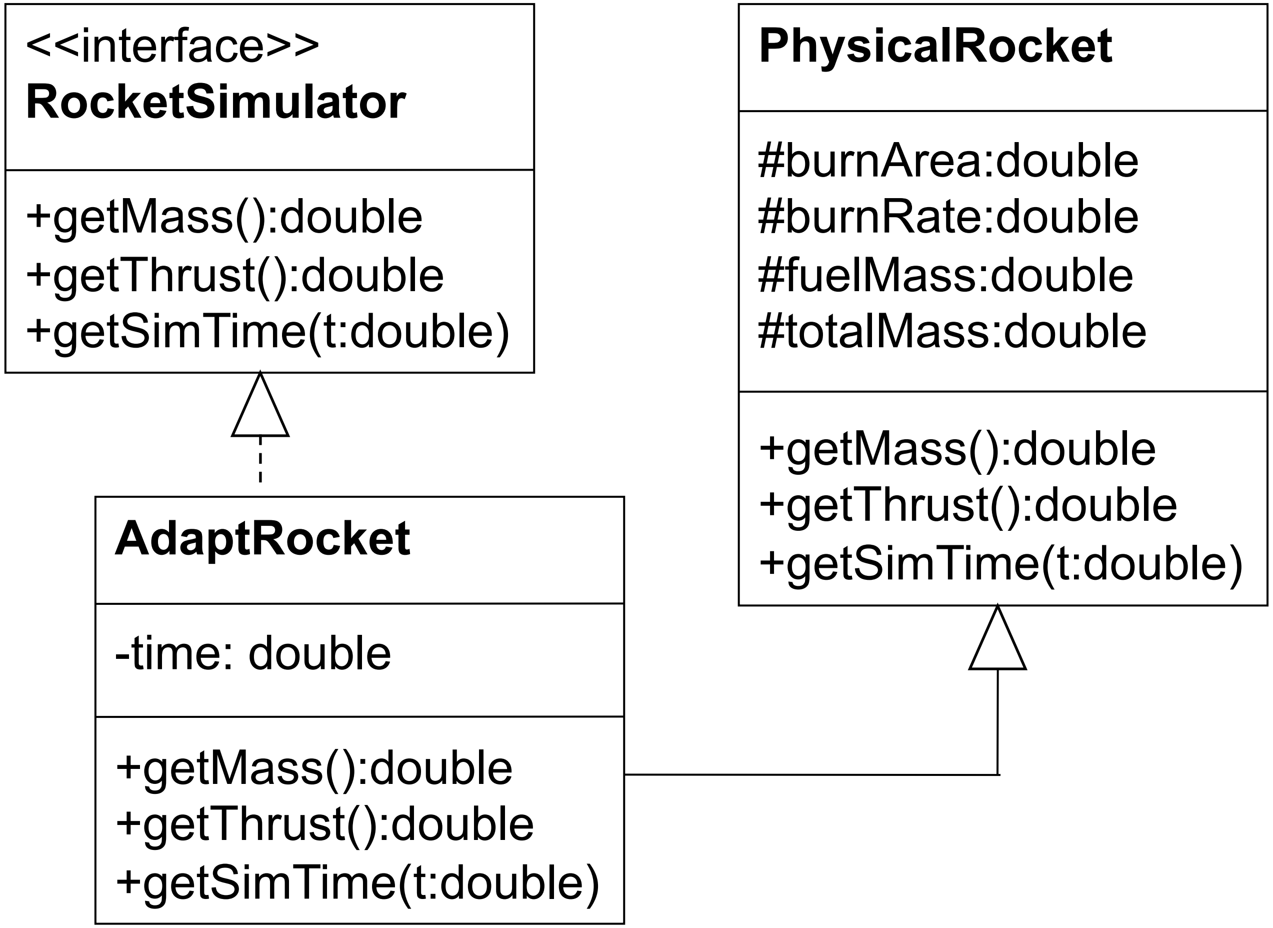
```
PhysicalRocket
---------------------------
#burnArea:double
#burnRate:double
#fuelMass:double
#totalMass:double
---------------------------
+getMass():double
+getThrust():double
+getSimTime(t:double)
```



```
SkyRocket
---------------------------
-burnTime:double
-thrust:double
-mass:double
---------------------------
+getMass():double
+getThrust():double
+getSimTime(t:double)
```

- Adesso definire una classe che adatti PhysicalRocket a Sky Rocket

Soluzione 1



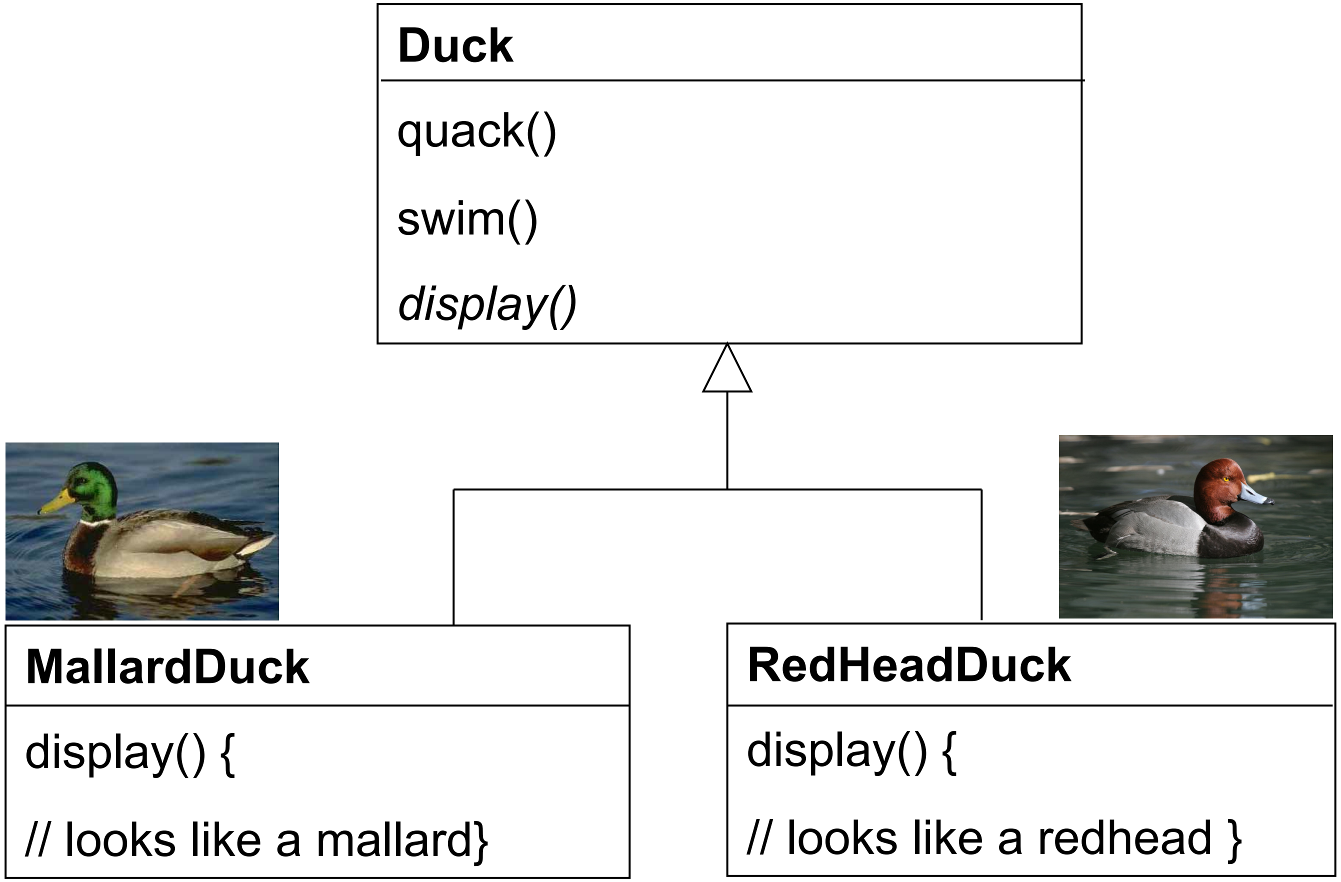
<<interface>>
RocketSimulator
+getMass():double
+getThrust():double
+getSimTime(t:double)
PhysicalRocket
#burnArea:double
#burnRate:double
#fuelMass:double
#totalMass:double
+getMass():double
+getThrust():double
+getSimTime(t:double)
AdaptRocket
-time: double
+getMass():double
+getThrust():double
+getSimTime(t:double)

Pattern GoF comportamentali

I pattern comportamentali risolvono le più comuni tipologie di interazione tra gli oggetti

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Esercizio: papere



Duck

quack()

swim()

display()

MallardDuck

display() {

// looks like a mallard}

RedHeadDuck

display() {

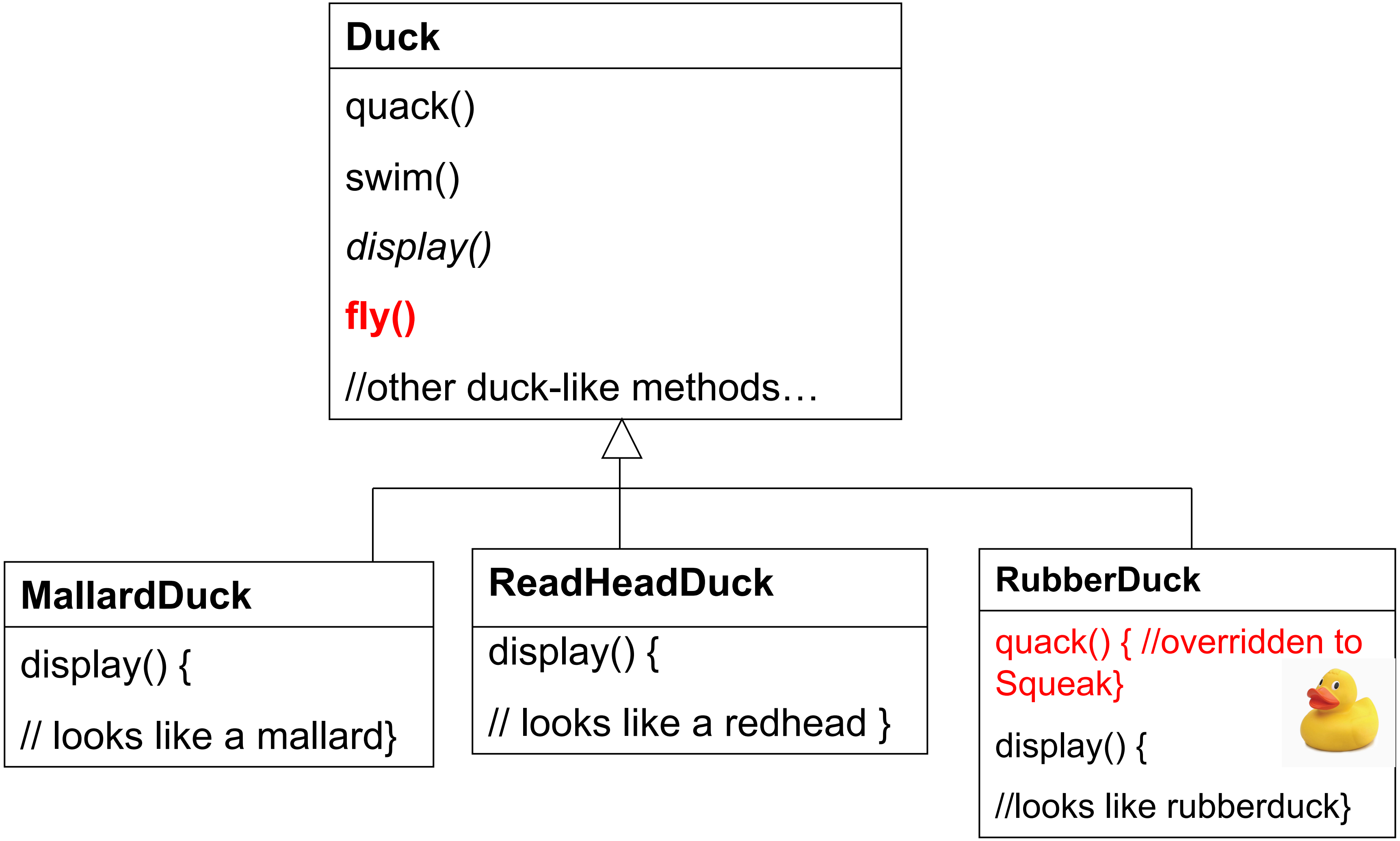
// looks like a redhead }

[da: ‘Head First. Design Patterns’]

Estendibile?

- Come aggiungere un nuovo tipo di anatra che fa un verso diverso dalle altre?
- Come aggiungere il comportamento *fly()*?

Problemi?



Duck

quack()

swim()

display()

fly()

//other duck-like methods…

MallardDuck

display() {

// looks like a mallard}

ReadHeadDuck

display() {

// looks like a redhead }

RubberDuck

quack() { //overridden to Squeak}

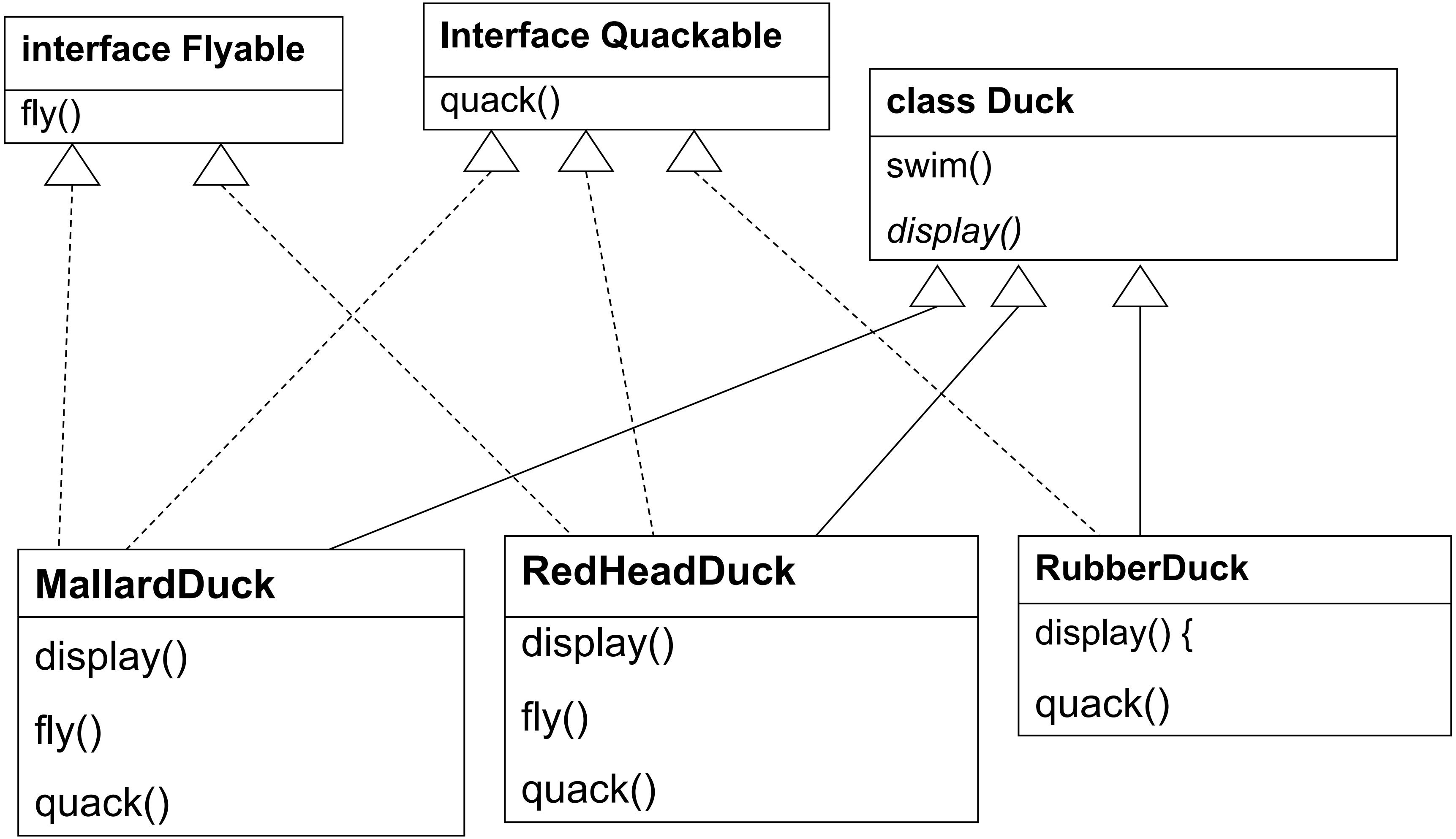
display() {

//looks like rubberduck}

Estendibile?

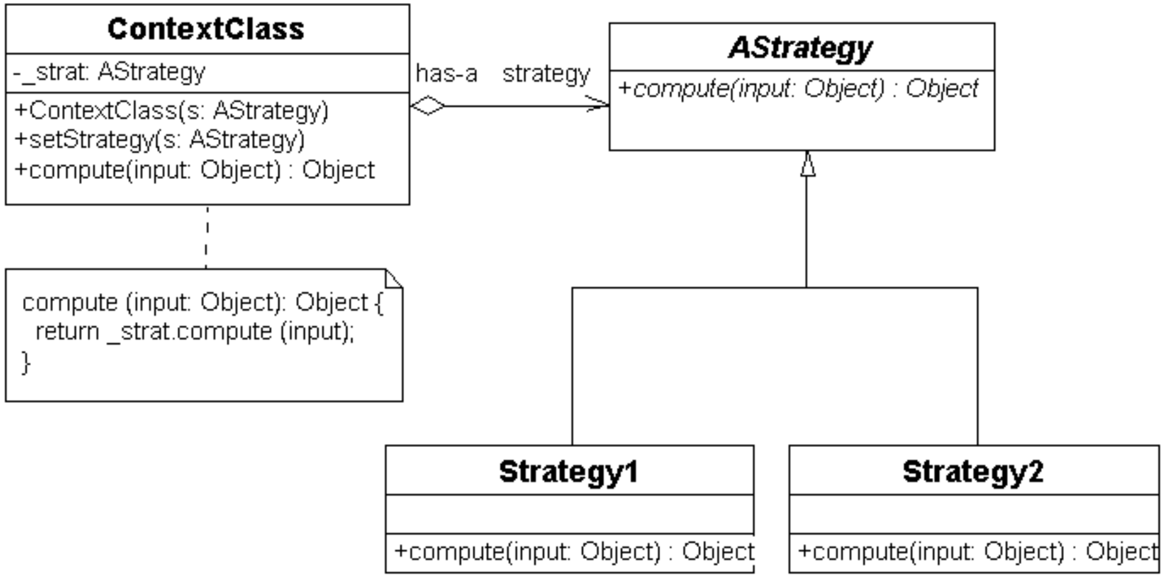
- Come aggiungere un nuovo tipo di anatra, ad esempio muta e che non vola?
- Basta fare *override* dei metodi quack() e fly()?
- E per nuovi tipi di anatra che hanno comportamenti – quack() e fly() – parzialmente sovrapposti alle altre?
- Come gestire le diverse combinazioni?

Interfacce? Problemi?

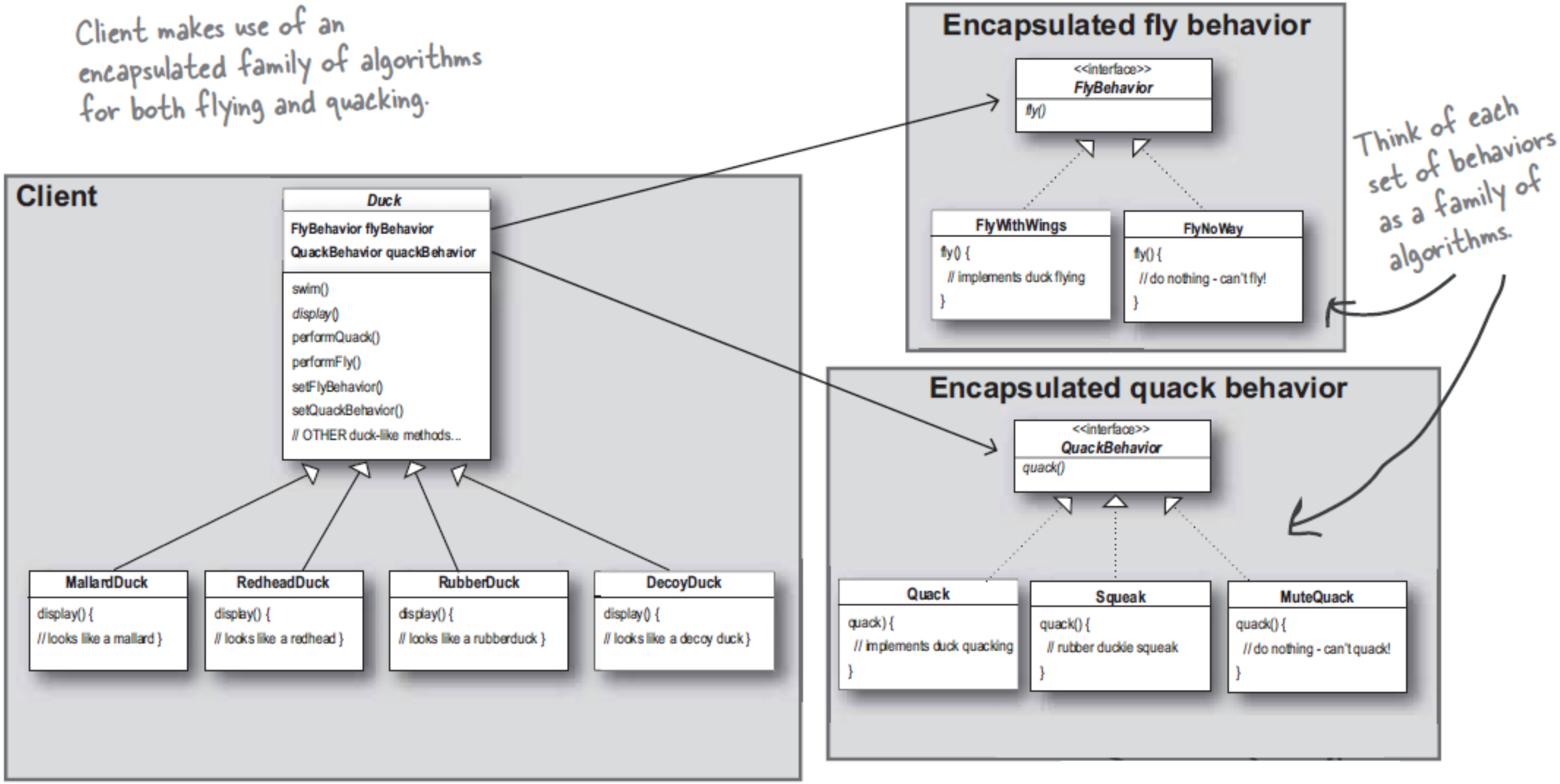


Pattern Strategy

- *Problema*: definire una famiglia di algoritmi e renderli interscambiabili
  - modificare il comportamento di una classe a run-time e disaccoppiare il comportamento (Algoritmo) dalla classe (Client) che lo usa


ContextClass
-_strat: AStrategy
+ContextClass(s: AStrategy)
+setStrategy(s: AStrategy)
+compute(input: Object) : Object
has-a
strategy
AStrategy
+compute(input: Object) : Object
compute (input: Object): Object {
return _strat.compute (input);
}
Strategy1
+compute(input: Object) : Object
Strategy2
+compute(input: Object) : Object

Soluzione con Strategy



Client makes use of an encapsulated family of algorithms for both flying and quacking.

Client

Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...

MallardDuck

display() {
// looks like a mallard }

RedheadDuck

display() {
// looks like a redhead }

RubberDuck

display() {
// looks like a rubberduck }

DecoyDuck

display() {
// looks like a decoy duck }

Encapsulated fly behavior

<<interface>>
FlyBehavior

fly()

FlyWithWings

fly() {
// implements duck flying
}

FlyNoWay

fly() {
// do nothing - can't fly!
}

Think of each set of behaviors as a family of algorithms.

Encapsulated quack behavior

<<interface>>
QuackBehavior

quack()

Quack

quack) {
// implements duck quacking
}

Squeak

quack() {
// rubber duckie squeak
}

MuteQuack

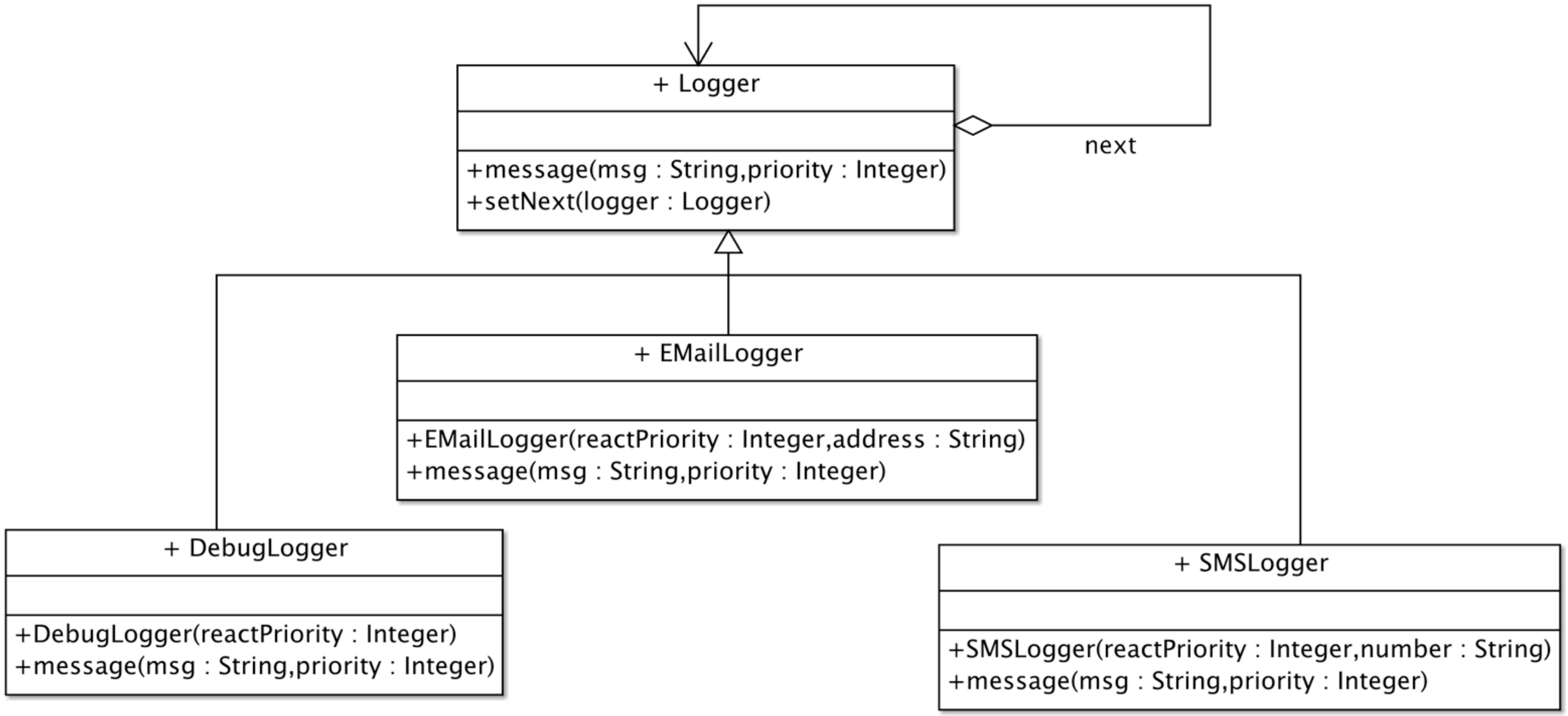
quack() {
// do nothing - can't quack!
}

[da: ‘Head First. Design Patterns.’

Esercizio

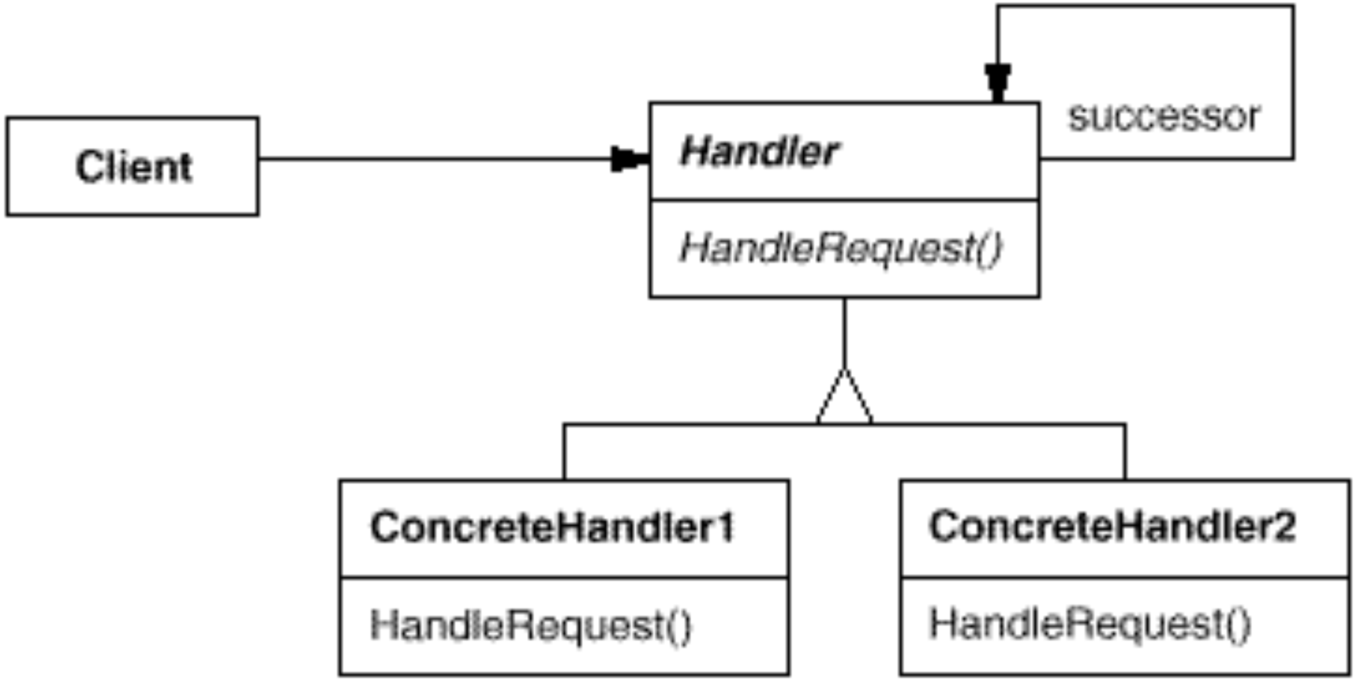
- Disegnare un diagramma UML che modella il seguente dominio:
  - Un'azienda deve gestire le richieste di credito dei clienti (customers).
  - Internamente l'azienda si organizza in diversi livelli:
    - Il livello più basso (vendor) può approvare le richieste fino a un dato importo.
    - Le richieste che superano questo importo vanno gestite da un livello superiore (sales manager), il quale ha un altro importo massimo da gestire.
    - Oltre tale importo le richieste sono gestiste da un 'client account manager'

Identifica il pattern

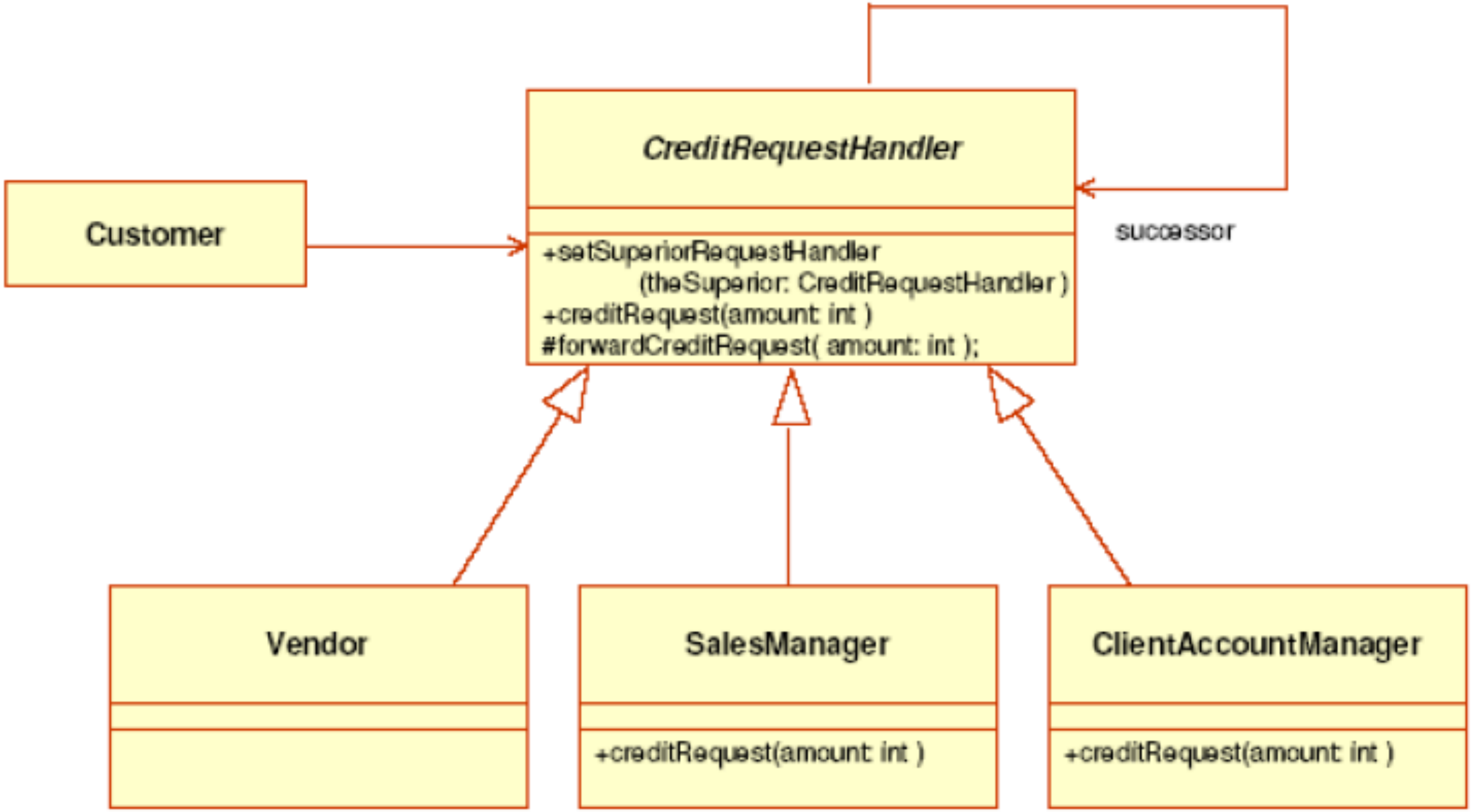

+ Logger
+message(msg : String,priority : Integer)
+setNext(logger : Logger)
next
+ EMailLogger
+EMailLogger(reactPriority : Integer,address : String)
+message(msg : String,priority : Integer)
+ DebugLogger
+DebugLogger(reactPriority : Integer)
+message(msg : String,priority : Integer)
+ SMSLogger
+SMSLogger(reactPriority : Integer,number : String)
+message(msg : String,priority : Integer)

Pattern: chain of responsibility

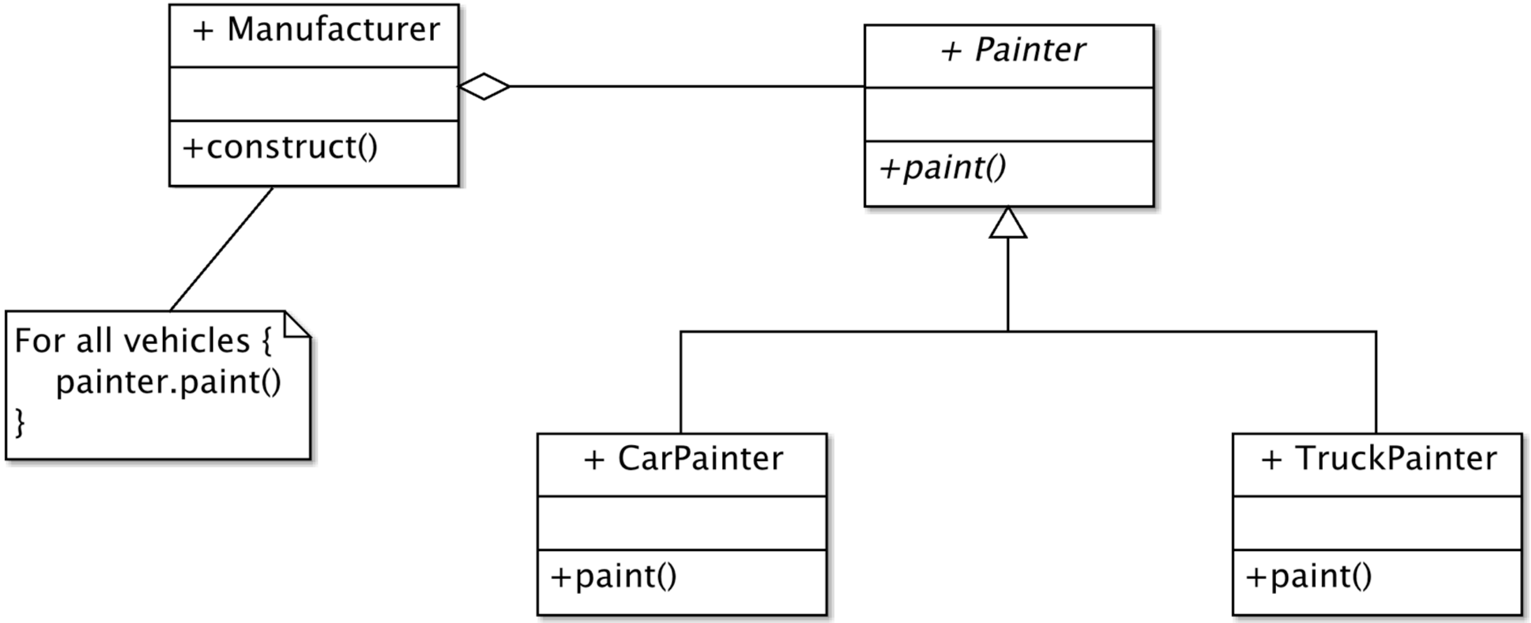
Chain of Responsibility



Chain of Responsibility

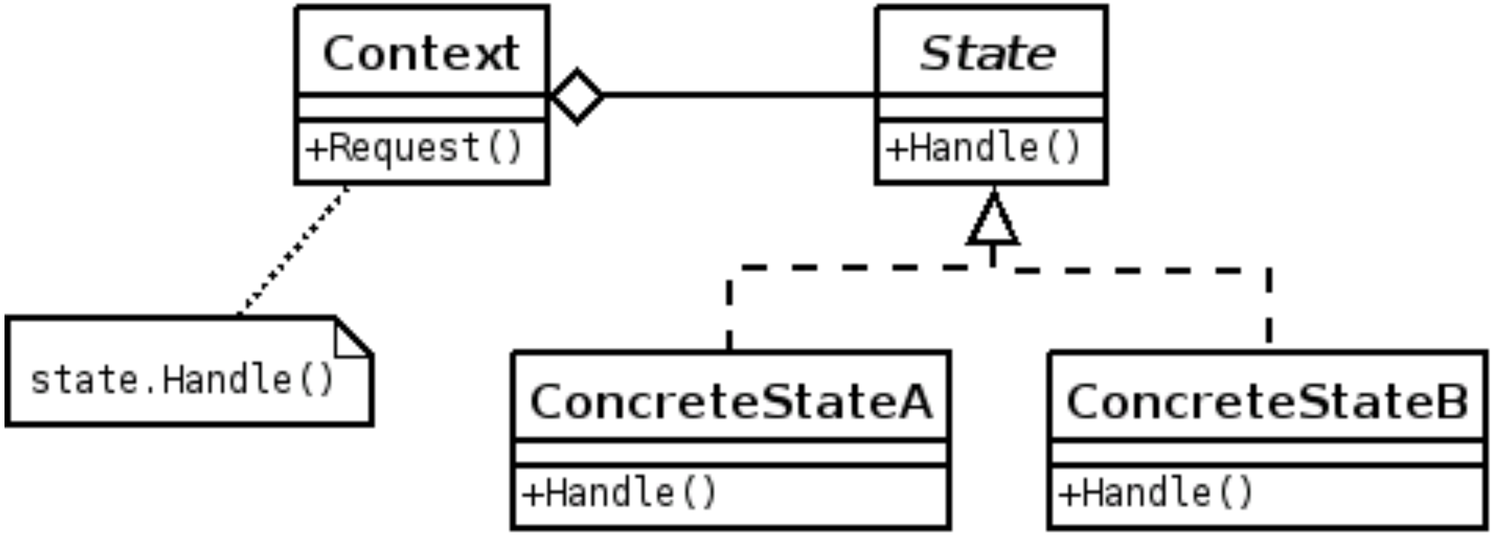

CreditRequestHandler
+setSuperiorRequestHandler
(theSuperior: CreditRequestHandler)
+creditRequest(amount: int)
#forwardCreditRequest(amount: int);
successor
Customer
Vendor
SalesManager
+creditRequest(amount: int)
ClientAccountManager
+creditRequest(amount: int)
]

Identifica il pattern



Pattern: State

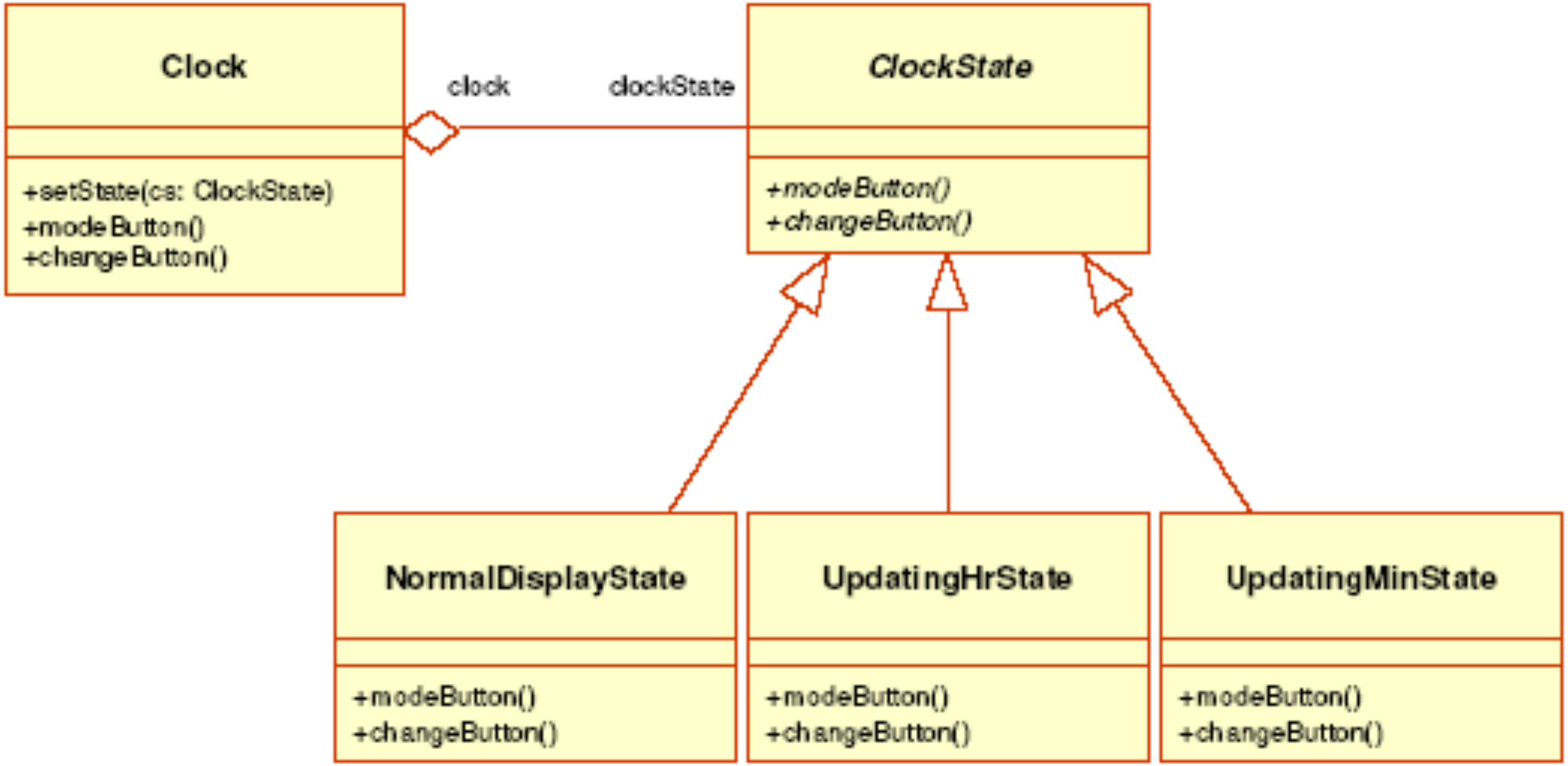
State


Context
+Request()
State
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Esercizio

- Modellare il seguente dominio:
  - Un orologio ha due pulsanti: MODE e CHANGE*.*
  - MODE permette di scegliere tra: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”
  - CHANGE esegue operazioni diverse in base alla modalità:
    - accendere la luce del display, se è in modalità di “visualizzazione normale”
    - incrementare in una unità le ore o i minuti, se è in modalità di “modifica” di ore o di minuti

State


Clock
+setState(cs: ClockState)
+modeButton()
+changeButton()
clock
clockState
ClockState
+modeButton()
+changeButton()
NormalDisplayState
+modeButton()
+changeButton()
UpdatingHrState
+modeButton()
+changeButton()
UpdatingMinState
+modeButton()
+changeButton()

On design patterns

Which GoF pattern inspires this diagram?

a) Bridge

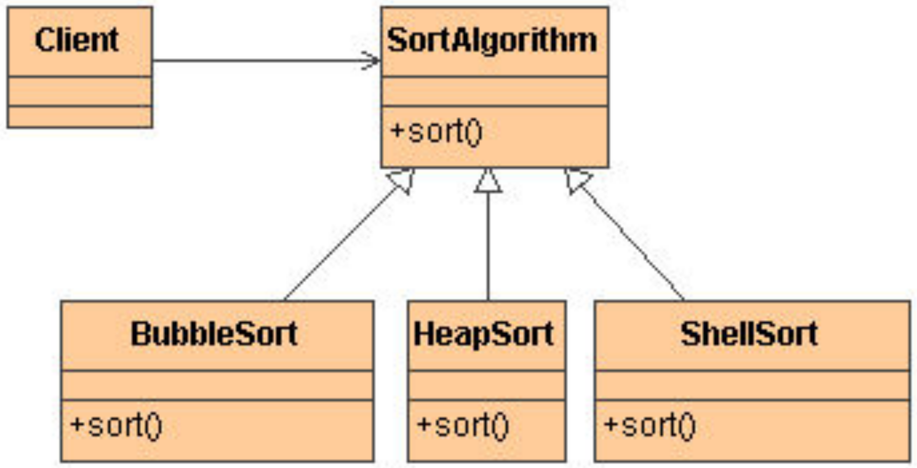
b) Composite

c) Abstract factory

d) [✓ Strategy]

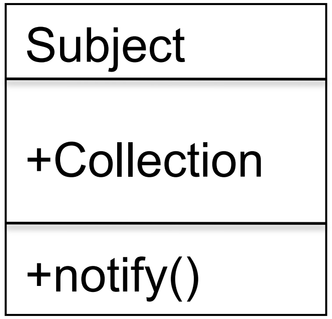
e) Singleton

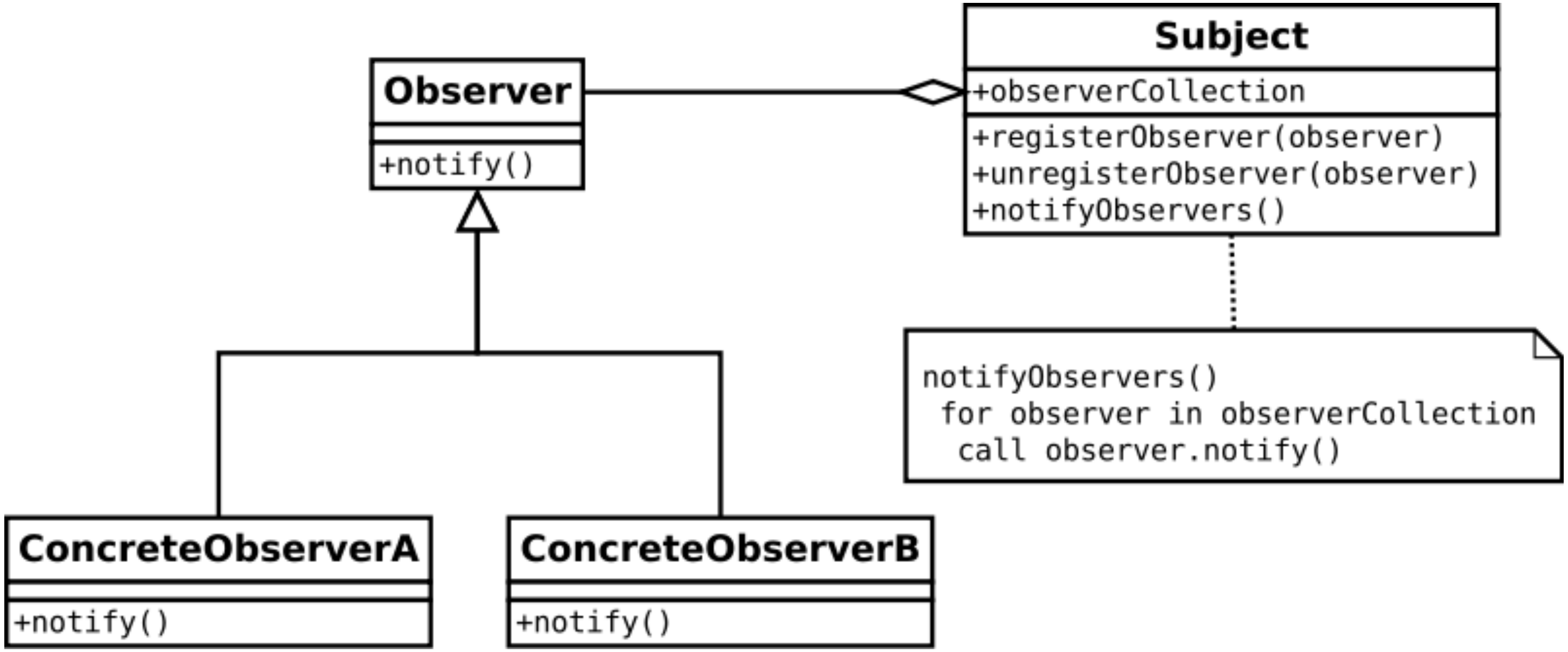
f) Facade



On design patterns

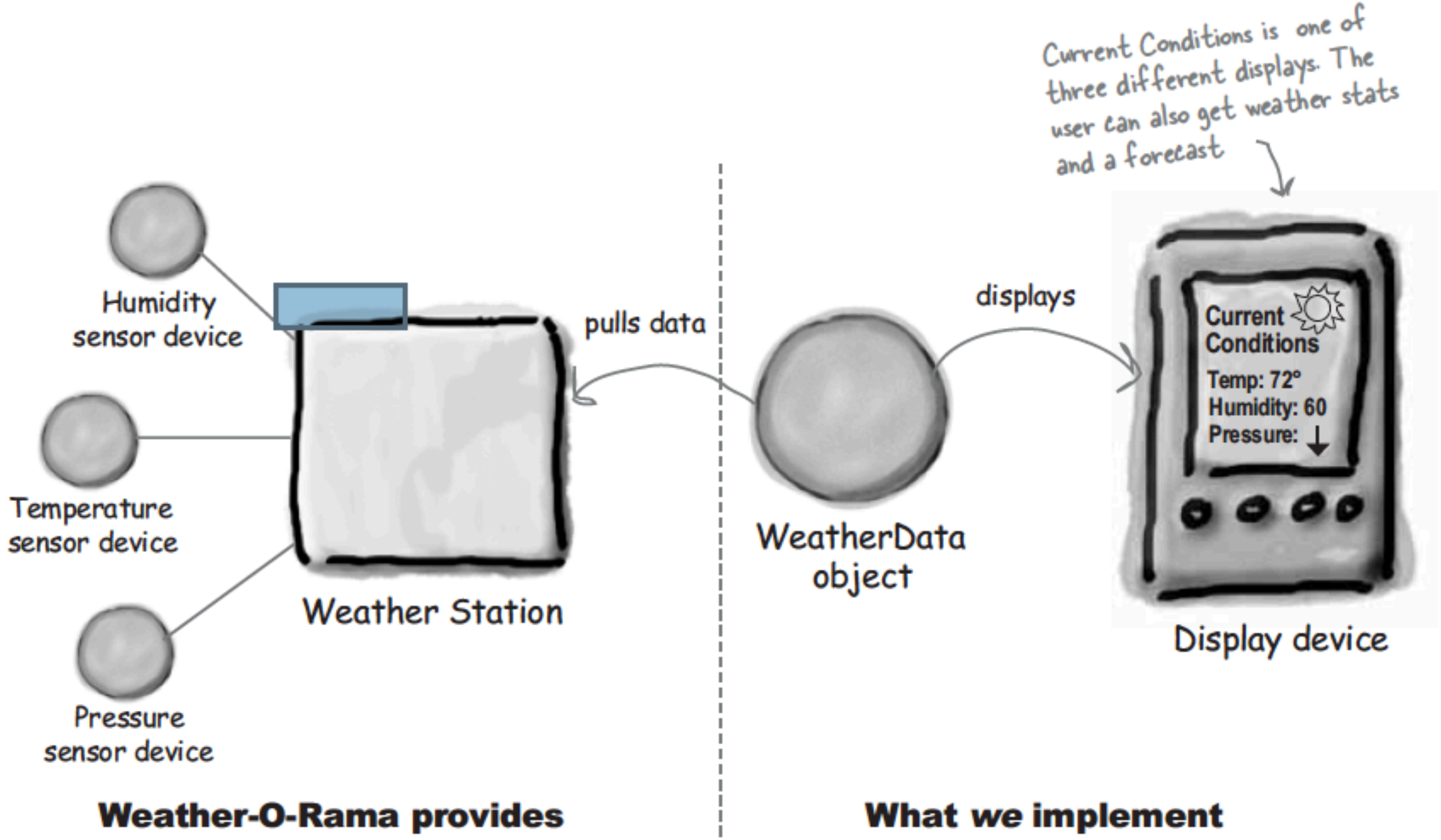
An object, called the *subject*, maintains a Collection of objects and notifies them of any change of its state, calling one of their methods. Which pattern is this?



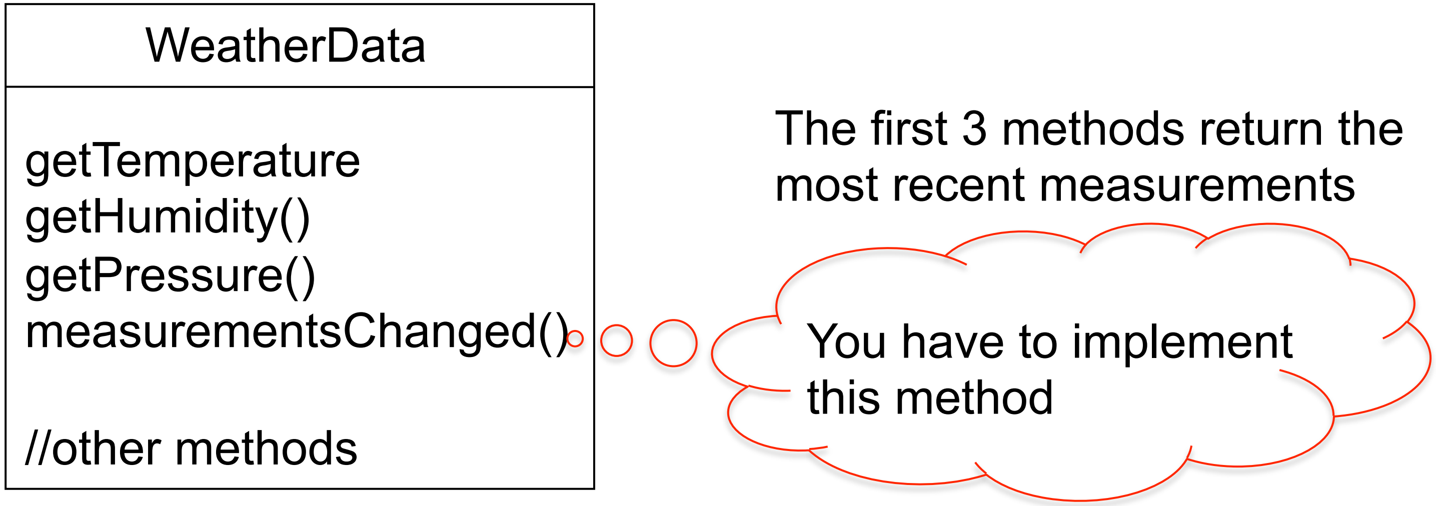


```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

Problema


Current Conditions is one of three different displays. The user can also get weather stats and a forecast
Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓
Display device
Weather-O-Rama provides
What we implement

WeatherData



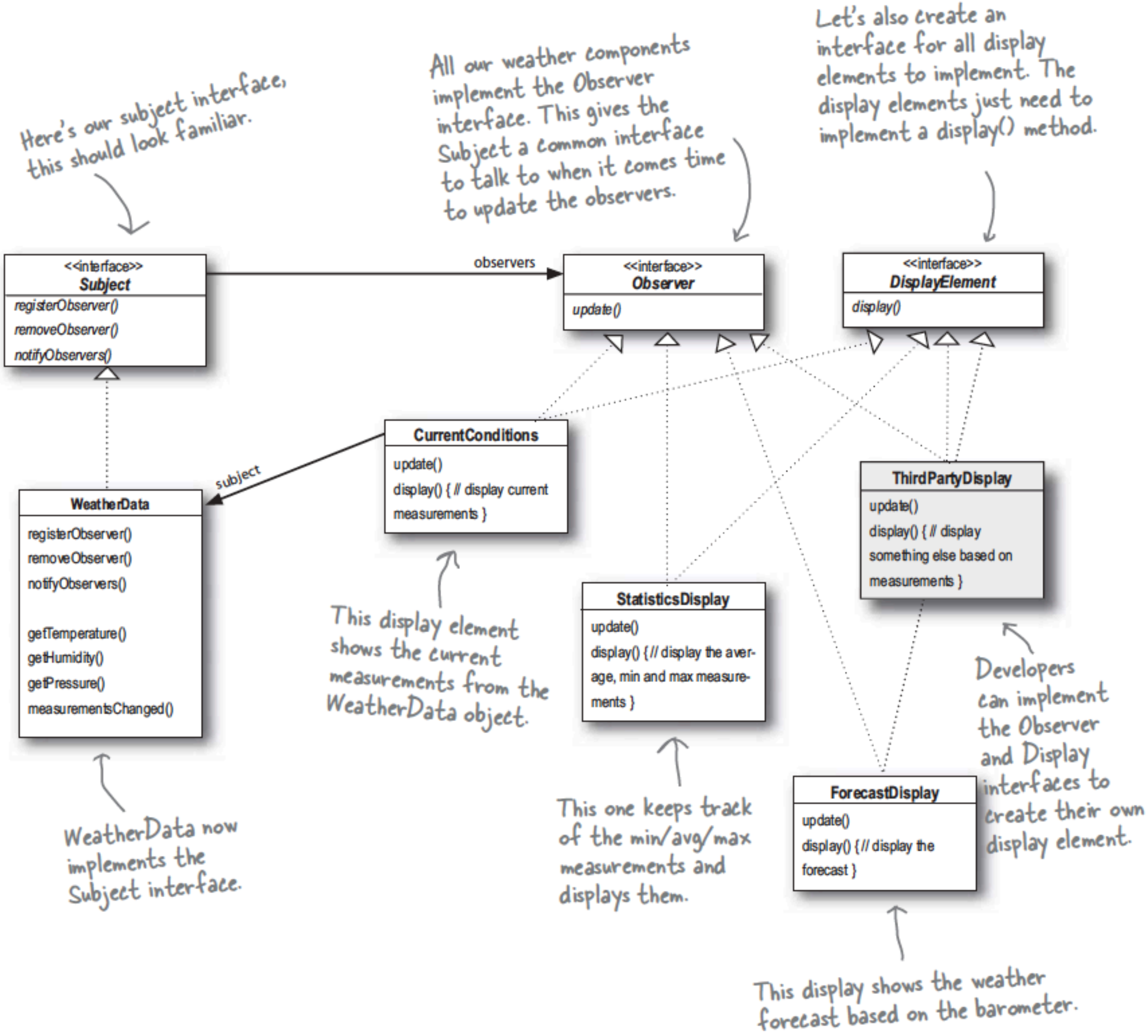
WeatherData

getTemperature
getHumidity()
getPressure()
measurementsChanged()

//other methods

The first 3 methods return the most recent measurements

You have to implement this method



Here's our subject interface, this should look familiar.

All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.

This display element shows the current measurements from the WeatherData object.

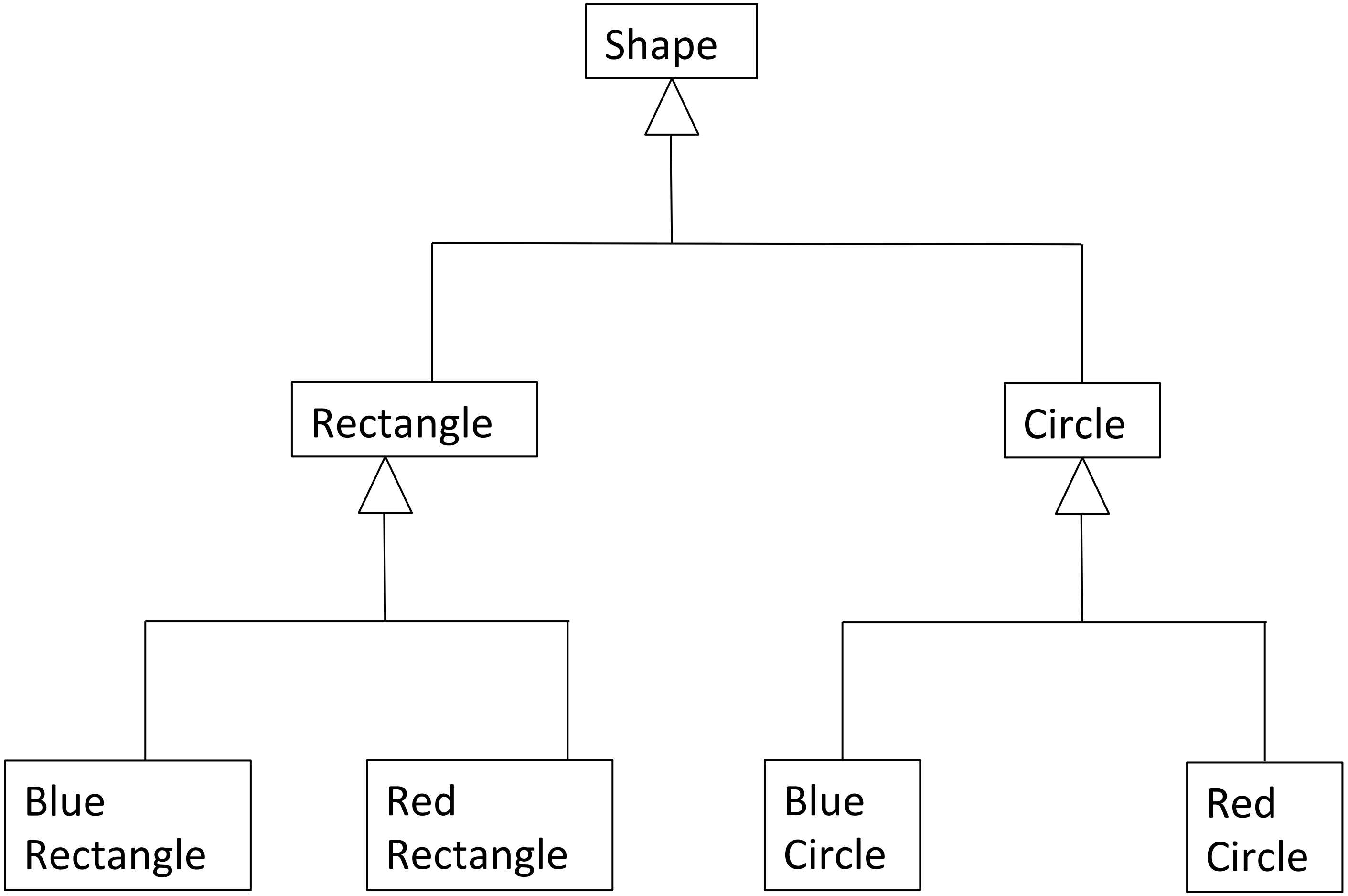
WeatherData now implements the Subject interface.

This one keeps track of the min/avg/max measurements and displays them.

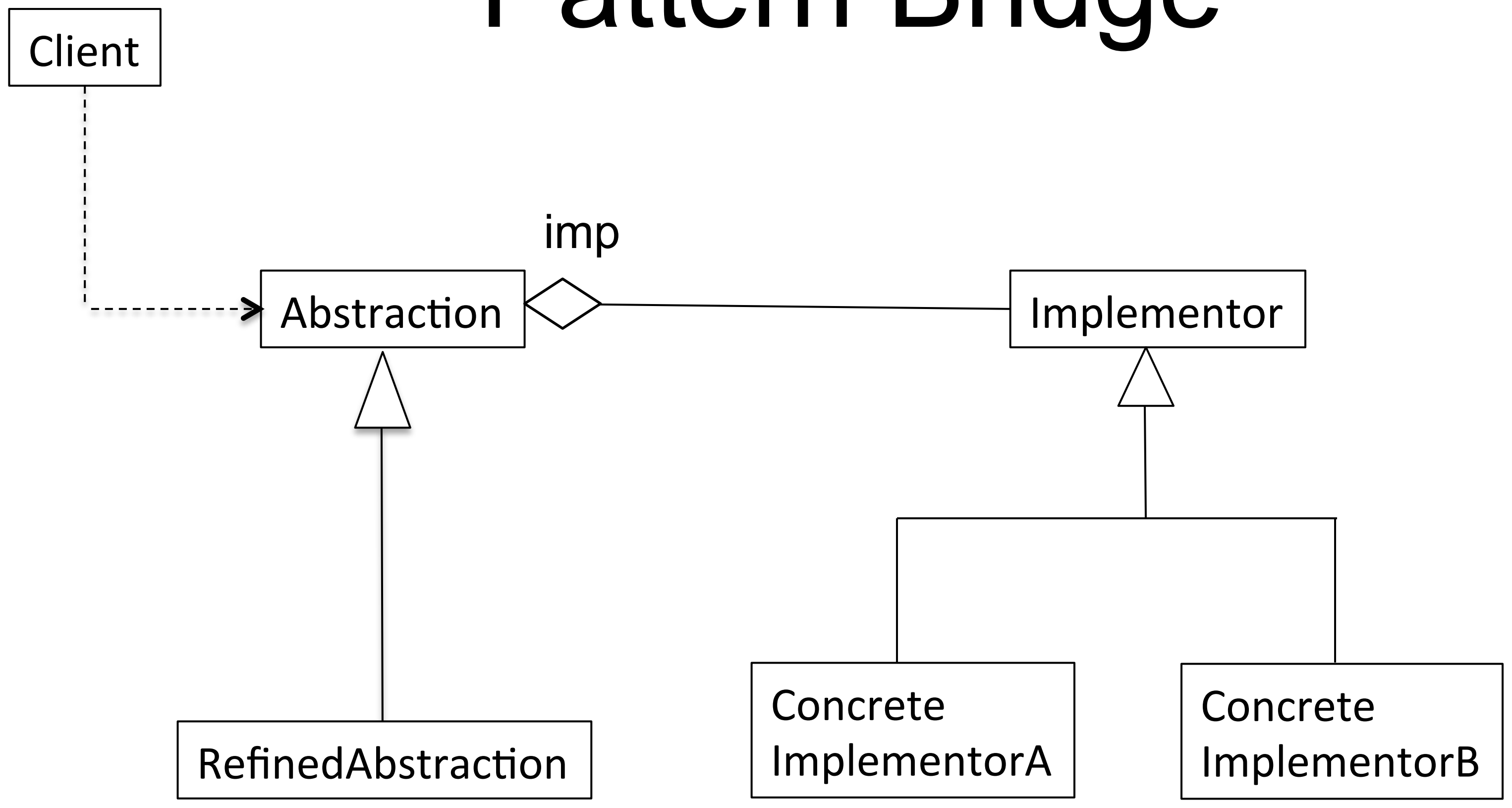
Developers can implement the Observer and Display interfaces to create their own display element.

This display shows the weather forecast based on the barometer.

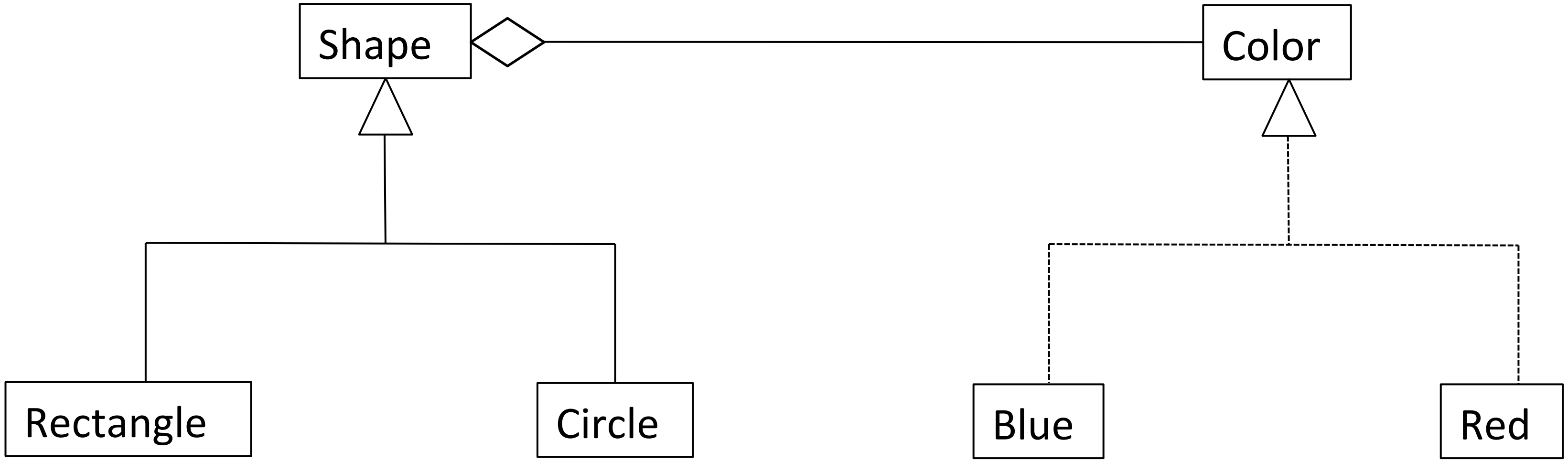
Problema


Shape
Rectangle
Circle
Blue Rectangle
Red Rectangle
Blue Circle
Red Circle

Pattern Bridge



Refactoring with Bridge



On design patterns

You are enhancing an existing application in a pizza shop. The price of the pizza depends on the options selected by the user. Each option carries a different additional price. There are a large number of options available (ex: extra cheese, type of crust, toppings and so on)

a)Abstract factory

b)Strategy

c)Composite

[✓ d)Decorator]

On design patterns

You are creating an application that simulates a technical support service provider. All requests are initially handled by front office support and are forwarded to higher levels as and when required

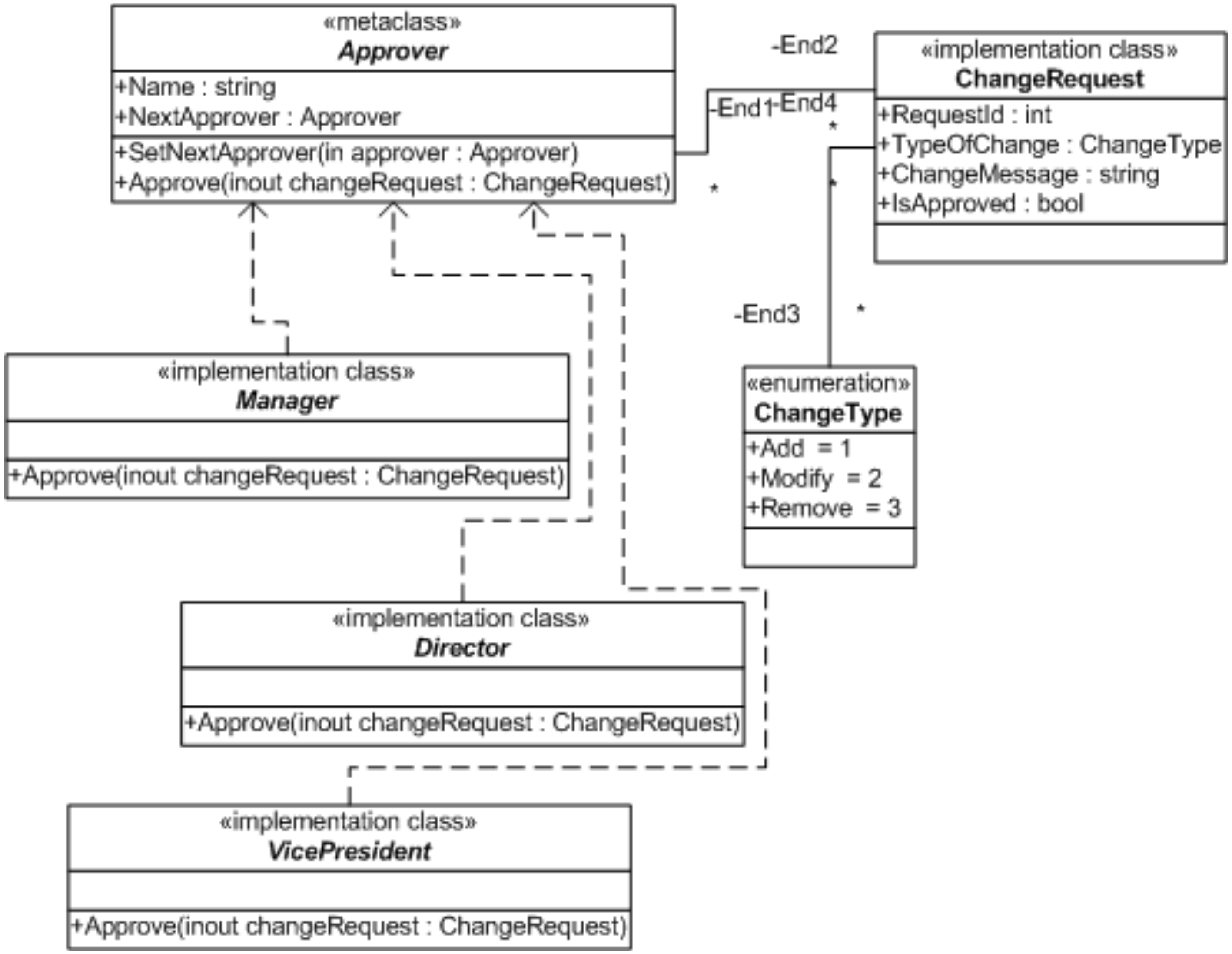
a)Strategy

[✓ b)Chain of responsibility]

c)Builder

d)State

DP: chain of responsibility



http://www.codeproject.com/Articles/9960/Refactoring-to-Patterns-Chain-Of-Responsibility-Pa

On design patterns

You are creating an application that needs functionality for logging. You need to implement a logger and log information into a file

a) [✓ Singleton]

b) Observer

c) Chain of responsibility

d) Abstract factory

On design patterns

Which design pattern would resolve incompatible interfaces or provide a stable interface to similar components with different interfaces?

a)Controller

b)Mediator

c)Visitor

[✓ d)Adapter]

On design patterns

Which design pattern would manage the reuse of objects for a type that is expensive to create or only a limited number of objects can be created?

a)Singleton

b)Object Pool

c)Memento

d)Connection pool

On design patterns

Which of the following statements are true about the Strategy pattern?

A. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it

B. The "context" part of the design deals with the behavior at an abstract level. The behavior is thus referred to as the "strategy" for accomplishing a task. The context is the "invariant" part of the design in that its behaviors do not change from situation to situation. Thus it can be encapsulated into a single object

C. The "strategy" part of the design captures the "variant" nature of the design where the particular actions are chosen dynamically in the run time. The strategy section consists of an abstract strategy class and a series of concrete strategy subclasses. Each subclass represents a possible specific action that could be taken when the strategy is executed by the context. The variant nature of the design has been abstracted and captured in a tree hierarchy.

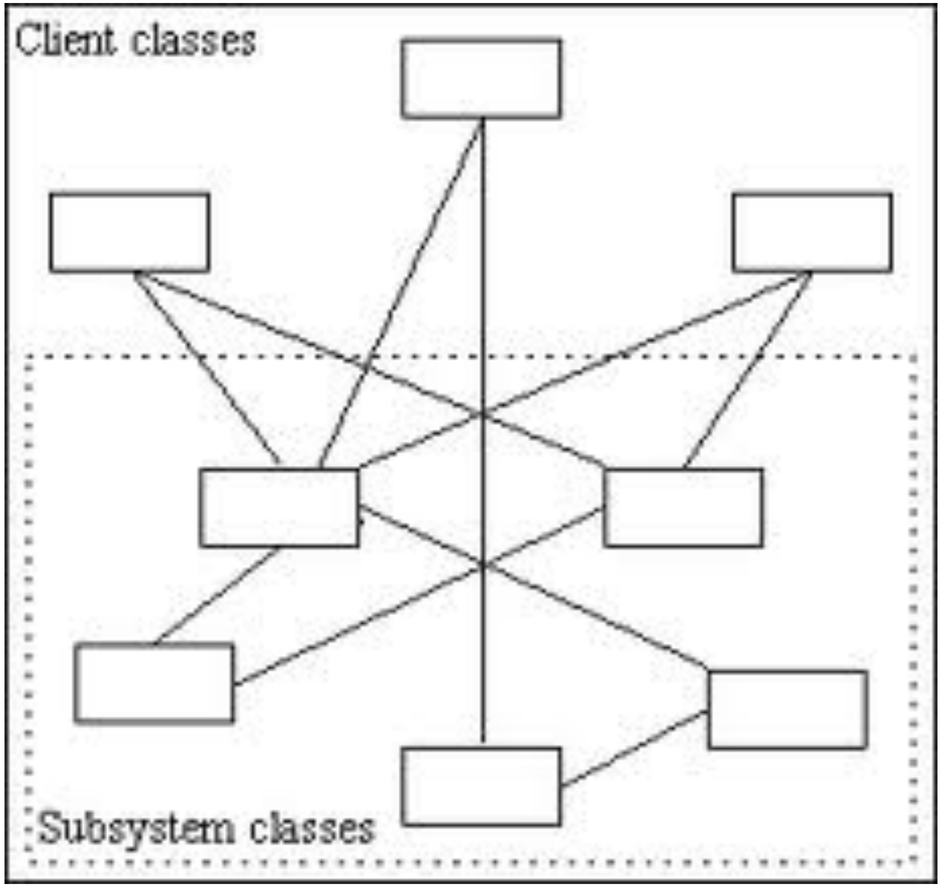
D. All of the above

E. B and C only

On design patterns

This is a problematic situation: client’s classes have too many relationships with subsystem’s classes.

Which pattern solves this problem?



On design patterns

Which pattern solves the problem shown in the figure, where "job" is assigned dynamically?

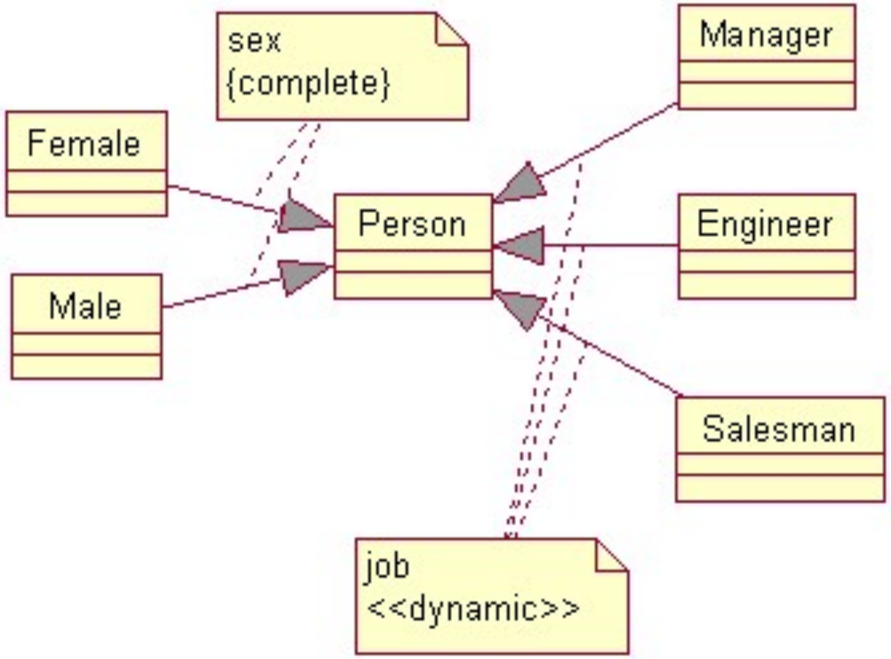
a) [✓ Strategy Pattern]

b) Composite Pattern

c) Adapter Pattern

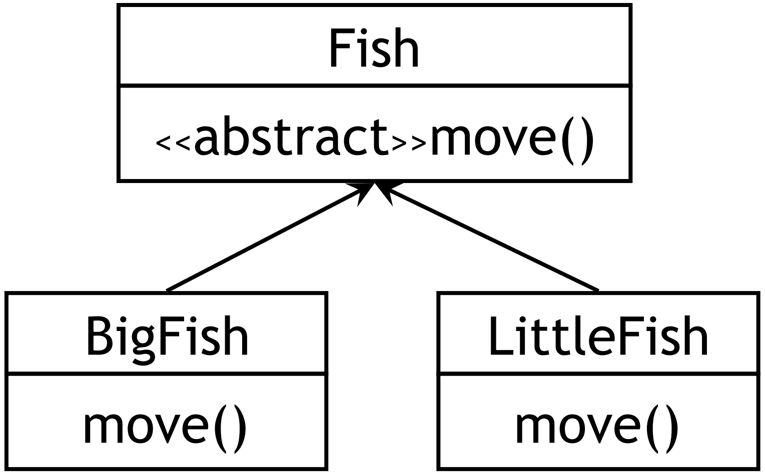
d) Bridge Pattern

e) Abstract Factory Pattern



Example: Big fish and little fish

- The scenario: “big fish” and “little fish” move around in an “ocean”
  - Fish move about randomly
  - A big fish can move to where a little fish is (and eat it)
  - A little fish will *not* move to where a big fish is



Problem: similar methods in subclasses

- we have a Fish class with two subclasses, BigFish and LittleFish
  - The two kinds of fish move the same way
  - To avoid code duplication, the move method ought to be in the superclass Fish
  - However, a LittleFish won't move to some locations where a BigFish will move
  - The test for whether it is OK to move really ought to be in the move method
- More generally, you want to have *almost* the same method in two or more sibling classes

Solution: Template method

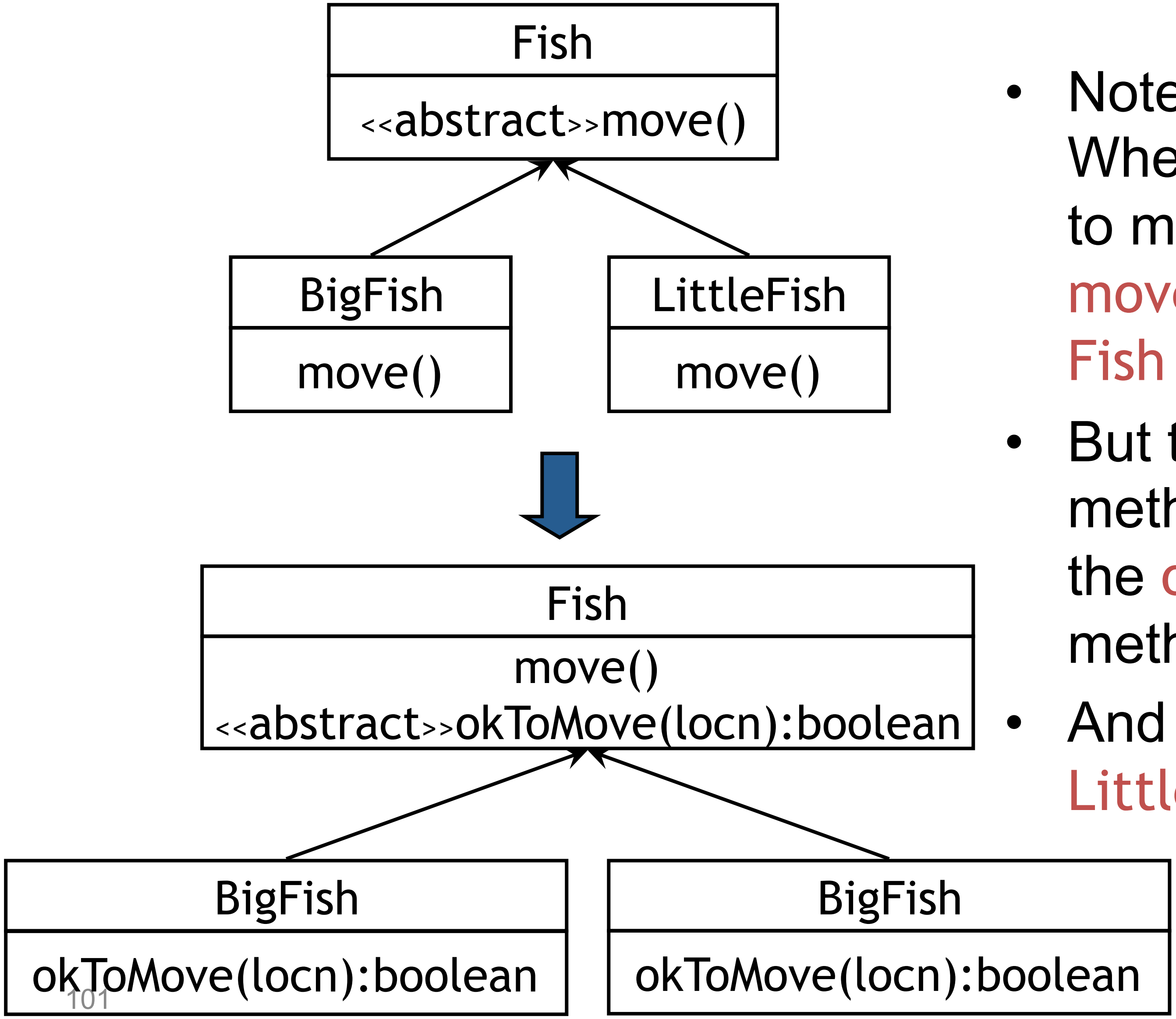
- The Design Pattern is called “Template Method”
- In the superclass, write the common method, but call an auxiliary method (such as okToMove) to perform the part of the logic that needs to differ
- Write the auxiliary method as an abstract method
  - This in turn requires that the superclass be abstract
- In each subclass, implement the auxiliary method according to the needs of that subclass
- When a subclass instance executes the common method, it will use its own auxiliary method as needed

The move() method

- General outline of the method:

  -
    ```
    public void move() {
        choose a random direction;           // same for both
        find the location in that direction; // same for both
        check if it’s ok to move there;      // different
        if it’s ok, make the move;           // same for both
    }
    ```
- To refactor:
  - Extract the check on whether it’s ok to move
  - In the Fish class, put the actual (template) move() method
  - Create an abstract okToMove() method in the Fish class
  - Implement okToMove() in each subclass

Refactoring

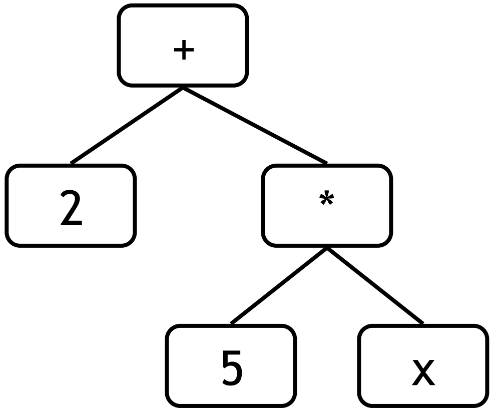


- Note how this works: When a BigFish tries to move, it uses the move() method in Fish
- But the move() method in Fish uses the okToMove(locn) method in BigFish
- And similarly for LittleFish

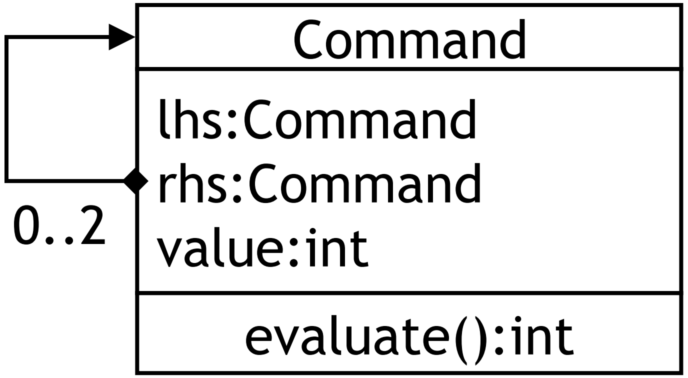
Example: evaluator

- A code to evalute expressions
- Expressions can be parsed into a tree structure

- You could walk the tree and, at each node, use a switch statement to do the right thing
- A better solution is a simple form of the Command dp



Tree for 2 + 5 * x

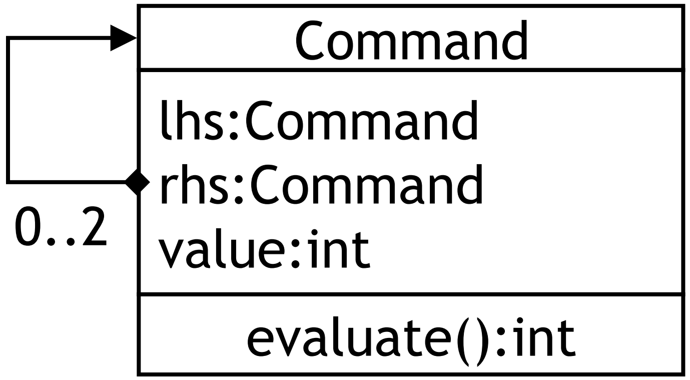


The Command Design Pattern

- Reasons for using the Command Design Pattern:
  - You want to control *if, when,* and *in what order* the commands are executed
  - You want to keep a log of commands executed
  - Popular reason: You want to manage *undo* and *redo* operations
- Possible class organization (from GoF):
  - AbstractCommand with doIt() and undoIt() methods
  - ***ConcreteCommand*** subclasses of AbstractCommand
  - Invoker is a class that creates ***ConcreteCommand*** objects if it needs to invoke a command
  - CommandManager to decide what, when, and how to execute and undo commands

Using the Command pattern

-
  ```
  class Add extends Command {
      int evaluate( ) {
          int v1 = lhs.evaluate().value;
          int v2 = rhs.evaluate().value;
          value = v1 + v2;
          return value;
      }
  }
  ```



Command

lhs:Command
rhs:Command
value:int

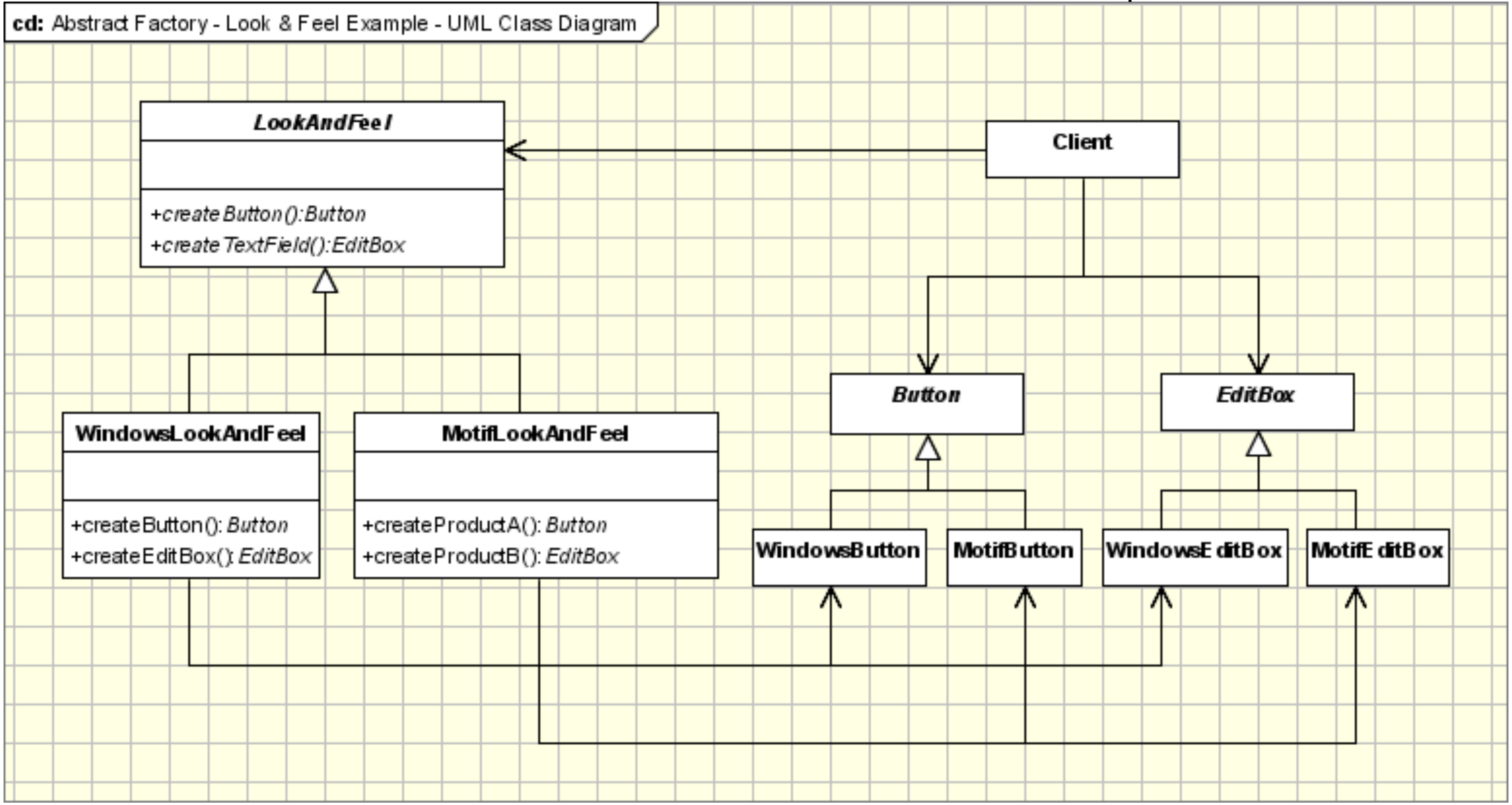
evaluate():int

0..2

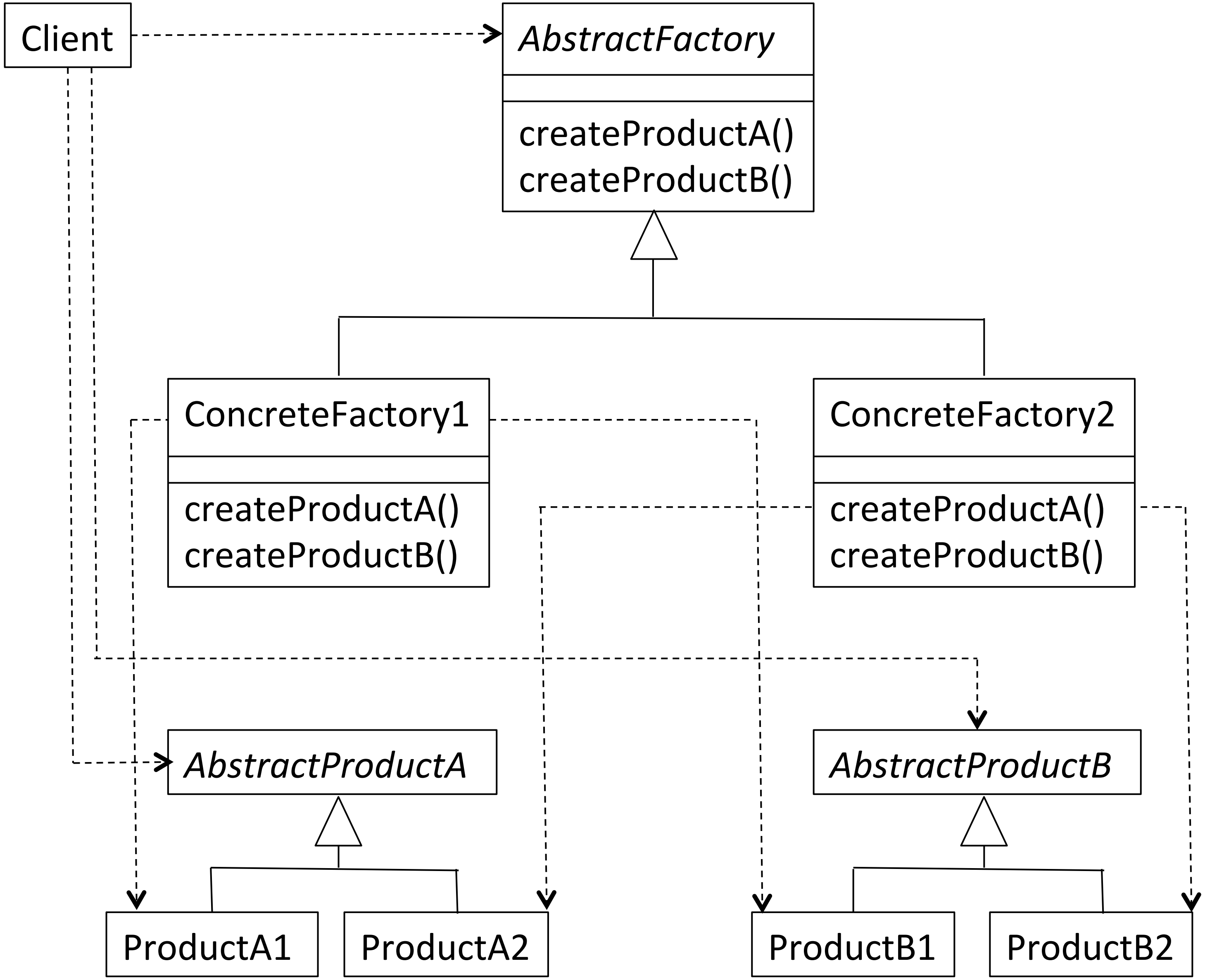
- To evaluate the entire tree, evaluate the root node
- This is just a rough description; there are a lot of other details to consider
  - Some operands are unary
  - You have to look up the values of variables
  - Etc.

Problem

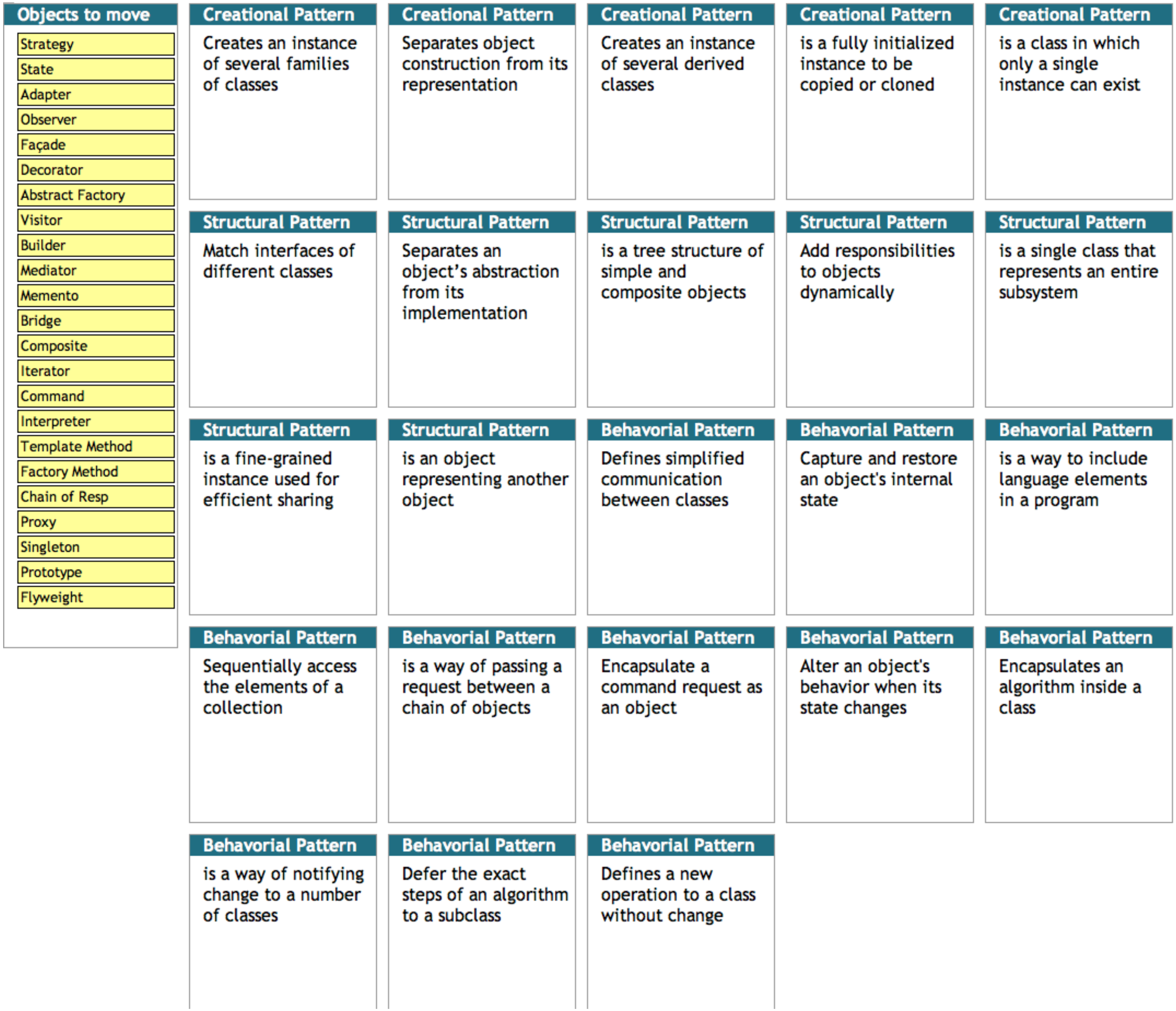
A GUI framework should support several look and feel themes, such as Motif and Windows look. Each style defines different looks and behaviors for each type of controls: Buttons and Edit boxes. In order to avoid hardcoding it for each type of control we define an abstract class LookAndFeel. This class will instantiate, depending on a configuration parameter in the application, one between WindowsLookAndFeel or MotifLookAndFeel. Each request for a new object will be delegated to the instances which will return the controls with the specific flavor


cd: Abstract Factory - Look & Feel Example - UML Class Diagram
LookAndFeel
+createButton():Button
+createTextField():EditBox
Client
WindowsLookAndFeel
+createButton(): Button
+createEditBox(): EditBox
MotifLookAndFeel
+createProductA(): Button
+createProductB(): EditBox
Button
EditBox
WindowsButton
MotifButton
WindowsEditBox
MotifEditBox

Abstract factory



Riconoscere i pattern GoF



Objects to move

- Strategy
- State
- Adapter
- Observer
- Façade
- Decorator
- Abstract Factory
- Visitor
- Builder
- Mediator
- Memento
- Bridge
- Composite
- Iterator
- Command
- Interpreter
- Template Method
- Factory Method
- Chain of Resp
- Proxy
- Singleton
- Prototype
- Flyweight

Creational Pattern

Creates an instance of several families of classes

Creational Pattern

Separates object construction from its representation

Creational Pattern

Creates an instance of several derived classes

Creational Pattern

is a fully initialized instance to be copied or cloned

Creational Pattern

is a class in which only a single instance can exist

Structural Pattern

Match interfaces of different classes

Structural Pattern

Separates an object's abstraction from its implementation

Structural Pattern

is a tree structure of simple and composite objects

Structural Pattern

Add responsibilities to objects dynamically

Structural Pattern

is a single class that represents an entire subsystem

Structural Pattern

is a fine-grained instance used for efficient sharing

Structural Pattern

is an object representing another object

Behavorial Pattern

Defines simplified communication between classes

Behavorial Pattern

Capture and restore an object's internal state

Behavorial Pattern

is a way to include language elements in a program

Behavorial Pattern

Sequentially access the elements of a collection

Behavorial Pattern

is a way of passing a request between a chain of objects

Behavorial Pattern

Encapsulate a command request as an object

Behavorial Pattern

Alter an object's behavior when its state changes

Behavorial Pattern

Encapsulates an algorithm inside a class

Behavorial Pattern

is a way of notifying change to a number of classes

Behavorial Pattern

Defer the exact steps of an algorithm to a subclass

Behavorial Pattern

Defines a new operation to a class without change

Creational Pattern

Creates an instance of several families of classes

Abstract Factory

Creational Pattern

Separates object construction from its representation

Builder

Creational Pattern

Creates an instance of several derived classes

Factory Method

Creational Pattern

is a fully initialized instance to be copied or cloned

Prototype

Creational Pattern

is a class in which only a single instance can exist

Singleton

Structural Pattern

Match interfaces of different classes

Adapter

Structural Pattern

Separates an object's abstraction from its implementation

Bridge

Structural Pattern

is a tree structure of simple and composite objects

Composite

Structural Pattern

Add responsibilities to objects dynamically

Decorator

Structural Pattern

is a single class that represents an entire subsystem

Façade

Structural Pattern

is a fine-grained instance used for efficient sharing

Flyweight

Structural Pattern

is an object representing another object

Proxy

Behavorial Pattern

Defines simplified communication between classes

Mediator

Behavorial Pattern

Capture and restore an object's internal state

Memento

Behavorial Pattern

is a way to include language elements in a program

Interpreter

Behavorial Pattern

Sequentially access the elements of a collection

Iterator

Behavorial Pattern

is a way of passing a request between a chain of objects

Chain of Resp

Behavorial Pattern

Encapsulate a command request as an object

Command

Behavorial Pattern

Alter an object's behavior when its state changes

State

Behavorial Pattern

Encapsulates an algorithm inside a class

Strategy

Behavorial Pattern

is a way of notifying change to a number of classes

Observer

Behavorial Pattern

Defer the exact steps of an algorithm to a subclass

Template Method

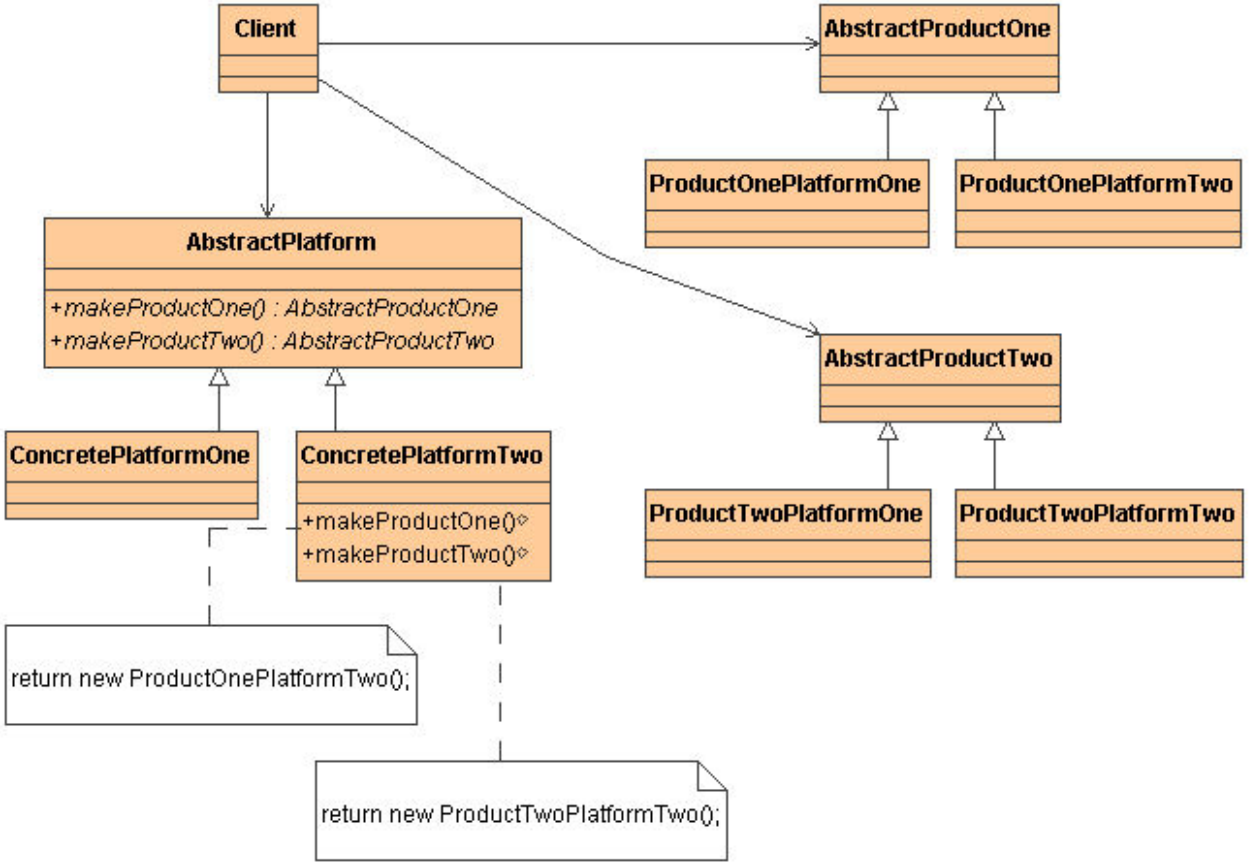
Behavorial Pattern

Defines a new operation to a class without change

Visitor

Which pattern?

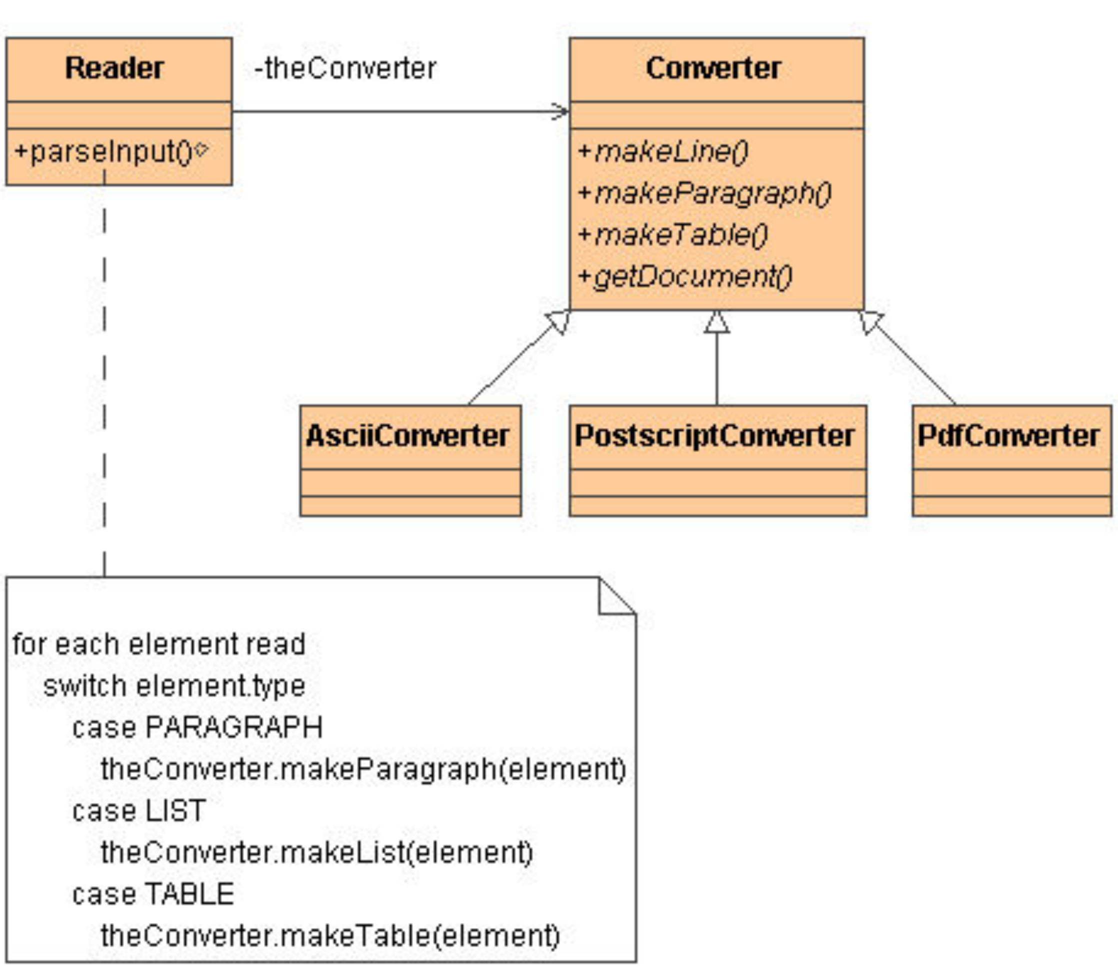
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Provide an interface for creating families of related objects, without specifying concrete classes

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10.Facade
11.Flyweight
12.Proxy
13.Chain of Responsibility
14.Command
15.Interpreter
16.Iterator
17.Mediator
18.Memento
19.Observer
20.State
21.Strategy
22.Template Method
23.Visitor

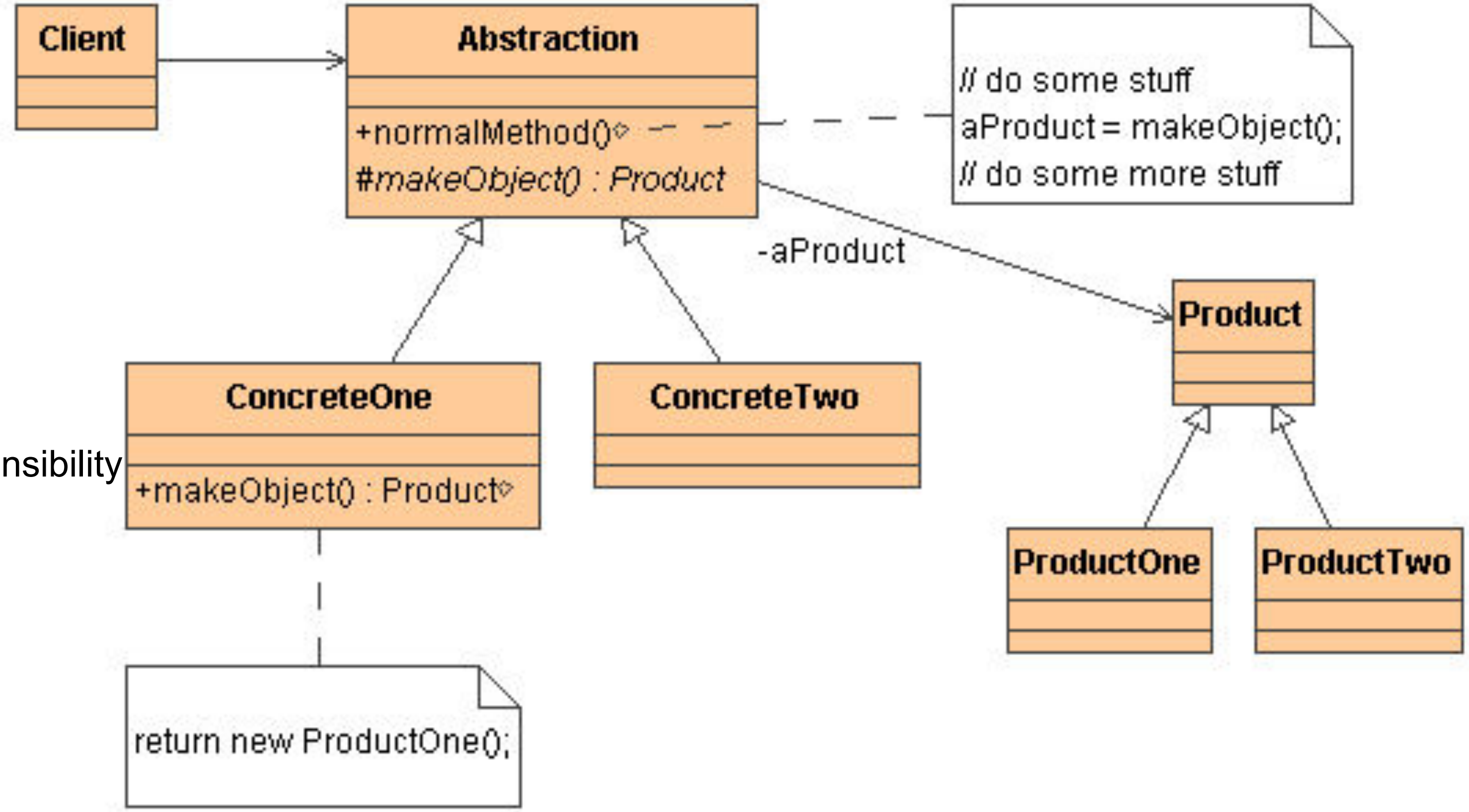


```
for each element read
  switch element.type
    case PARAGRAPH
      theConverter.makeParagraph(element)
    case LIST
      theConverter.makeList(element)
    case TABLE
      theConverter.makeTable(element)
```

Separate the construction of a complex object from its representation so that the same construction process can create different representations. One common input, many possible outputs

Which pattern?

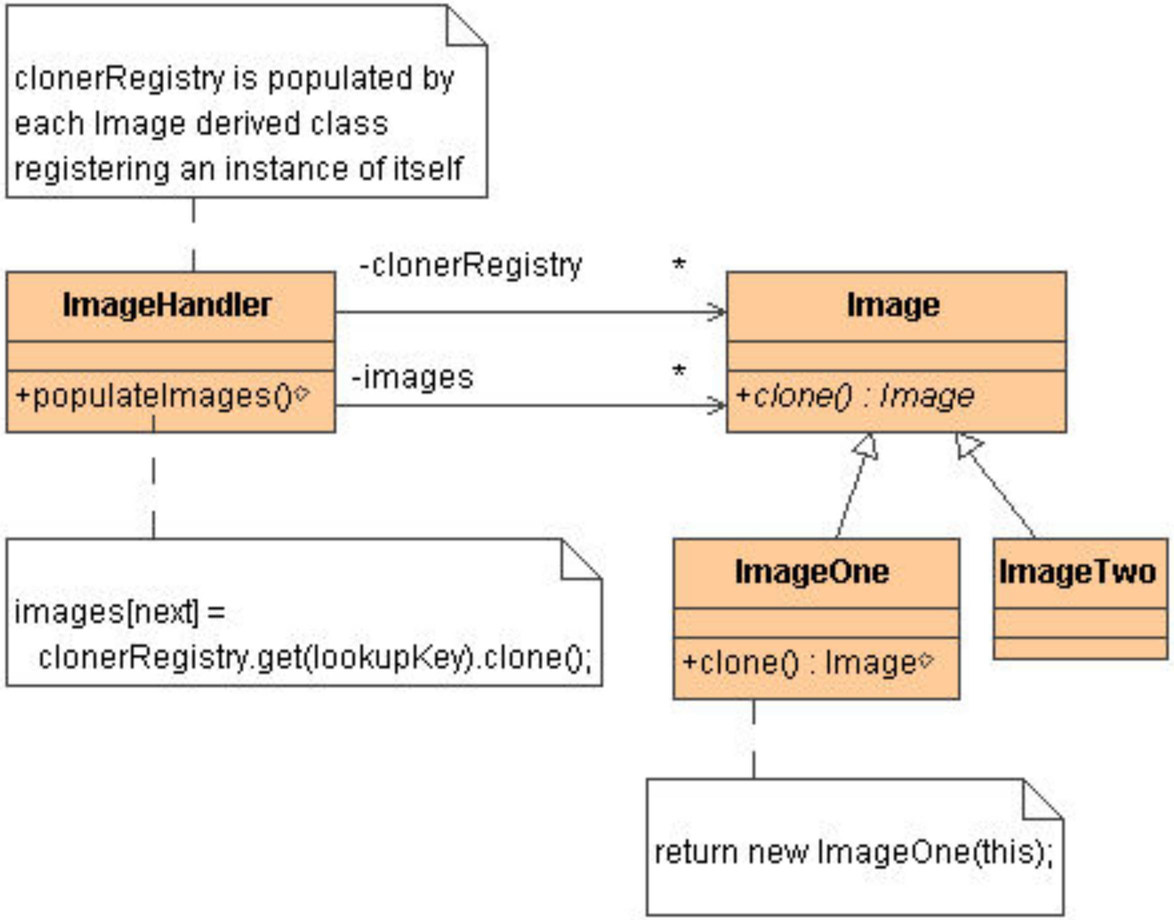
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



defines an interface for creating objects, but lets subclasses decide which classes to instantiate

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

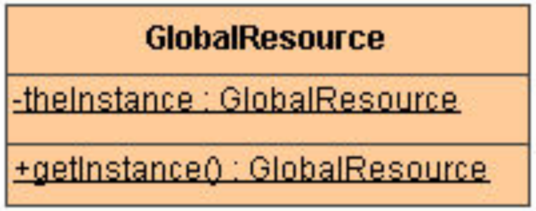


clonerRegistry is populated by each Image derived class registering an instance of itself
ImageHandler
+populateImages()
-clonerRegistry
*
-images
*
Image
+clone() : Image
ImageOne
+clone() : Image
ImageTwo
images[next] = clonerRegistry.get(lookupKey).clone();
return new ImageOne(this);

Specify the kinds of objects to create using a cloneable instance and create new objects by copying this instance. Indirect creation through delegation; cloning; the "new" statement considered harmful

Which pattern?

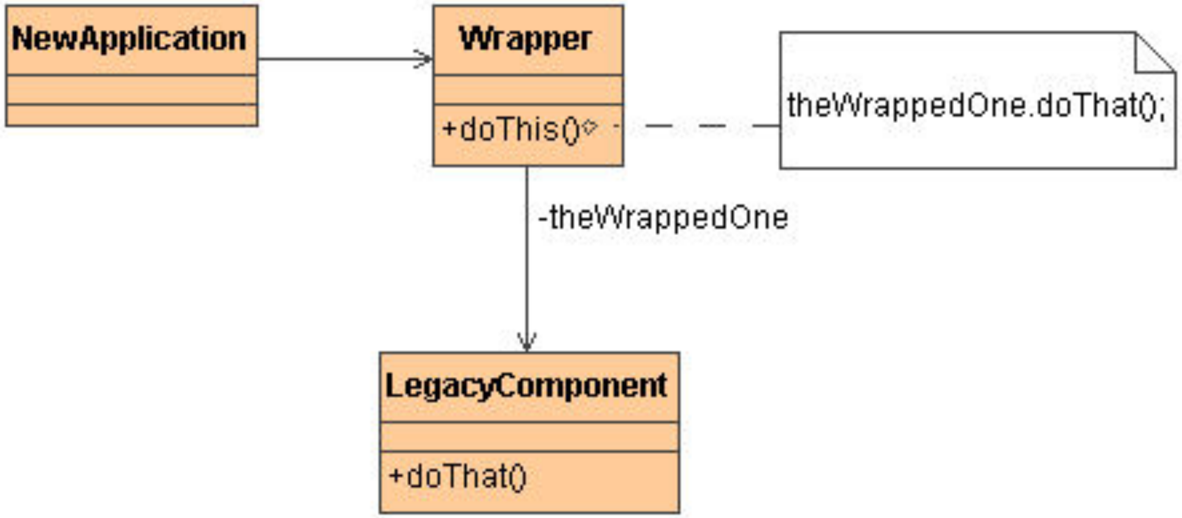
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

Ensures that a class has only one instance and provides a global point of access to that instance

Which pattern?

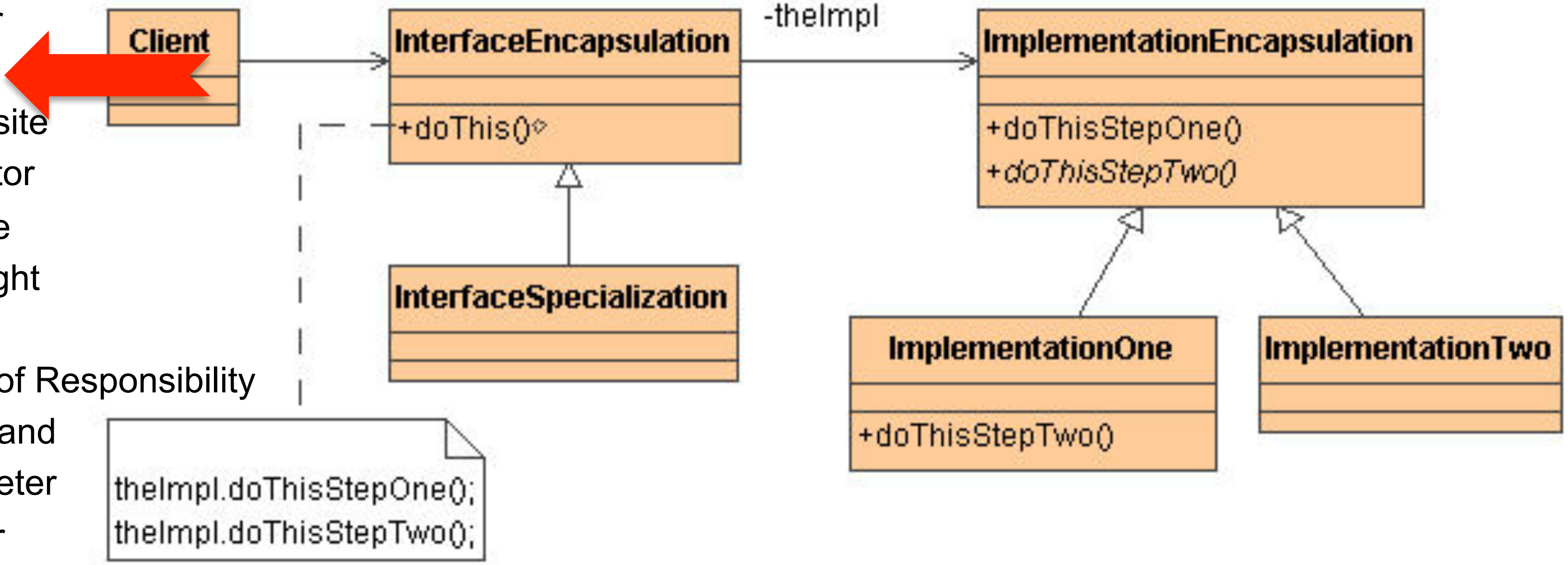
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients

Which pattern?

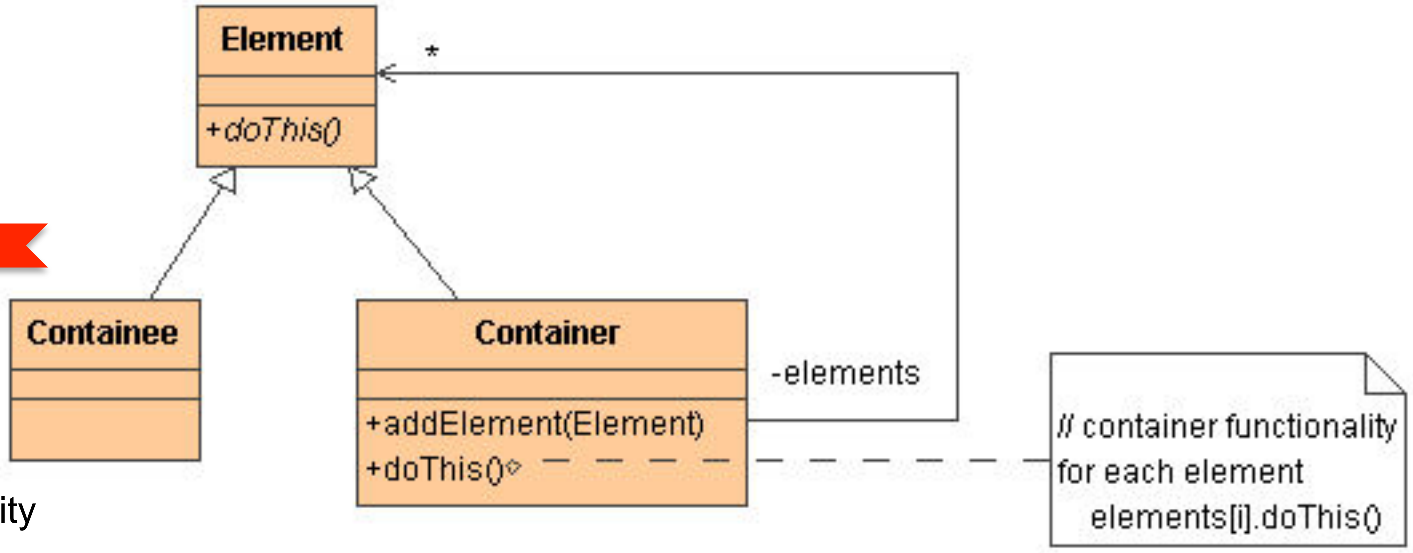
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



decouples an abstraction from its implementation, so that the two can vary independently

Which pattern?

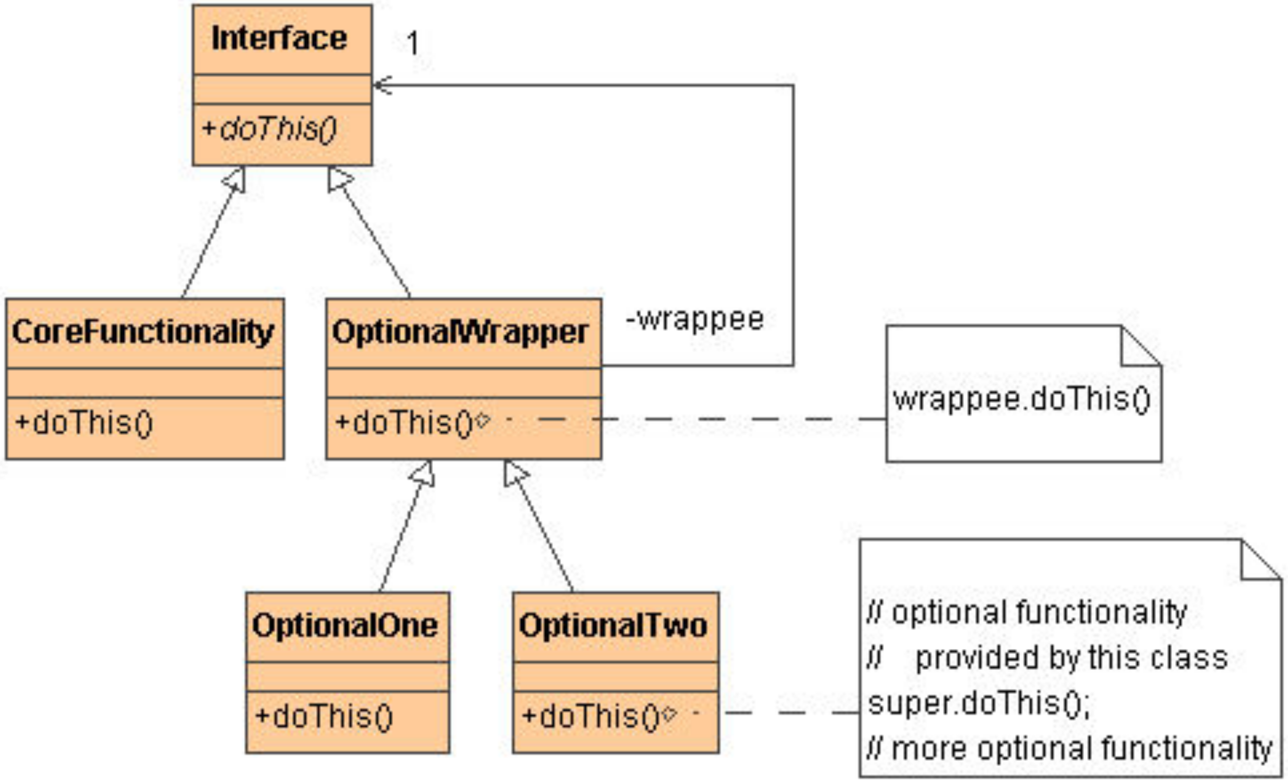
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



composes objects into tree structures and lets clients treat individual objects and compositions uniformly

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

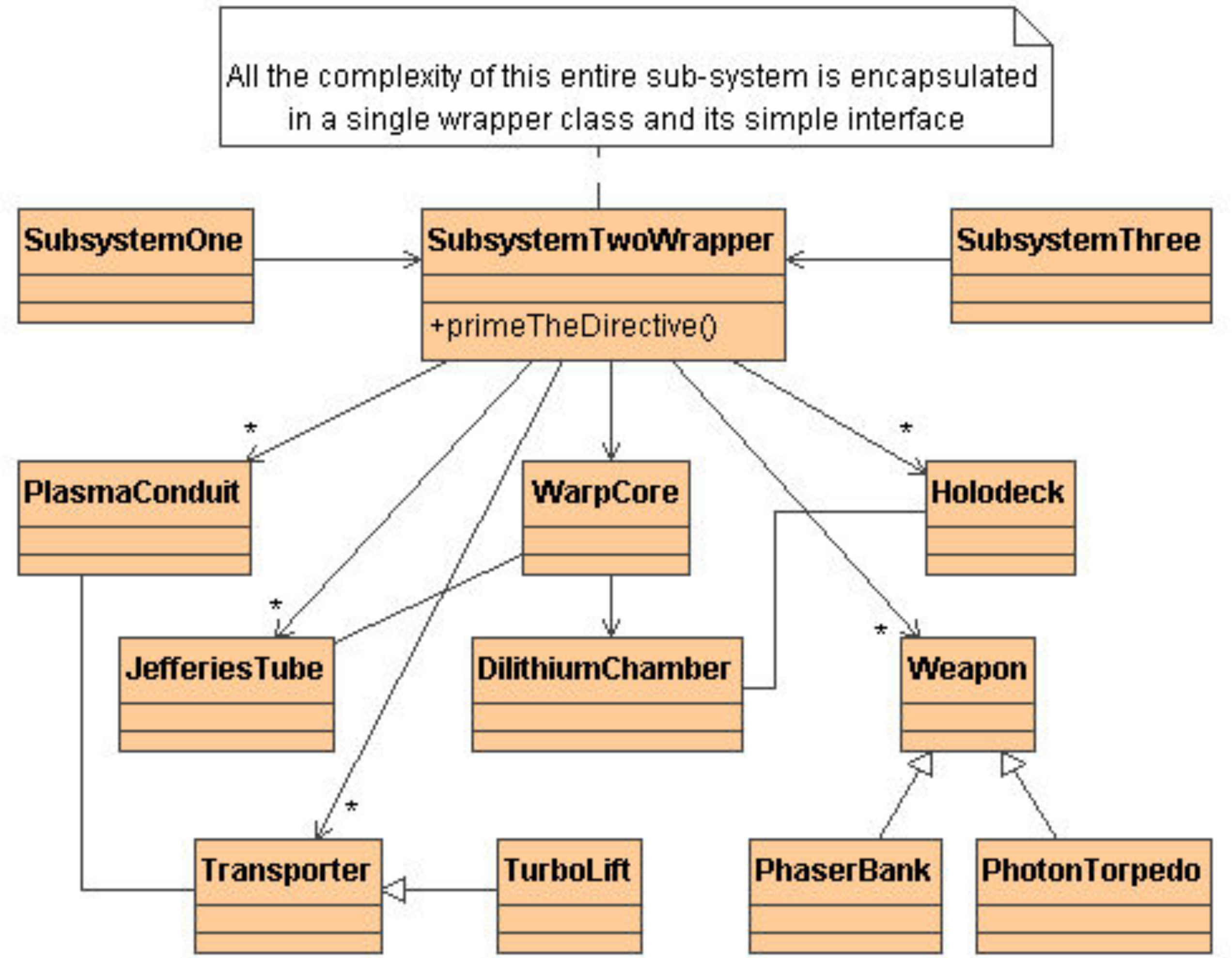


Attach additional responsibilities to an object dynamically. Provide a flexible alternative to subclassing for extending functionality. Recursive composition;

- 1-to-1 "has a" up the "is a" hierarchy
- a single core object wrapped by possibly many optional objects
- user configuration of optional features to an existing class

Which pattern?

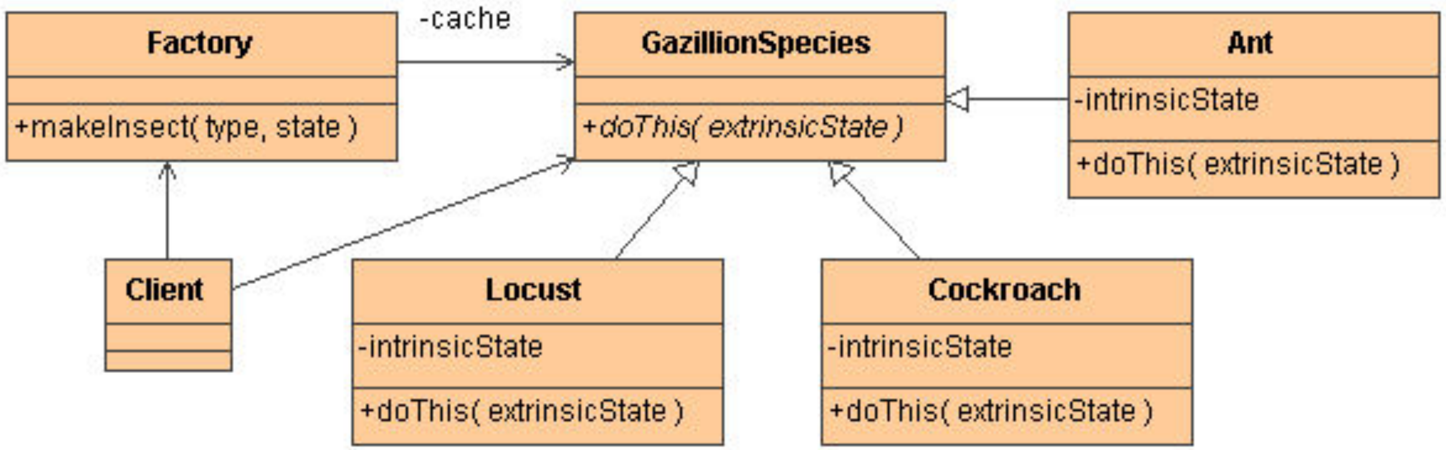
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



defines a unified, higher level interface to a subsystem that makes it easier to use

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Factory

+makeInsect(type, state)

-cache

GazillionSpecies

+*doThis(extrinsicState)*

Ant

-intrinsicState

+doThis(extrinsicState)

Client

Locust

-intrinsicState

+doThis(extrinsicState)

Cockroach

-intrinsicState

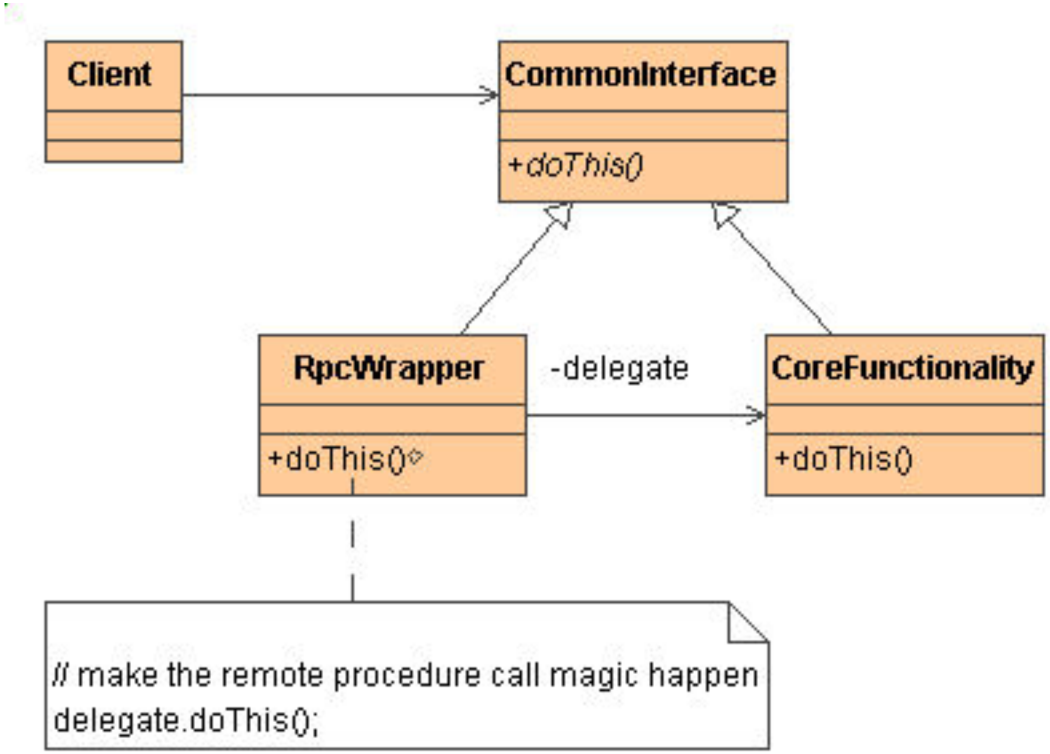
+doThis(extrinsicState)

Use sharing to support large numbers of fine-grained objects efficiently.

- how to design dozens of small objects that incur minimal overhead
- instance-independent state stays in the class
- instance-dependent state is supplied by the customer
- a factory facilitates object reuse

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

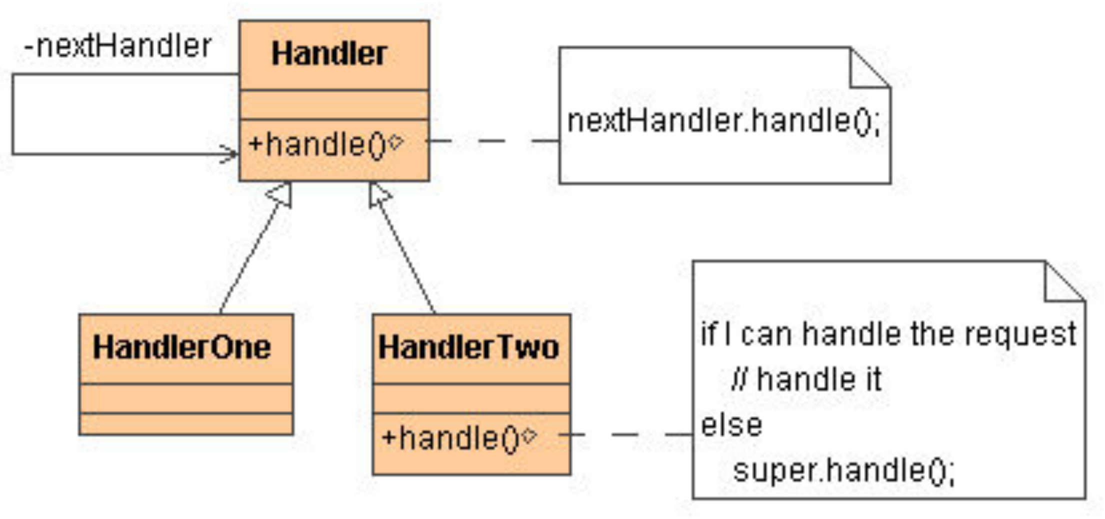


```
// make the remote procedure call magic happen
delegate.doThis();
```

provides a surrogate or place holder to provide access to an object

Which pattern?

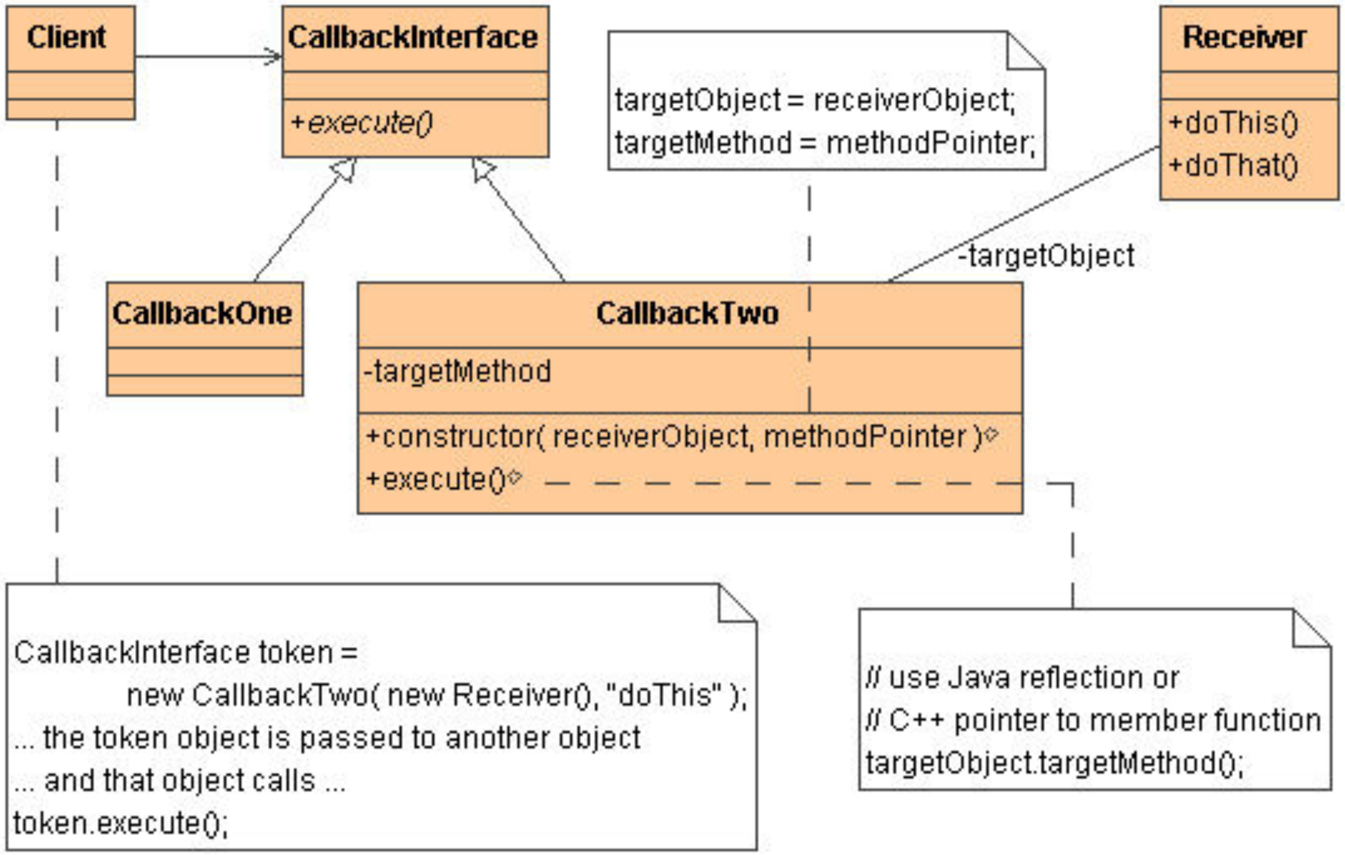
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request.

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10.Facade
11.Flyweight
12.Proxy
13.Chain of Responsibility
14.Command
15.Interpreter
16.Iterator
17.Mediator
18.Memento
19.Observer
20.State
21.Strategy
22.Template Method
23.Visitor



```
targetObject = receiverObject;
targetMethod = methodPointer;
```

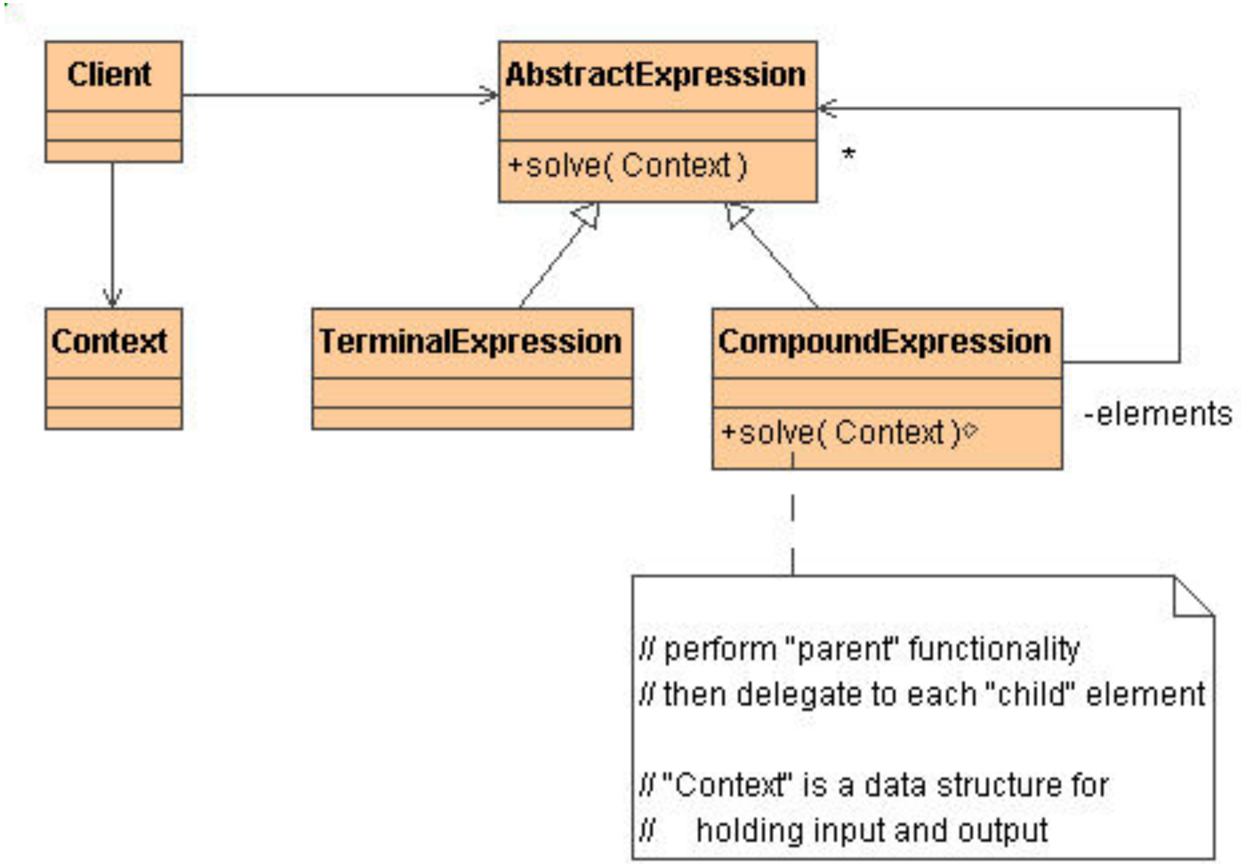
```
CallbackInterface token =
          new CallbackTwo( new Receiver(), "doThis" );
... the token object is passed to another object
... and that object calls ...
token.execute();
```

```
// use Java reflection or
// C++ pointer to member function
targetObject.targetMethod();
```

allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



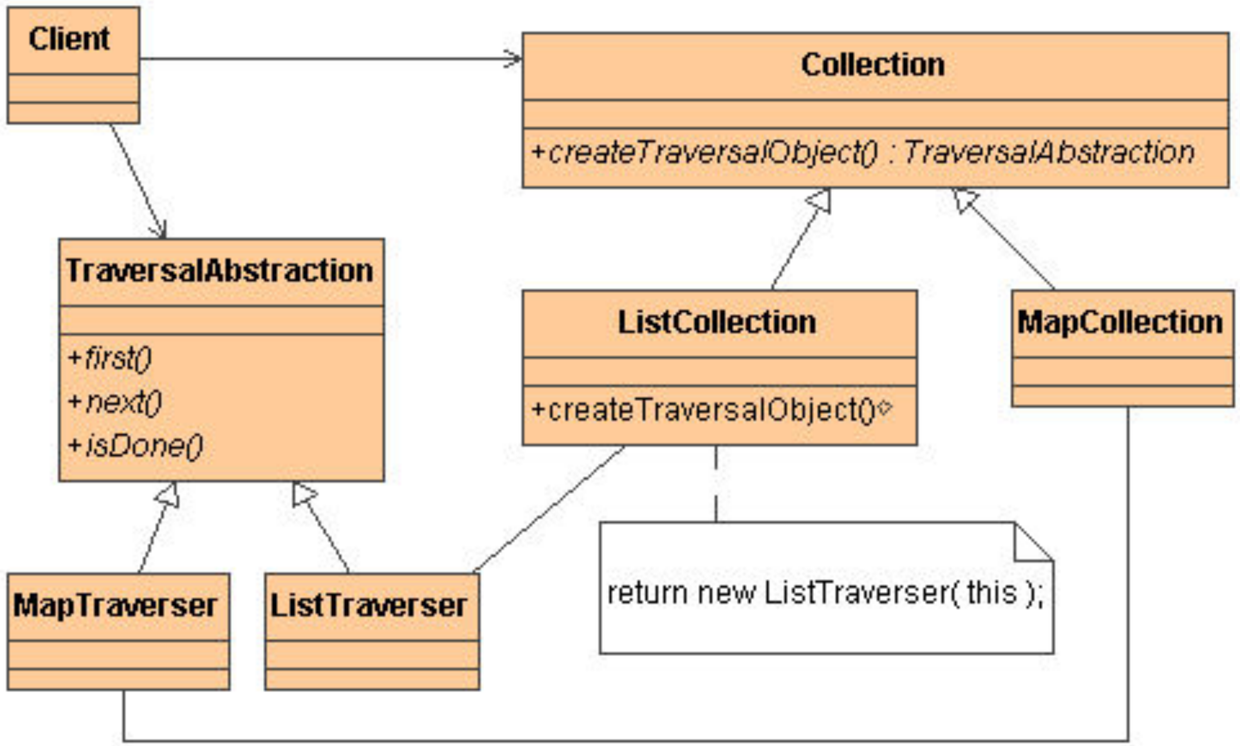
// perform "parent" functionality
// then delegate to each "child" element

// "Context" is a data structure for
//   holding input and output

Given a language, define a representation for its grammar along with a processor that uses the representation to parse sentences in the language

Which pattern?

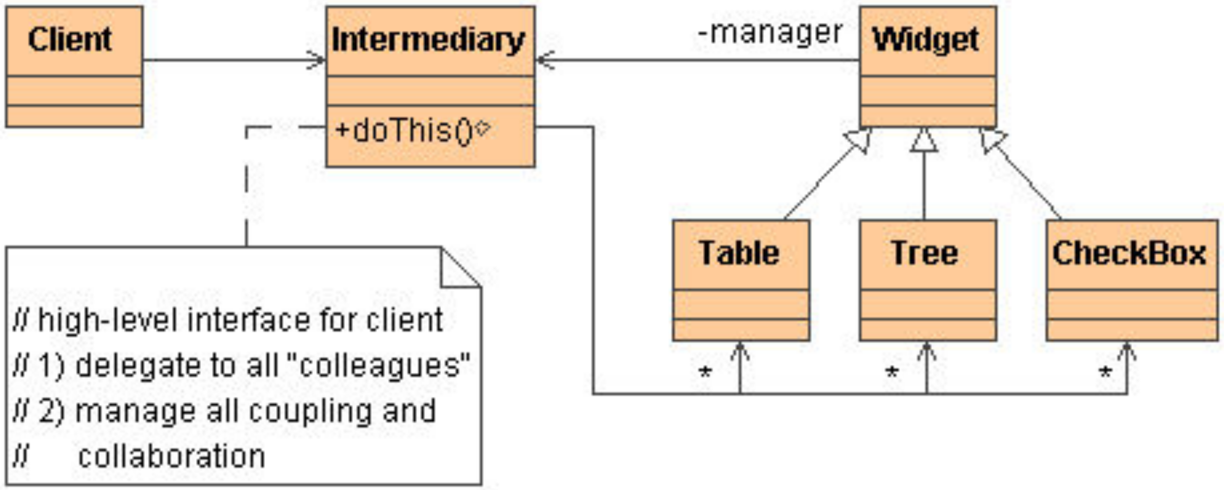
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

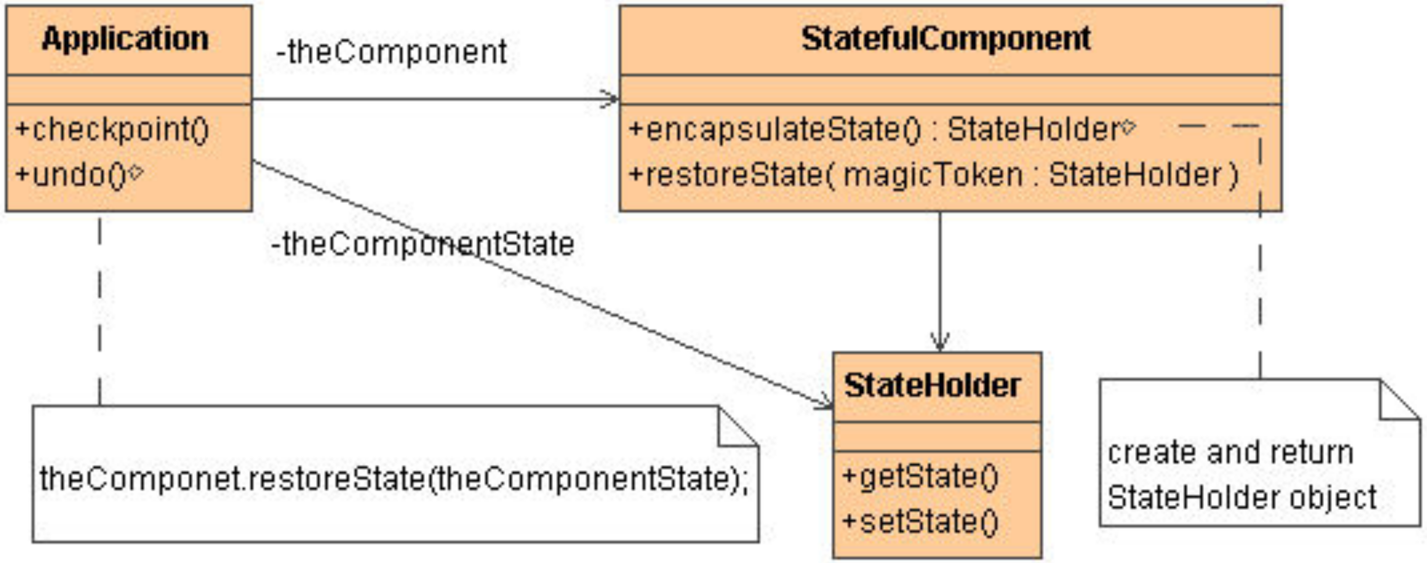
Which pattern?


Client
Intermediary
+doThis()
-manager
Widget
Table
Tree
CheckBox
*
*
*
// high-level interface for client
// 1) delegate to all "colleagues"
// 2) manage all coupling and
//   collaboration

defines an intermediary object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by avoiding colleagues communicate directly with each other

Which pattern?

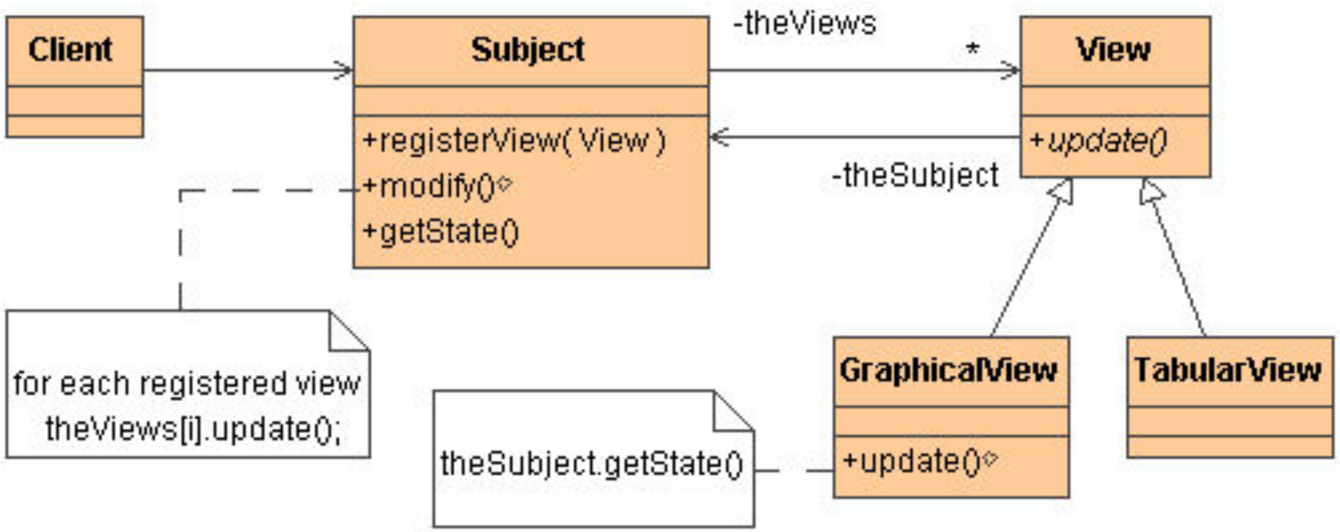
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
Undo, rollback: a magic cookie that encapsulates a "check point" capability

Which pattern?

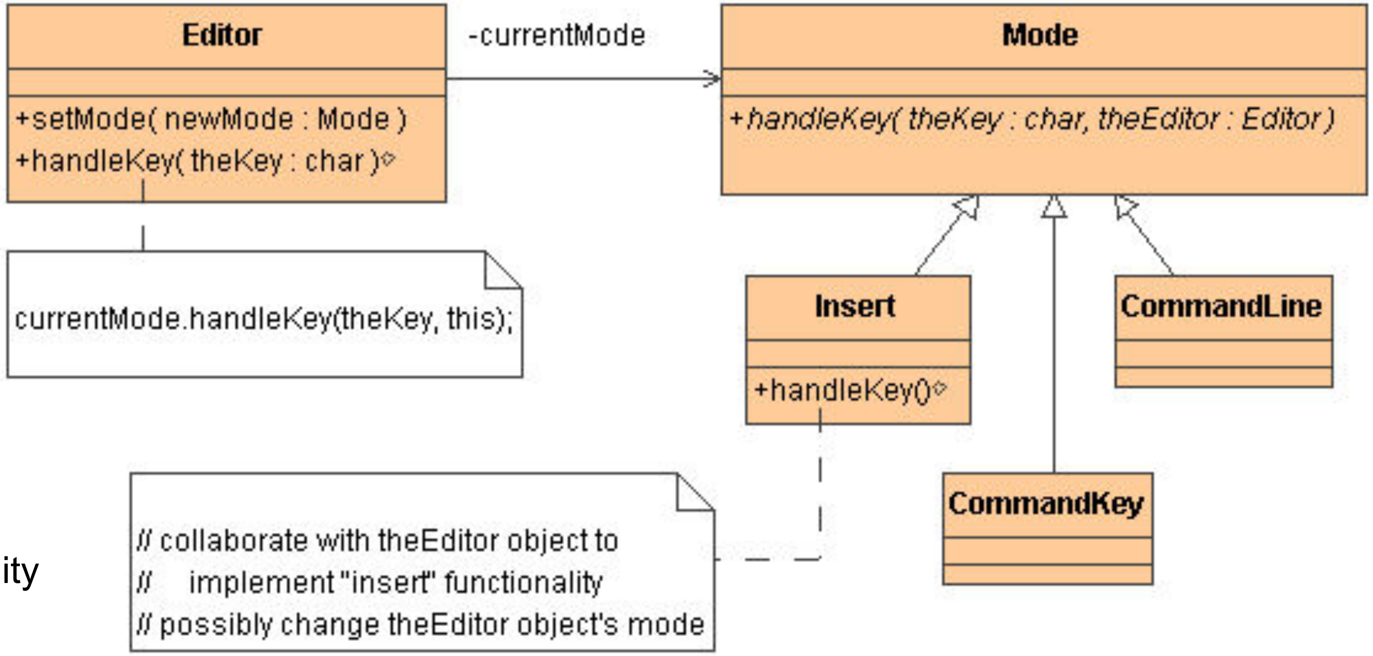
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



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

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor


Editor
+setMode(newMode : Mode)
+handleKey(theKey : char)
-currentMode
Mode
+handleKey(theKey : char, theEditor : Editor)
Insert
+handleKey()
CommandLine
CommandKey

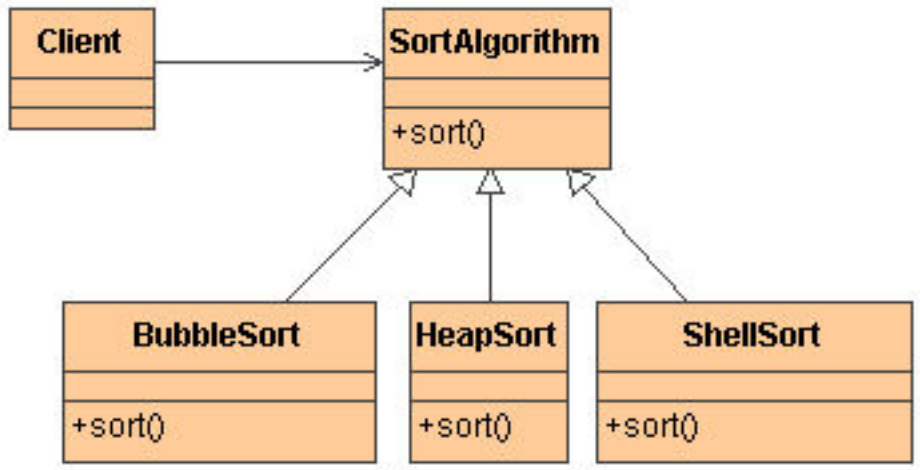
```
currentMode.handleKey(theKey, this);
```

```
// collaborate with theEditor object to
//    implement "insert" functionality
// possibly change theEditor object's mode
```

how to make behavior depend on state? This pattern allows an object to change its behavior when its internal attribute values change. This approach uses code (instead of data structures) to specify state transitions

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

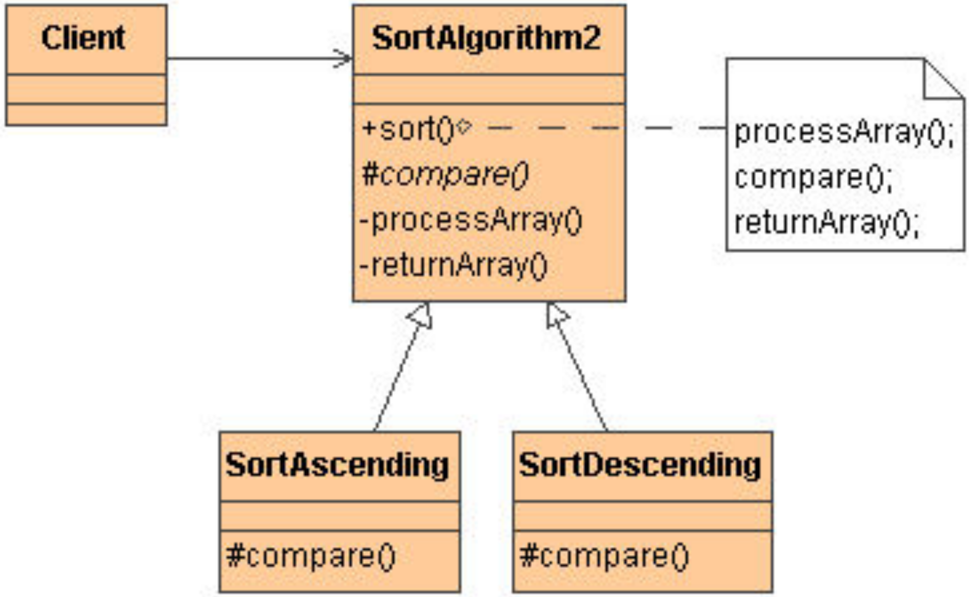
Which pattern?



Define a family of algorithms, encapsulate each one, and make them interchangeable. Let the algorithm vary independently from clients that use it. Configure the choice of algorithm so that the the client is the wrapper, the algorithm object is the delegate

Which pattern?

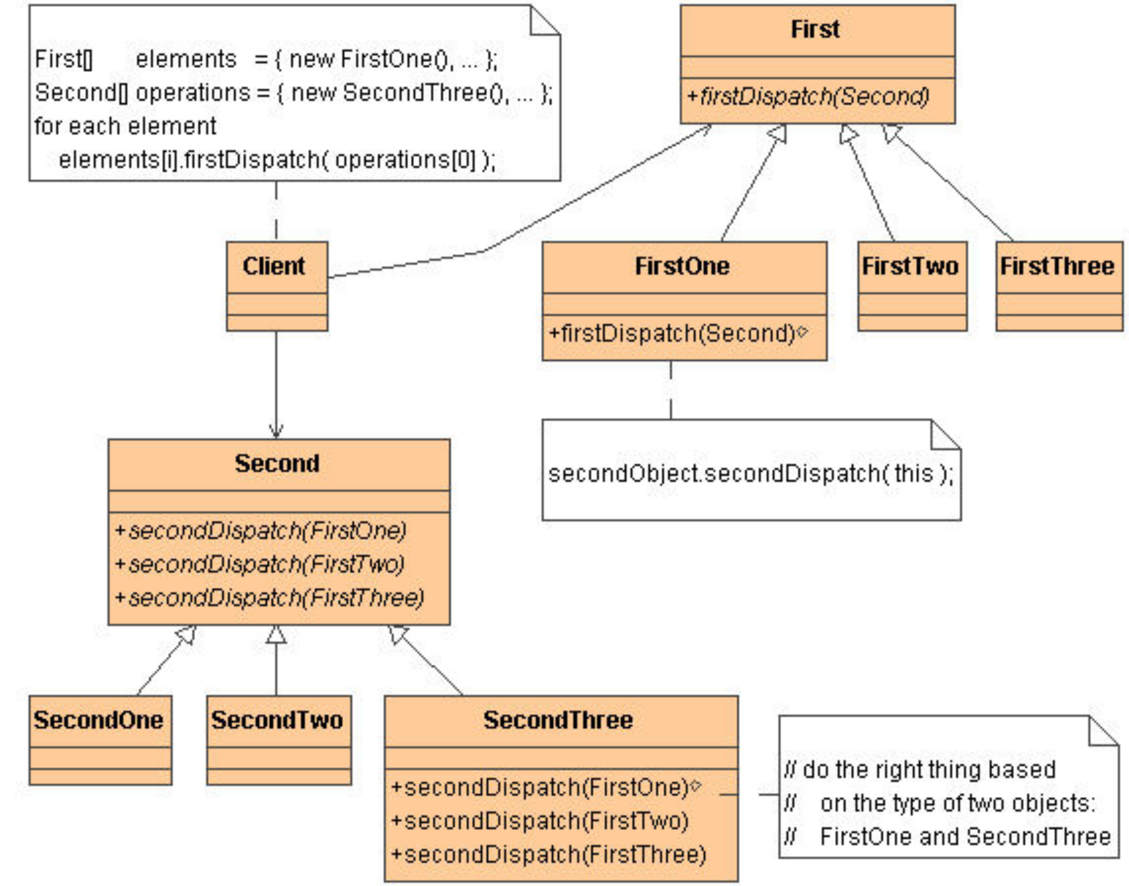
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



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.

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



```
First[]      elements   = { new FirstOne(), ... };
Second[] operations = { new SecondThree(), ... };
for each element
   elements[i].firstDispatch( operations[0] );
```

First
+firstDispatch(Second)

Client

FirstOne
+firstDispatch(Second)

FirstTwo

FirstThree

```
secondObject.secondDispatch( this );
```

Second
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)

SecondOne

SecondTwo

SecondThree
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)

```
// do the right thing based
//    on the type of two objects:
//    FirstOne and SecondThree
```

An operation to be performed on the elements of an object structure without changing the classes on which it operates. When a person calls a taxi company, the company dispatches a cab to the customer. Upon entering the taxi the customer is no longer in control of his own transportation, the taxi (driver) is

References

- Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley*, 1994.*
- Larman, *Applying UML and patterns.* Pearson*, 2005*
- Freeman, Freeman, Sierra, and Bates, *Head First Design Patterns*. O'Reilly, 2004

Useful sites

www.pearsonvue.com/omg/

www.vincehuston.org/dp/patterns_quiz.html

www.objectsbydesign.com/projects/umltest/bparanj-answers-1.html

dn.codegear.com/article/31863

Think about it!