

Appunti per GOF Design Patterns (Java)

~ Tommaso Pellegrini



Creatational

- Abstract factory 1.1
- Factory method 1.2
- Builder 1.3
- Prototype 1.4
- Singleton 1.5

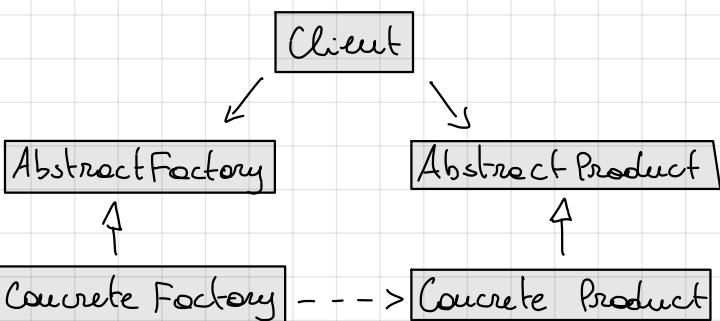
Structural

- Adopter 2.1
- Bridge 2.2
- Composite 2.3
- Decorator 2.4
- Proxy 2.5

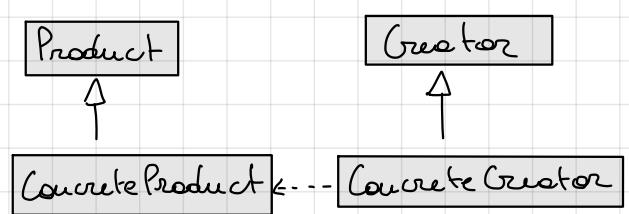
Behavioural

- Iterator 2.6
- Command 2.7
- Observer 2.8
- Interpreter 2.9
- Visitor 2.10

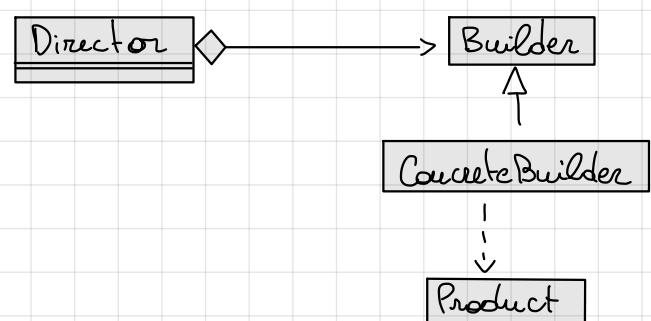
1.1 Abstract Factory



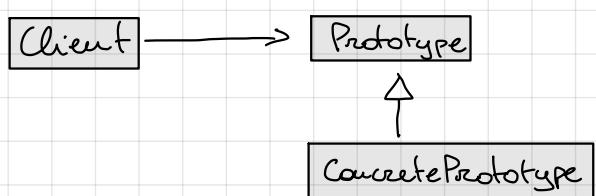
1.2 Factory Method



1.3 Builder

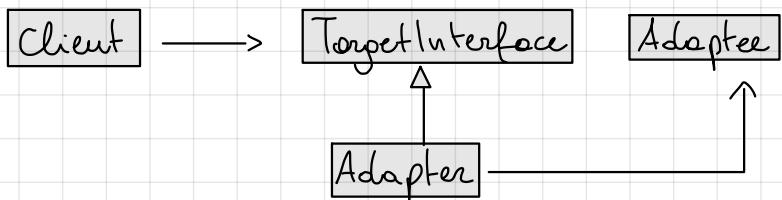


1.4 Prototype

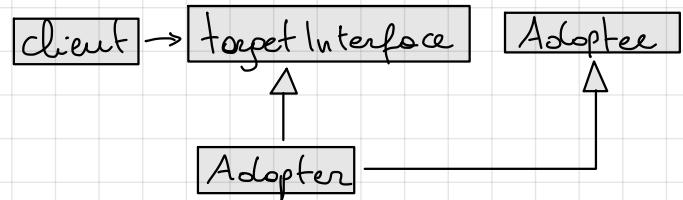


2.1 Adapter

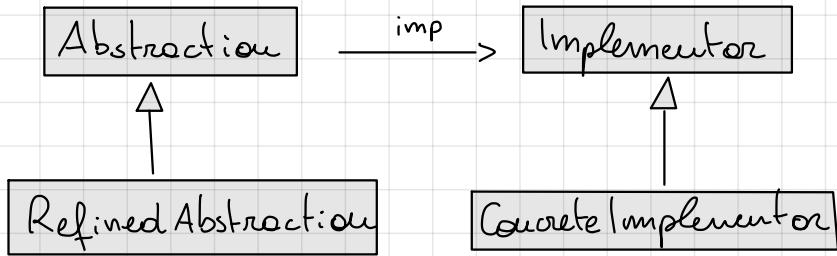
Obj Adapt



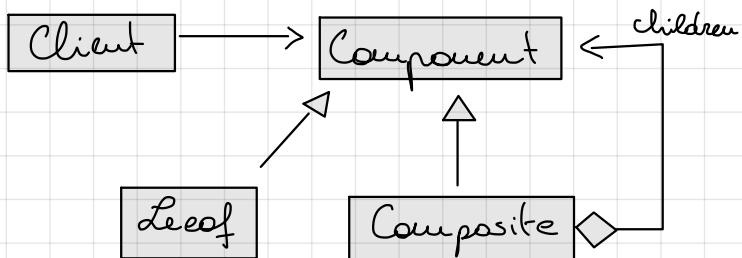
class Adapt



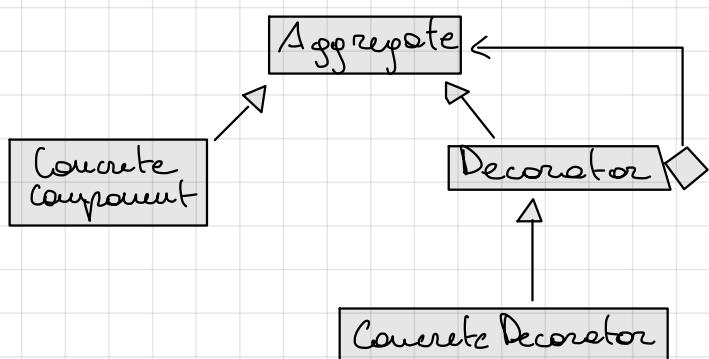
2.2 Bridge



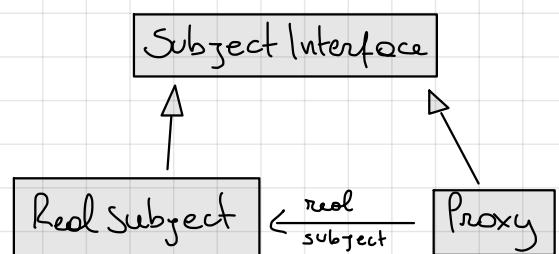
2.3 Composite



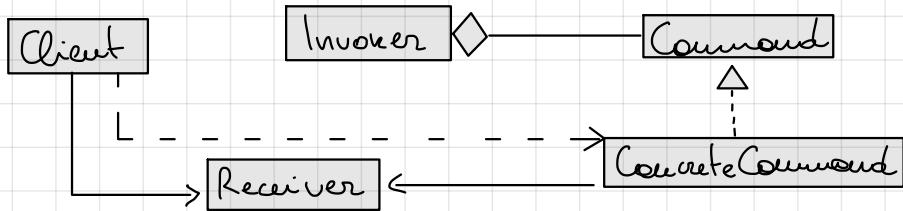
2.4 Decorator



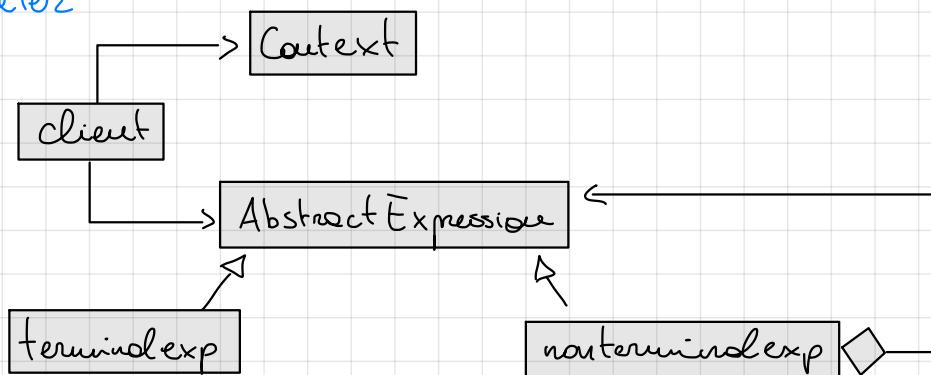
2.5 Proxy



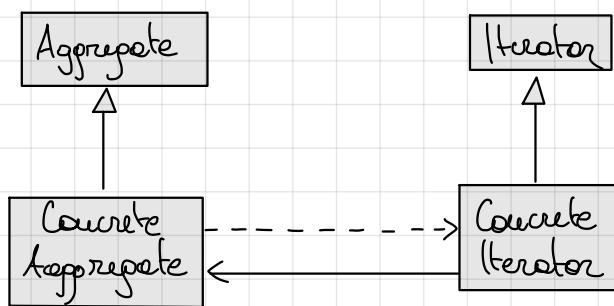
3.1 Command



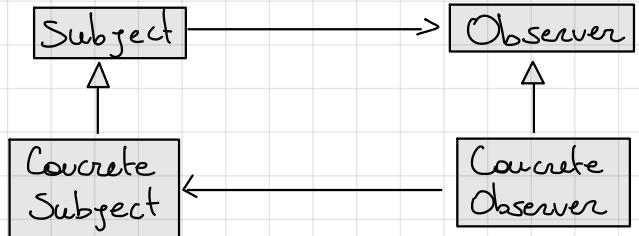
3.2 Interpreter



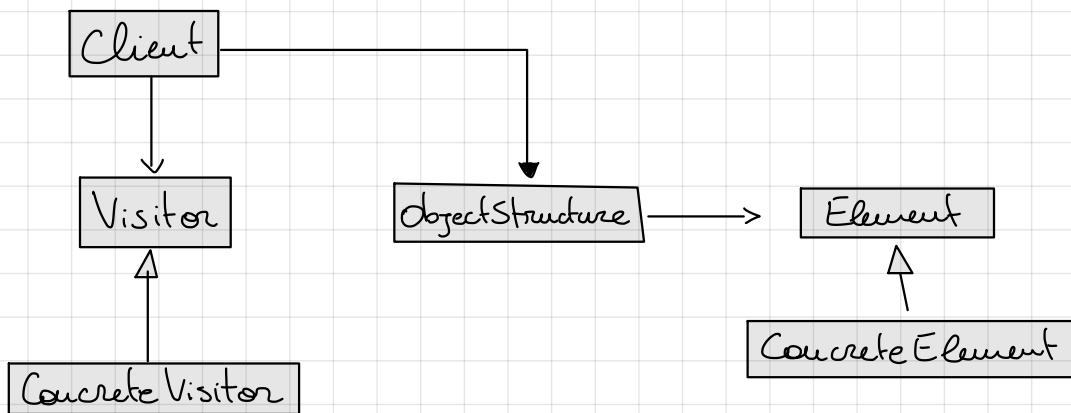
3.3 Iterator



3.4 Observer



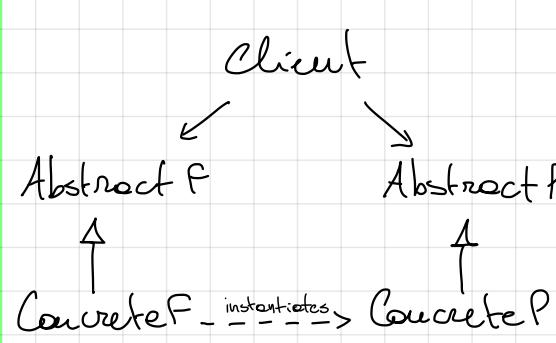
3.5 Visitor



Abstract factory

CREATIONAL

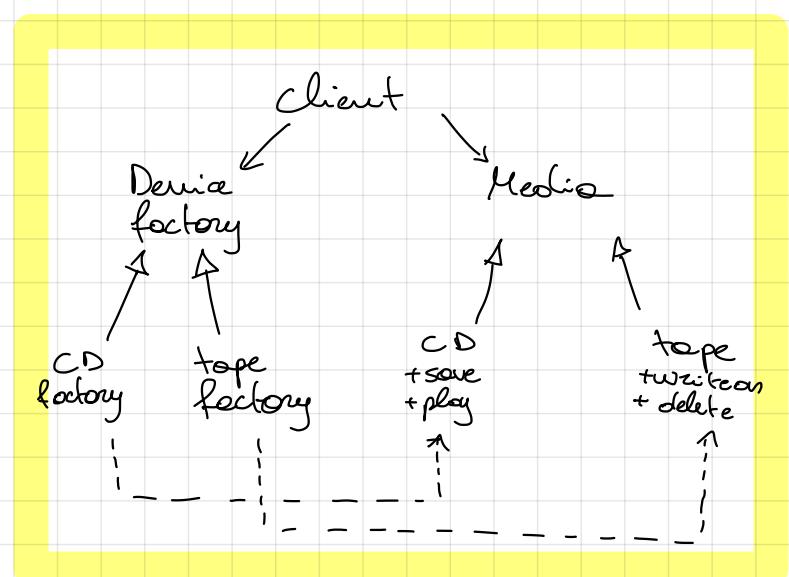
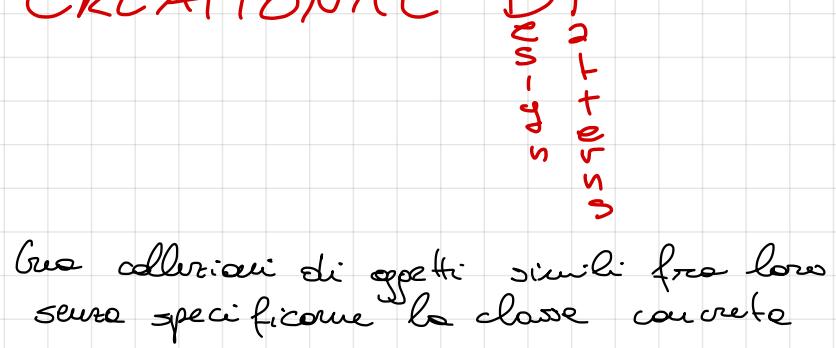
Dantes
Dantes



```
public interface DeviceFactory {  
    ... + create Media  
    {  
        Player  
        CD  
    }  
}  
  
public interface Media {  
    ...  
}  
}
```

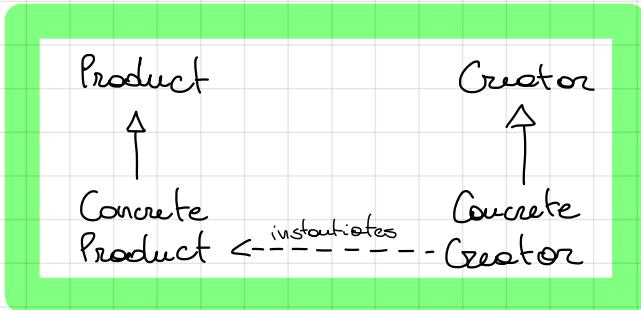
```
public class CDFactory implements DeviceFactory { ... }  
" " tape " " " " { ... }
```

```
public class CD implements Media {  
    public play();  
    public saveOn();  
}  
  
public class Tape implements Media {  
    ...  
}
```

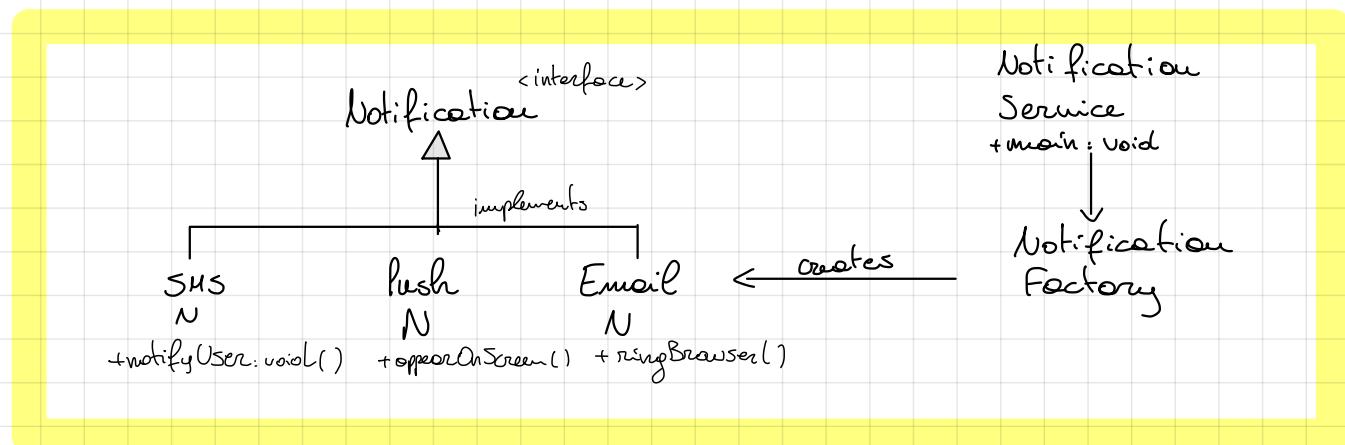


client $\xrightarrow{\text{req}}$ Ab factory $\xrightarrow{\text{req}}$ Conc Factory $\xrightarrow{\text{instan}}$ object

Factory Method



Dato uno interface, lasciamo che le sottoclassi scelgano che tipo di oggetto creare. C'è una delega fra istanziazioni.



```

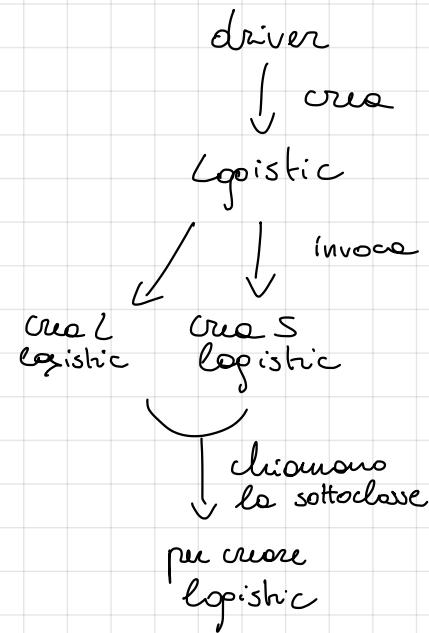
public interface Notification {
    void notifyUser();
}

public class SMS implements Notification {
    @Override
    // metodi
}

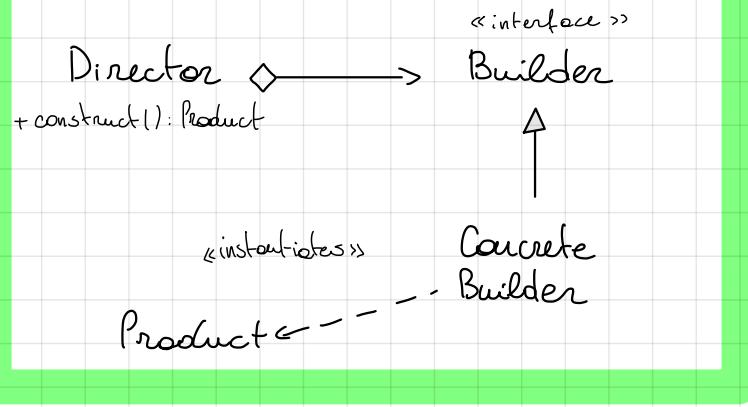
public class NotificationFactory {
}

public class NotificationService {
    public void main(String[] args) {
        NotificationFactory nfactory = new NotificationFactory();
        Notification not... = nfactory.create("sms");
        not... .notifyUser();
    }
}

```

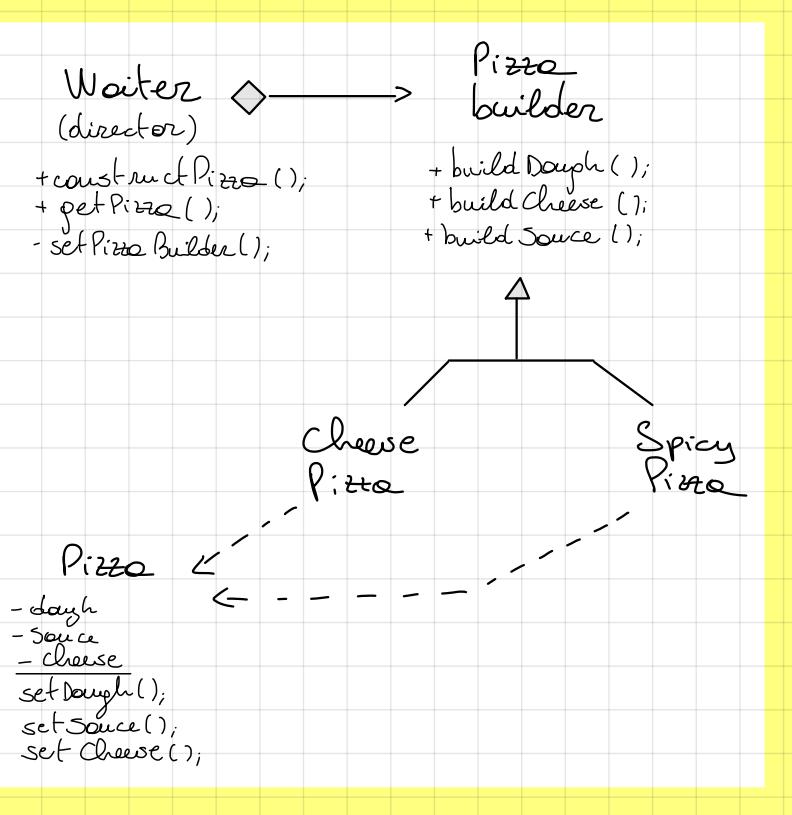


Builder



Separa la costruzione di un oggetto complesso dalla sua rappresentazione, così che lo stesso processo possa creare rappresentazioni diverse.

- Product - class: type of complex obj
- Builder - abstract class / interface: definisce i passi per la creazione.
- Concrete B - class: possono esserci n classi, ereditano da Builder.
- Director - class: controlla s/dg che genera il prodotto s/dg finale.



```

public class Pizza {
    public dough;
    public source;
    ...
    +setDough();
}
  
```

```

public class Waiter {
    public constructPizza();
    public getPizza();
    private setPizzaBuilder();
}
  
```

```

abstract class PizzaBuilder {
    public buildDough();
}
  
```

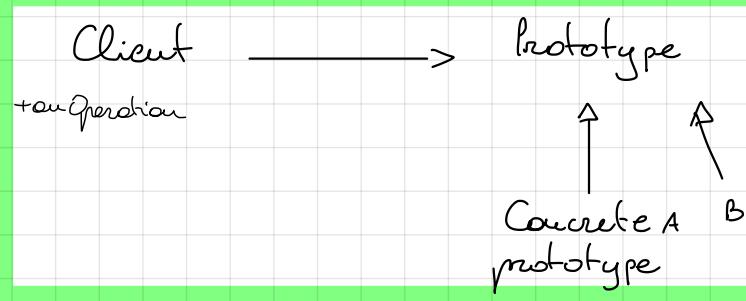
```

class CP extends PB {
    @Override
    // methods
}
  
```

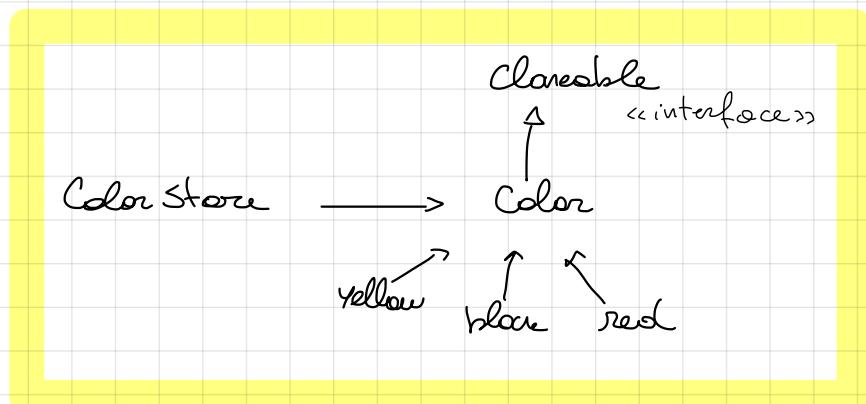
```

class SP extends PB {
    // same ...
}
  
```

Prototype



Prototype permette di nascondere la complessità di creare nuove istanze dal client. Al posto di creare new, copia un oggetto che esiste già, che può poi essere modificato.



```
public class ColorStore { ... }

abstract class Color implements Cloneable { ... }

class yellow extends Color { ... }

class red extends Color { ... }

class black extends Color { ... }

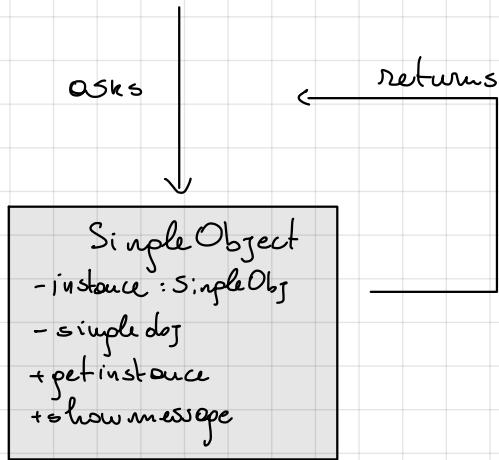
//driver
class prototype {
    main {
        ColorStore.addColor("blue").getColor()
        ...
    }
}
```

Singleton

Una singola classe crea un singolo oggetto e permette di accederne diretti senza dover istanziare l'oggetto nelle classi.

Singleton Pattern Diagram

+ main(1: void)



SingleObject.java

```
public class SingleObject {  
    private static SO instance = new SO();  
    private SO() {} //constructor  
  
    public static SO getInstance() {  
        return instance;  
    }  
    public void showmsg() { cout >> "hi!"; }  
}
```

public class Driver {

pub. static void main(String[] args) {

SO object = SO.getInstance(); \rightsquigarrow Now si fa come al solito:
 object.showmsg();
}

so object = new SO();

Il costruttore della classe è privato (non lo usiamo direttamente) e la classe ha una istanza statica di sé stessa.

↳ Il metodo che chiama l'istanza è anche lui statico e serve per portare le sue istanze statiche nel mondo esterno.

STRUCTURAL DESIGN PATTERNS

Adapter

Obj adapt

Client → Target Interface

↑
adapter
↓

adoptee

Class adapt

Client → Target Interface

↑
adapter
↓

adoptee

Permette alle classi con interfacce incompatibili di lavorare insieme
facendo wrap delle proprie interfacce attorno a quelle di una classe esistente

«class»
ShapeCalculator
oddShape();
area();
perimeter();



«interface»
Shape
perimeter();
area();
↑
Square
side: double
Square();
perimeter();
Area();
↑
Triangle
side1: int
side2: int
side3: int
Triangle();
perimeter();
Area();

```
public class ShapeCalculator {  
    List<Shape> shapes = new ArrayList<Shape>();  
    public void oddShape() { ... }  
    public double perimeter() { ... }  
    public double area() { ... }  
}
```

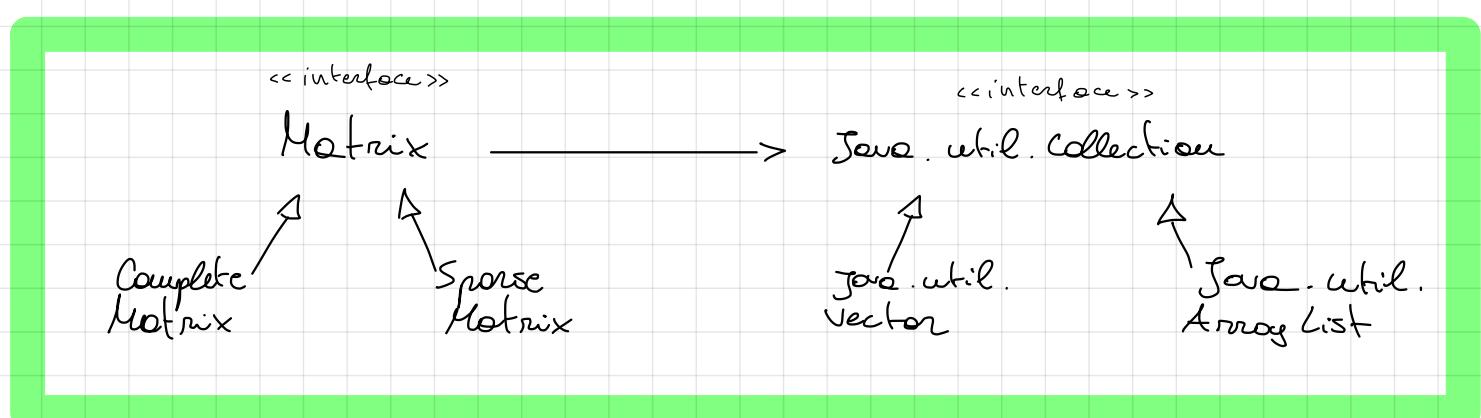
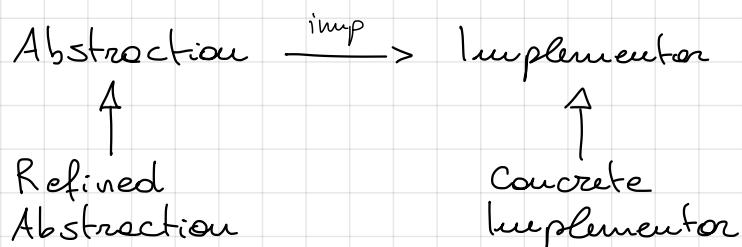
```
public interface Shape {  
    + double perimeter();  
    + double area();  
}
```

= triangle

```
public class Square implements Shape {  
    + double side = 1.0;  
    public Square() {}  
    public Square(double side) {  
        this.side = side  
    }  
    public double perimeter() {  
        return side * 4;  
    }  
    public double area() { ... }
```

Bridge :

Separare una astrazione dalla sua implementazione così che le due possono lavorare separatamente.



```
public interface Matrix {  
    public put();  
    public get();  
}
```

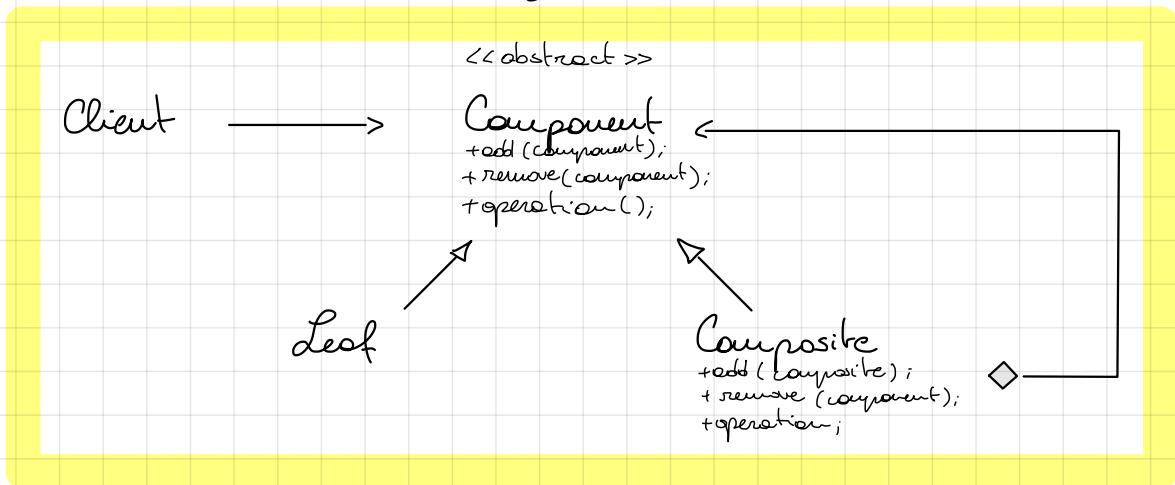
```
public interface java.util.Collection {  
    ...  
}
```

```
public class CM = some SM implements Matrix {  
    @override  
    public put();  
    public get();  
    public CM() {};  
    public CM(Num[]) {}  
}
```

```
public class java.util.xxx {  
    ...  
}
```

Composite:

Componere zero o più oggetti simili così che possono essere manipolati come se fossero un unico oggetto.



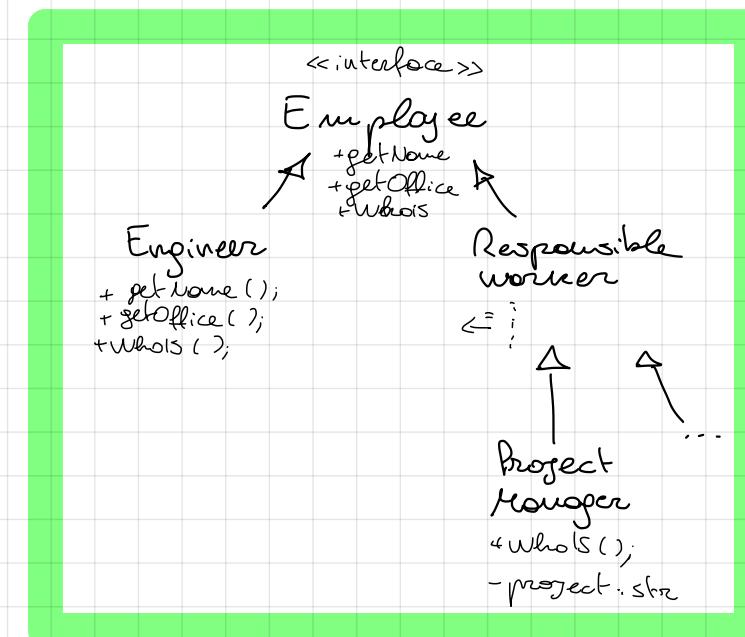
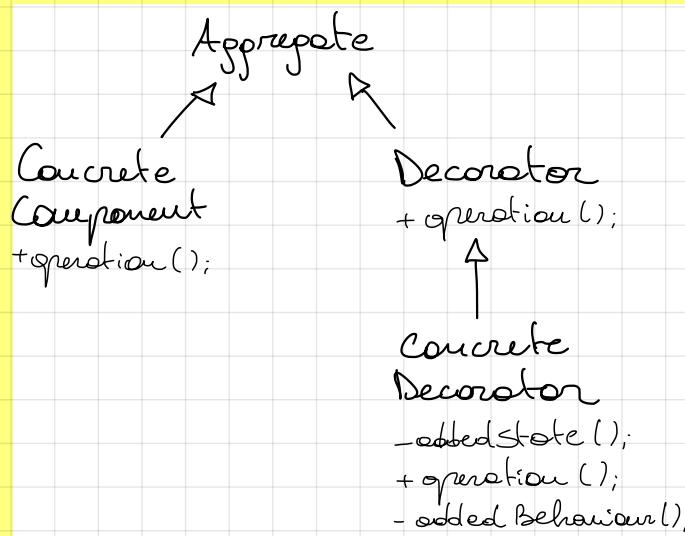
```
public abstract class Component {  
    + add (component);  
    + remove (component);  
    + operation();  
}
```

```
public class Leaf extends Component {  
    //operations  
}
```

```
public class Composite extends Component {  
    @Override  
    // metodi  
    // :  
}
```

Decoratore:

Aggiunge o fa acciudere dinamicamente del comportamento in un metodo esistente di un oggetto.



```

public interface Employee {
    +getName();
    +getOffice();
    +whoIs();
}
public Employee () { }
  
```

```

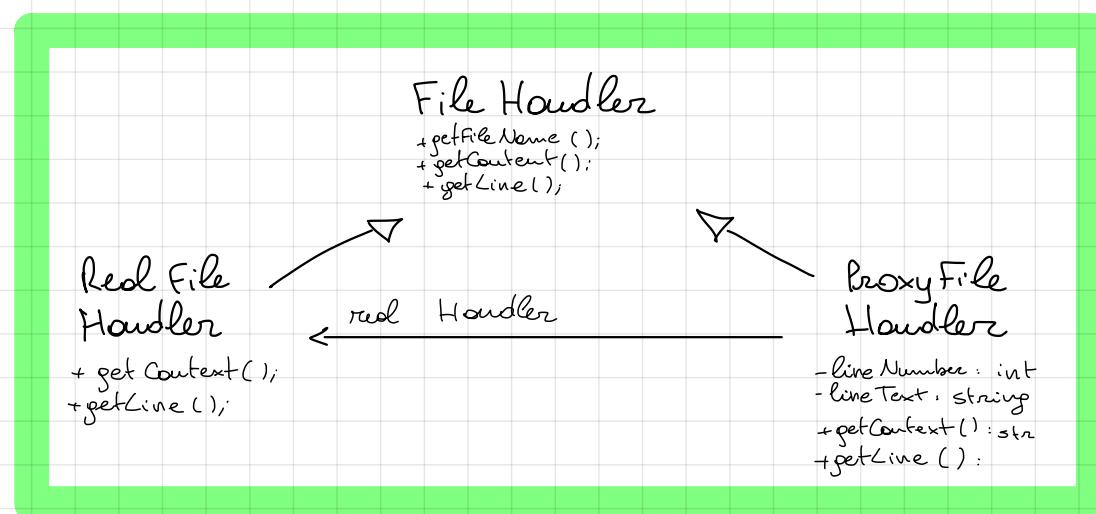
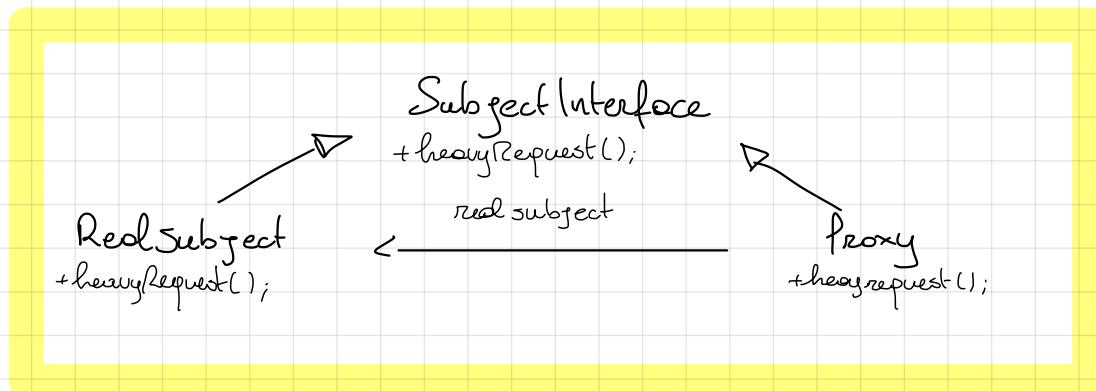
public class Engineer implements Employee {
    //Metodi
    //Responsabile
    //worker
}
  
```

```

public class projectManager extends ResponsibleWorker {
    //-----
}
  
```

Proxy:

Fornisce un placeholder per un altro oggetto per controllo dell'accesso e ne riduce la complessità.



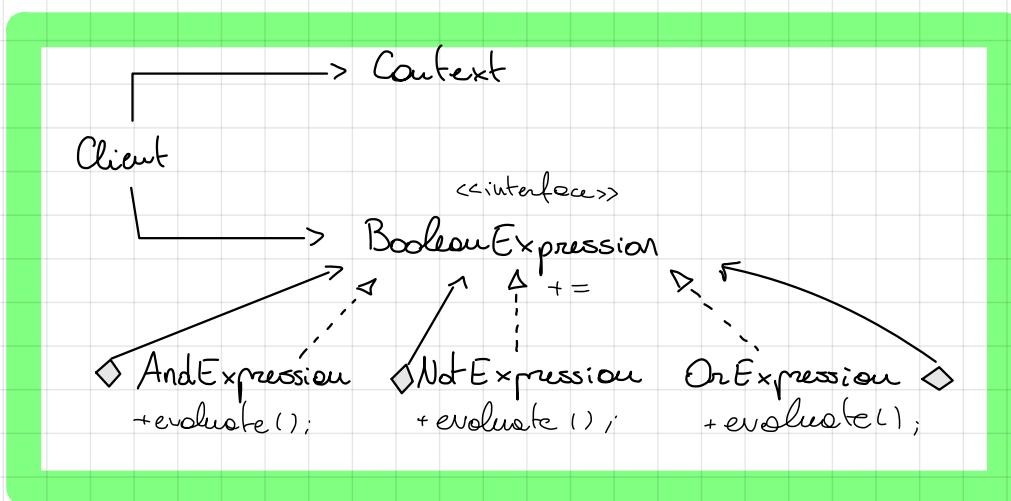
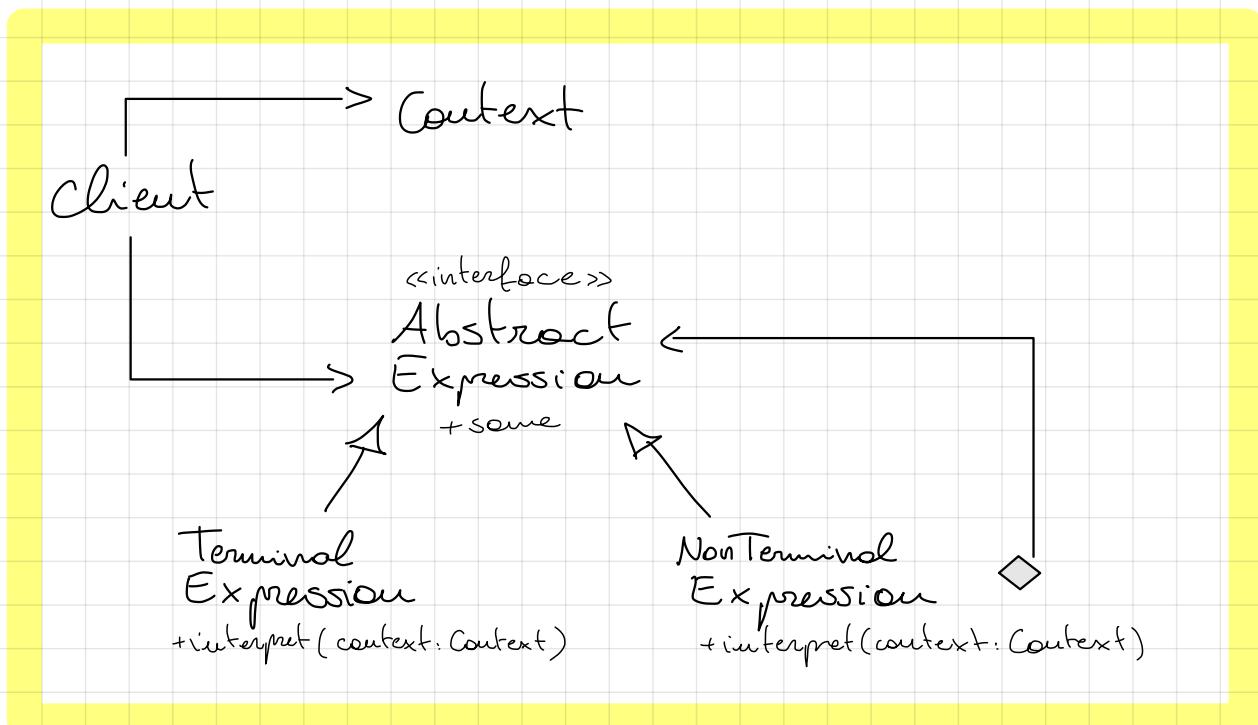
```
public interface File Handler {  
    // metodi  
}
```

```
public class ProxyFile implements File Handler {  
    ...  
}
```

```
public class RealFileHandler implements File Handler {  
    ...  
}
```

BEHAVIORAL PATTERNS

Interpreter: implementa un interprete per un linguaggio specializzato, spesso un DSL.

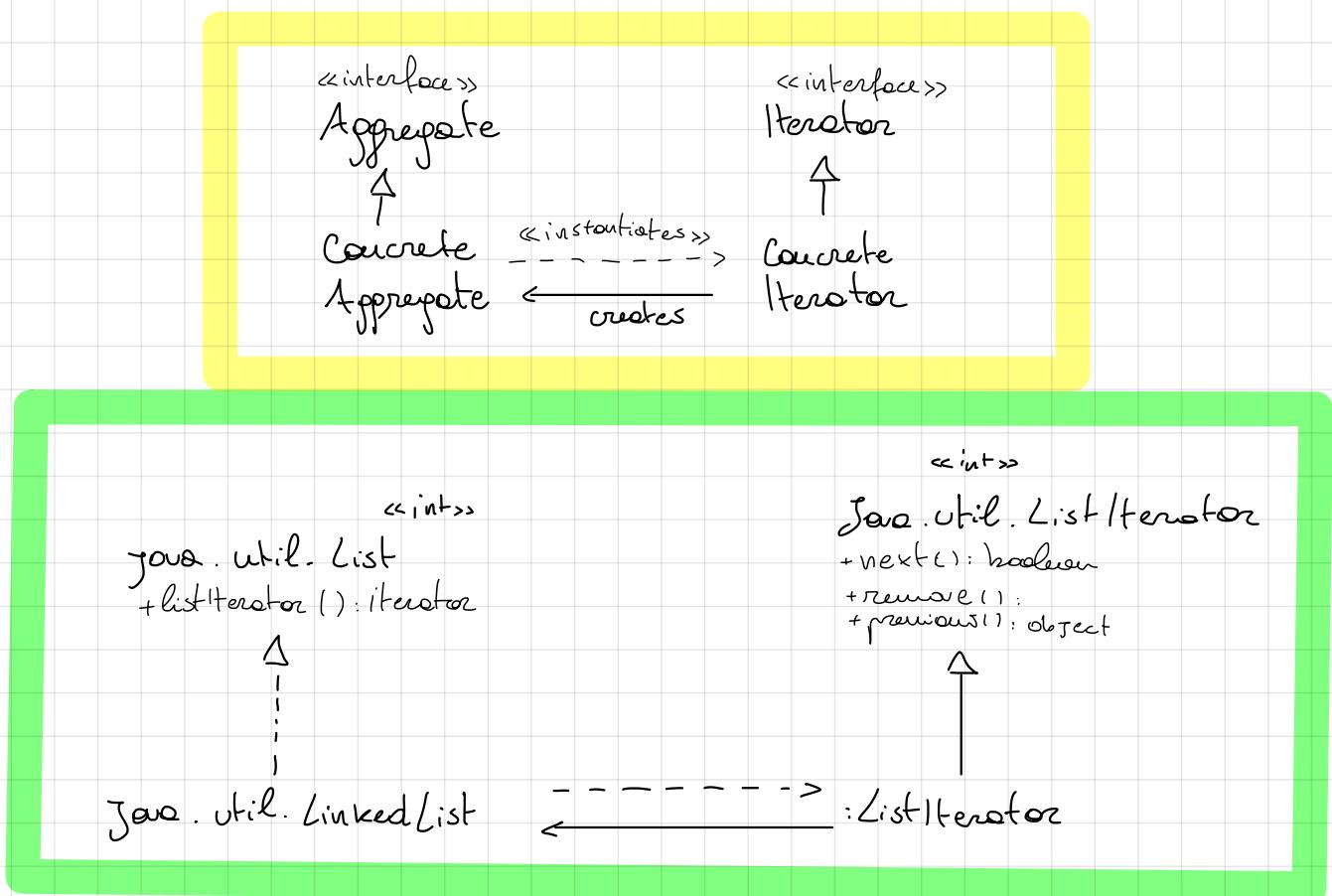


```
public interface BooleanExpression {  
    evaluate();  
}
```



```
public class Expression implements BooleanExpression {  
    @Override  
    public evaluate() {  
    }  
}
```

Iterator: accede agli elementi di un oggetto sequenzialmente senza esporre le loro rappresentazioni sottostante.



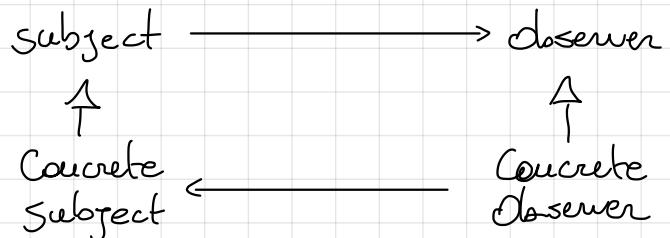
```
public interface java.util.List {  
    ...  
}
```

```
public interface java.util.ListIterator {  
    ...  
}
```

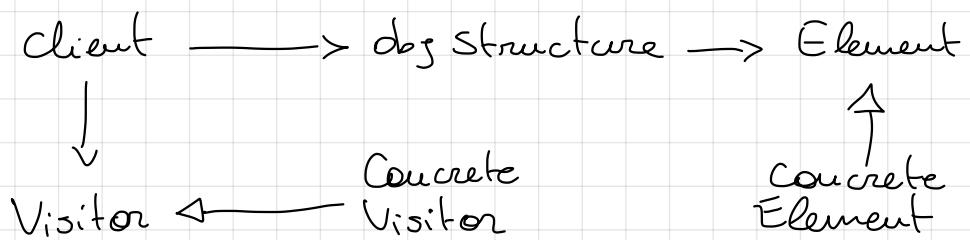
```
public class java.util.LinkedList implements java.util.List {  
    ...  
}
```

```
public class ListIterator implements java.util.ListIterator {  
    ...  
}
```

Observer: è un pattern publish / subscribe. Permette ad un numero di osservatori opposto di vedere un evento.



Visitor : separa un algoritmo da un oggetto structure muovendone le gerarchie dei metodi in un oggetto.



Observer e visitor sono noiose...

Riassunto per design patterns:

Creational

1. Factory method: dà una interfaccia, lasciamo decidere alle sottoclassi che tipo di oggetto creare. C'è una differenza fra istanze.
- Animal creator
2. Abstract factory: crea una collezione di oggetti simili fra loro senza specificarne la classe concreta.
- Furniture factory - chaise longue (modern, classic)
3. Builder: separa la costruzione di un oggetto dalla sua rappresentazione, così che lo stesso processo possa creare rappresentazioni diverse.
- User → .ope() .det()
4. Prototype: permette di nascondere le complessità di creare nuove istanze dal client. Al posto di new() copiamo un oggetto esistente + modifichiamo.
- Shape <sup>square
_{circle} Proto → getClone()
5. Singleton: costringe una classe a creare una singola istanza di un oggetto.

Structural

1. Adapter: permette a classi con interface incompatibili di lavorare insieme facendo wrapping della propria interface attorno a quelle di una classe esistente.
- BankCustomer / CreditCard / Bank details
2. Bridge: separa una astrazione dalla sua implementazione cosicché possono lavorare separatamente.
- DrawAPI $\begin{cases} \rightarrow red \\ \leftarrow blue \end{cases}\> circle$
3. Composite: compone zero o più oggetti simili così da poterli manipolare come se fossero un solo oggetto.
- Person → addSubordinate();
4. Decorator: aggiunge o sovrascrive il comportamento di un metodo esistente di un oggetto.
- Shape Decorator

So Proxy: fornisce un placeholder per un oggetto per controllarne l'accesso e ridurne la complessità.

→ Image / Real Image

Behavioral:

1. Interpreter: implementa un interprete per il linguaggio L specializzato, spesso DSL.

→ Terminal Expression

2. Iterator: accede agli elementi di un oggetto in modo sequenziale, senza esporre la sua rappresentazione sottostante.

→ Name Repository Iter

3. Observer: è una pattern publish / subscribe e permette ad un numero di osservatori oggetto di vedere una azione.

→ Binary / HexObserver

4. Visitor: separa un algoritmo da un oggetto strutturando la gerarchia dei metodi in un oggetto.

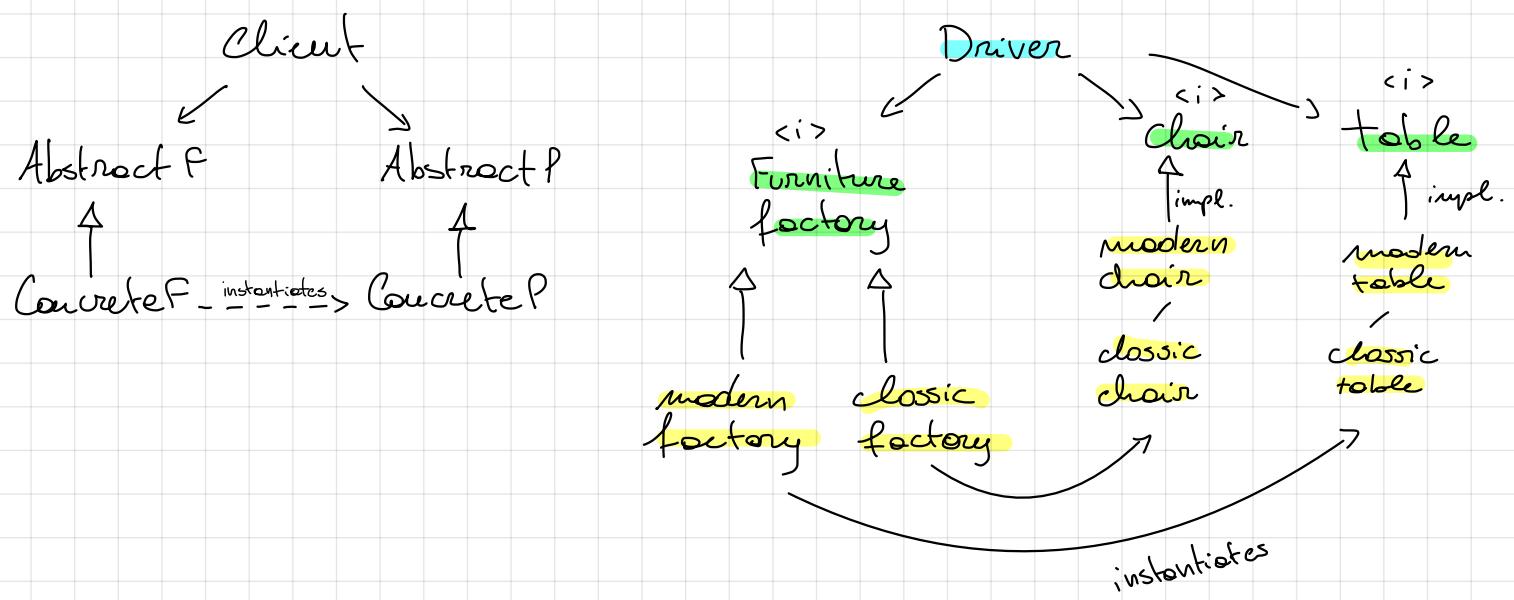
→ Computer Parts

5. Command: trasforma una richiesta in un oggetto stand alone che contiene tutte le informazioni della richiesta. Permette di passare una richiesta come argomento di un metodo ma anche ritardare o mettere in coda richieste e supportare operazioni.

→ Broker / Buy Sell / Stock

Implementazioni:

1) Abstract factory



2. Concrete product B ← 4. Concrete factory B

1 Abstract product A B

↑ implements

2 Concrete product A ← 4. Concrete factory A
instantiates

3. Abstract factory → +⁴_B ← 5. Client

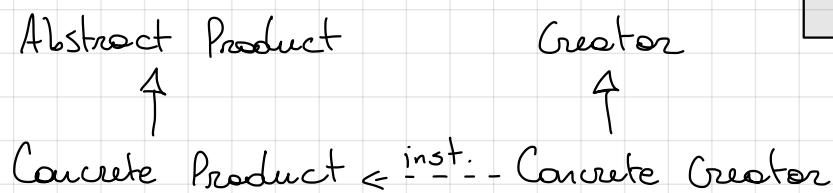
calls

↑ creates

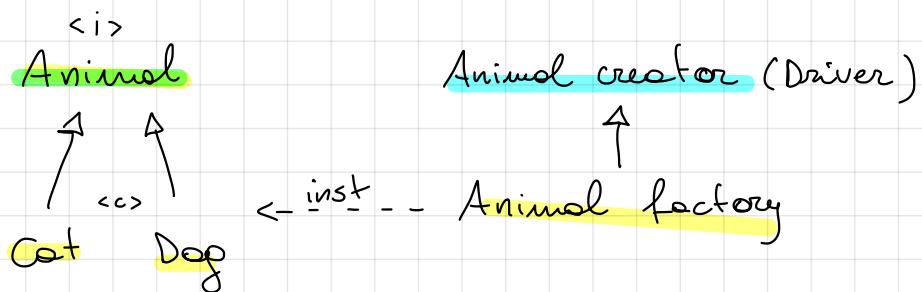
Client calls AF → creates CF → inst CP → AP

→ Produce famiglie di oggetti simili fra loro senza specificare le classi concrete

2) Factory method



Permette ad una interfaccia di creare oggetti: in una superclasse, me lascia modificare l'oggetto creato alle sottoclassi

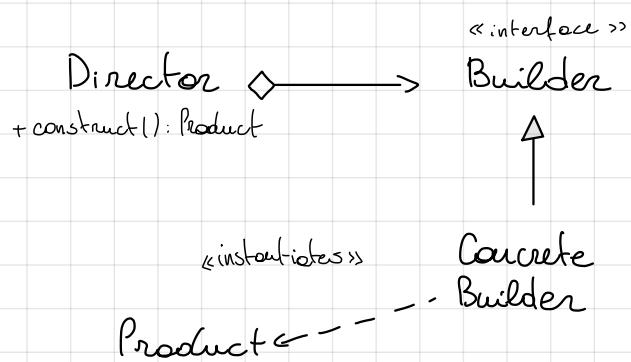


AnimalCreator $\xrightarrow{\text{creates}}$ Animal factory of $\xrightarrow[\text{of}]{\text{uses}}$ creates animal

AnimalFactory of = new Animalfactory()
 $\text{Animal } a = \text{of}. \text{createanimal}(\text{animal})$

a. bite();

3) Builder



Permette di costruire oggetti complessi usando oggetti semplici con un approccio passo a passo.

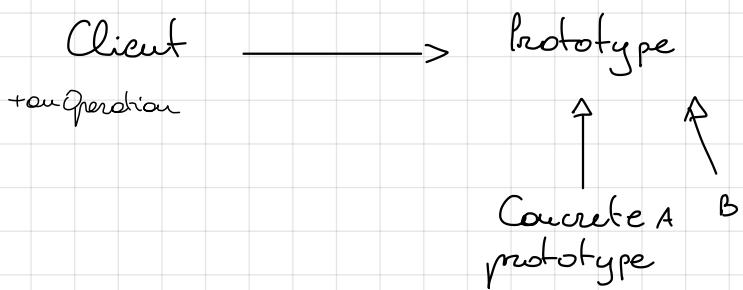
Director costruisce \rightarrow Builder \dashrightarrow product

Driver \rightarrow **UserBuilder** $\xrightarrow{\text{costruttori compi obbligatori + costruttori per altri campi + return}}$ stessi attributi
 $\xrightarrow{\text{metodo build() = User user = new User(this)}}$

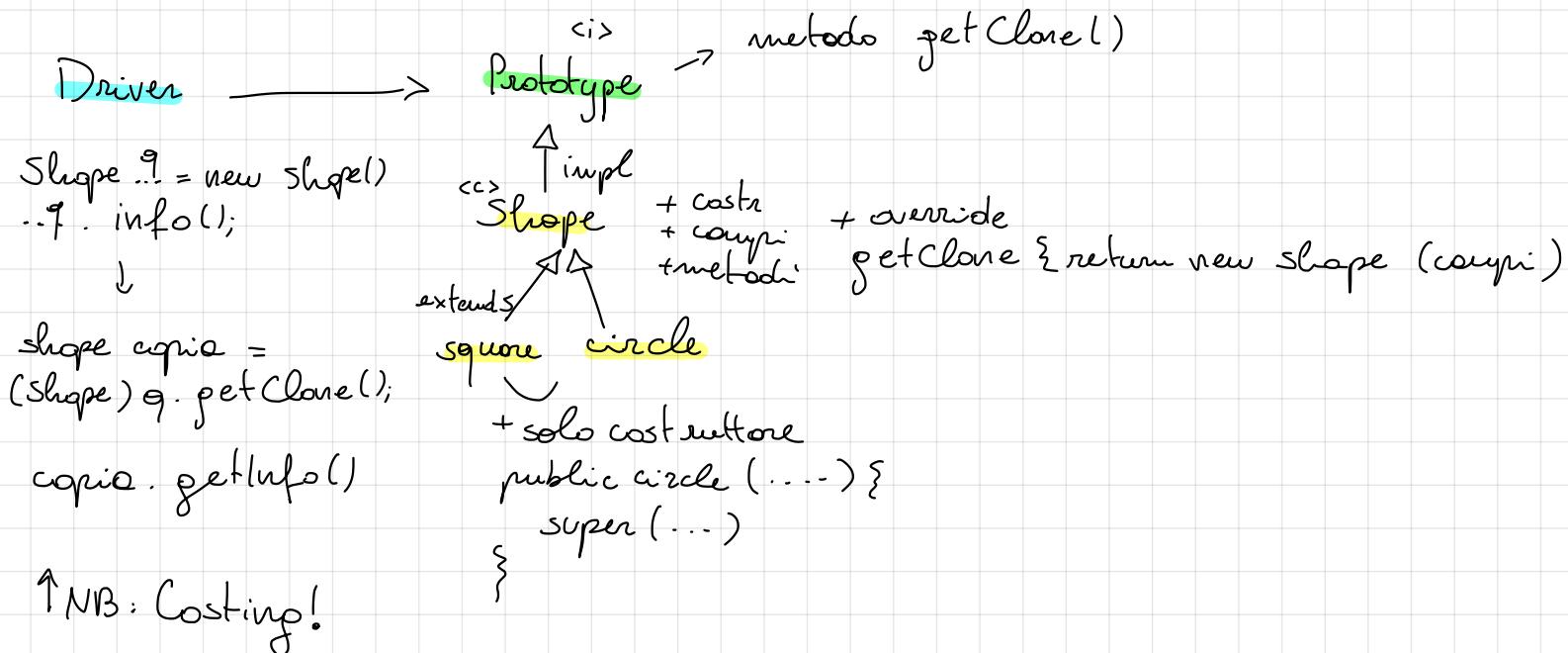
User user1 = new User (obblig)
 ↗
 {
 .ape
 .phone } costruttori
 .build(); metodo

User \rightarrow classe con attr final, 1 costruttore completo e metodi

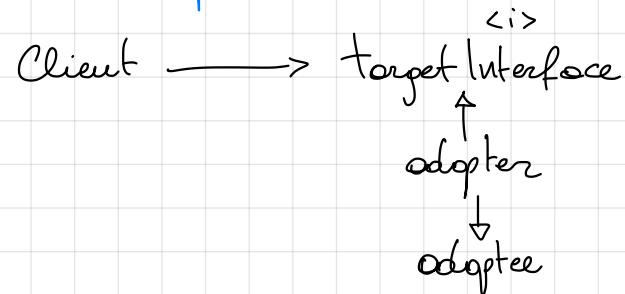
4) Prototype.



Nasconde la complessità di creare nuove istanze di un oggetto dal client. Al posto di new chiamano un oggetto per poi modificarlo.



5) Adapter:

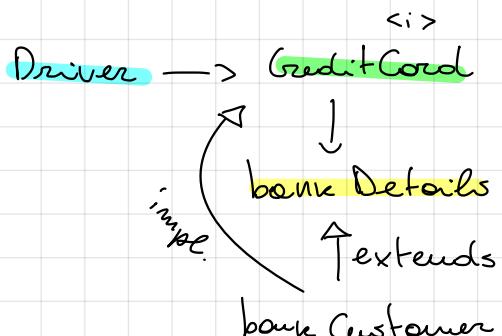


CreditCard <i> → details();
→ methods();

Driver = CreditCard targetInterface =
new bankCustomer();

target... . details();
target... . methods();

Bank Details = { attributi, gettor e settori }



Bank Customer = details() usa :
setter per impostare i dettagli
@ Overrides getCardNo() {
} getters + print

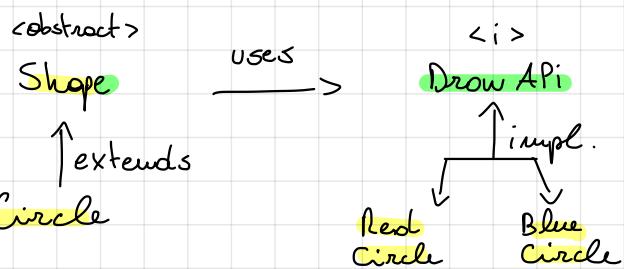
Funziona come ponte fra due interface non compatibili fra loro.

6) Bridge:

Abstraction $\xrightarrow{\text{imp}}$ Implementor



Separare una astrazione dalle sue implementazioni, così che possono venire separate.



Circle: extends shape, override draw(), ha attributi e costruttore SUPER — (draw API)

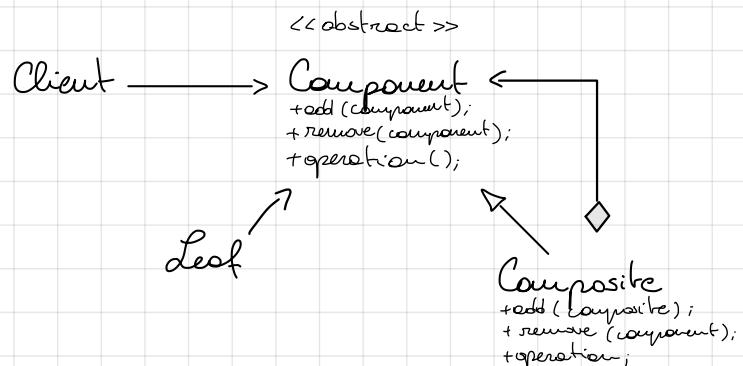
Driver: Shape rc = new Circle (attr, RedCircle);
rc.draw

Draw API = metodo draw(circle());

Red/Blue Circle = classe con solo override di drawCircle(); aggiunge il colore nel print

Shape: ha attribut. draw API protected costruttore che lo prende in input la metoda draw();

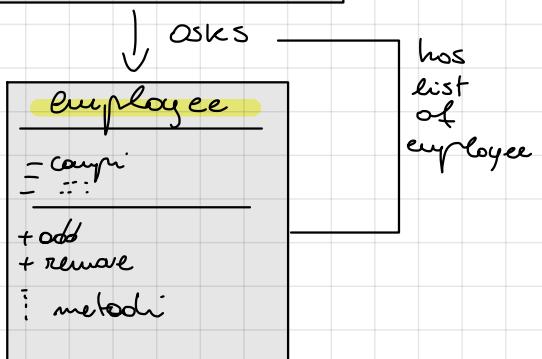
7) Composite:



Usato quando abbiamo bisogno di trattare un gruppo di oggetti in modo simile ad un singolo oggetto. Funziona bene con rappresentaz ed albero

Composite Demo

+ main : void

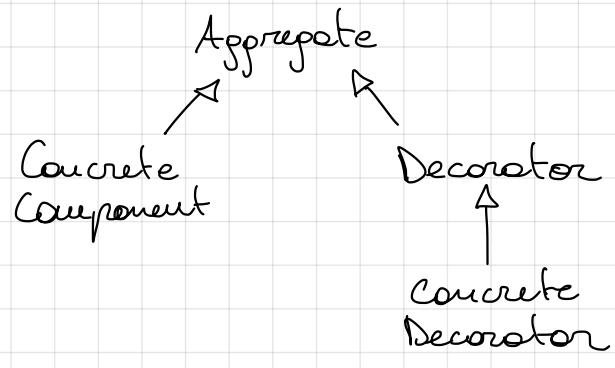


Driver: new Employee
e.add() come subordinato

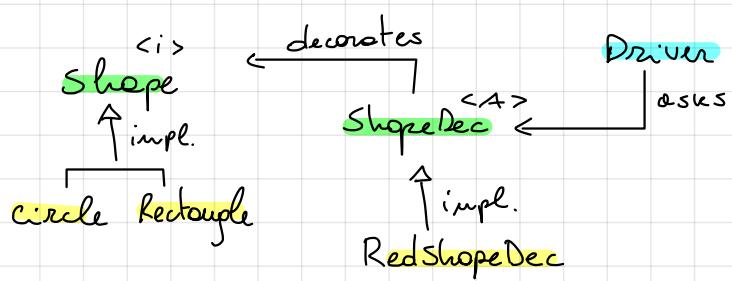
for (...) getSubordinates()

+ List<Employee> getSubordinates();
return subordinates

8) Decorator



Permette di aggiungere nuove funzionalità ad un oggetto senza alterarne la struttura.



Shape: dichiara draw();

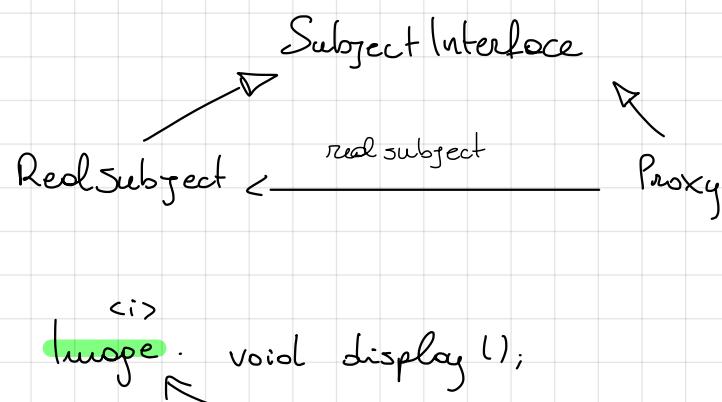
Circle/Rect: override e basta <impl shape (println)

ShapeDec impl shape: protected Shape shapeDec + costruttore (shape shapeD)
+ void draw → chiama draw su \Rightarrow
= shapeD.draw();

RedShapeDec estende Shape = costruttore super + draw() \Rightarrow decs. draw()
 \Rightarrow setRed(decs)

Driver → new Circle new RedCircle + obow for both

9) Proxy



Una classe rappresenta una funzionalità per un'altra classe - fa da placeholder e ne riduce la complessità.

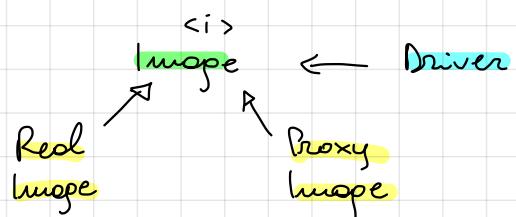


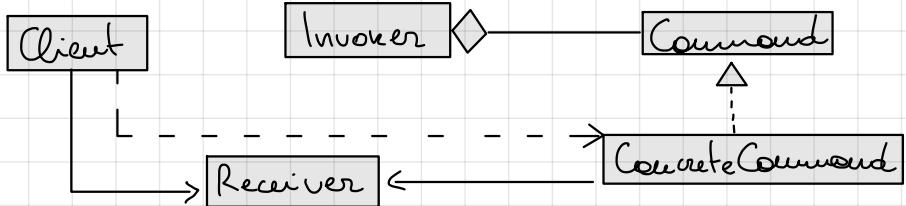
Image: void display();

RealImage impl: costruttore + overridable + void loadDisk();

ProxyImage: RealImage image + override display + proxyImage costruttore
+ altri campi
if realImage = null
realImage = new RealImage();

Driver: Image image = new ProxyImage();
image.display();

10) Command



Trasforma una richiesta in un oggetto stand-alone che contiene tutti i dettagli. Permette anche di passare una richiesta come argomento di un metodo.



Order: execute();

Stock: costru + costru + buy/sell

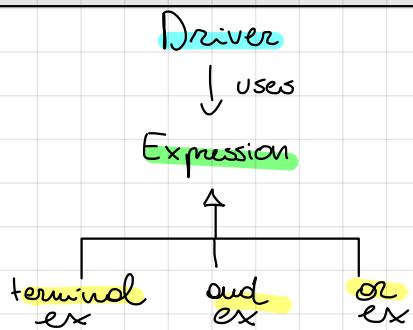
Buy / Sell stock = campo stock + costruttore + execute

Broker = priv List ordini + takeOrder() { add } + placeOrder { clear }

Driver = new Buy / Sell + new Broker + Broker.takeOrder (BuyStock);

11) Interpreter

Fornisce un modo per valutare una grammatica (linguaggio) o regEx.

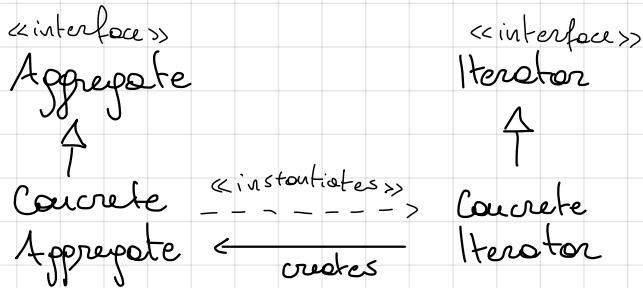


interface expression: evaluate (Str context);

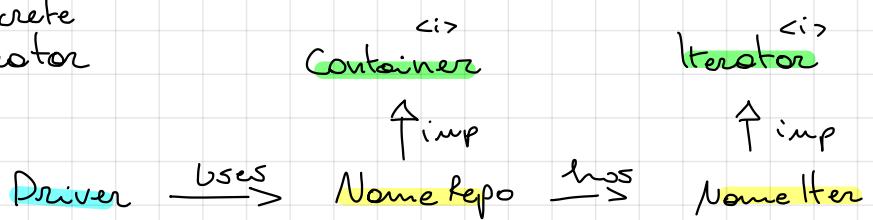
terminal
and
or { ex impl expr = costru + costru + @ evaluate

Expression e₁, e₂ + this.e₁ = e₁ (...)]
or { return e₁.eval (context) || e₂.eval (context) }
and { .. }
or { .. }

12) Iterator



Viene usato per accedere agli oggetti di una collezione sequenzialmente, senza bisogno di conoscerne la rappresentazione sotto stante.



Iterator : getIter();

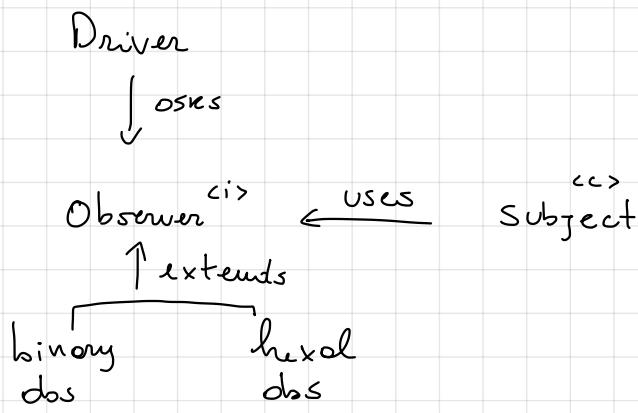
Container : next();, bool hasNext();

NameRepo: array con nomi

NameIter: @next e hasNext

Driver : for Iter iter = nameRepo.getIter(); hasNext()
steps
prints

13) Observer



Viene usata quando c'è una relazione uno-a-tanti tra oggetti di modo che se un oggetto viene modificato, anche i suoi oggetti dipendenti vengano modificati.

