

*Corso di Laurea in Ingegneria Informatica*

# **Corso di Ingegneria del Software**

*Prof. Roberto Pietrantuono*

---

## **Patterns**

# Riferimenti

☞ E. Gamma, R. Helm, R. Johnson, J.M. Vlissides  
“Design Patterns: Elements of Reusable Object-  
Oriented Software”,  
Addison Wesley Professional Computing Series.

# Christopher Alexander – 1/2



- ☞ L'architetto C. Alexander introdusse il concetto di pattern nel suo libro "The Timeless Way of Building".
- ☞ La sua nozione di pattern come best practice per problemi di progettazione comuni e ricorrenti è stata adottata dalla comunità software.
- ☞ Alexander introdusse il concetto di linguaggio di pattern, come un insieme di schemi che complessivamente forniscono un vocabolario base per la progettazione all'interno di uno specifico contesto o dominio applicativo.

# Christopher Alexander – 2/2



- ☞ In “A Pattern Language”, Alexander introdusse uno specifico linguaggio di pattern per il dominio architettonico, con ad esempio sottolinguaggi per la progettazione urbanistica delle città (rivolto a chi fa pianificazione urbanistica) e edile (rivolto agli architetti).
- ☞ Esempi di pattern definiti da Alexander sono:
  - Town patterns: Ring roads (circonvallazioni), night life, and row houses (case a schiera);
  - Building patterns: Roof garden, indoor sunlight, ...

# Pattern di progettazione

- ☞ C. Alexander definisce un pattern come una tripla che esprime una relazione tra un problema, una soluzione, e un contesto:  
Pattern = coppia (problema, soluzione) in un contesto
- ☞ Nel campo dell'ingegneria del software, un **pattern di progettazione** (*design pattern*) è una soluzione comprovata a un problema tipico che sorge nello sviluppo software in uno specifico contesto.

# Design Patterns



Generalmente i **pattern di design** sono catalogati e descritti utilizzando pochi elementi essenziali:

- **nome:** descrive in maniera simbolica il problema di progettazione e le sue soluzioni;
- **problema:** descrive *quando* applicare il pattern. Può descrivere problemi di progettazione specifici o anche strutture di classi o di oggetti sintomo di progettazione non flessibile;
- **soluzione:** descrive gli elementi (oggetti, classi e idiomi) che costituiscono il progetto, le loro relazioni, le responsabilità e le collaborazioni. La soluzione è la descrizione astratta di un problema di progettazione e del modo in cui una configurazione di elementi (classi e oggetti, nel nostro caso), può risolverlo, ma non descrive una particolare progettazione o implementazione;
- **conseguenze:** sono i risultati e i vincoli che si ottengono applicando il pattern, utili per valutare soluzioni alternative, capire e stimare i benefici derivanti dall'applicazione del pattern.

# Un po' di storia ...

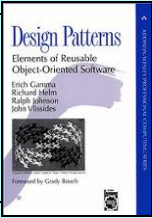
- ➡ Nel 1987 W. Cunningham e K. Beck lavoravano al linguaggio Smalltalk e individuarono alcuni pattern
- ➡ La nozione di pattern è stata resa popolare da Gamma, Helm, Johnson e Vlissides, che lavoravano alla definizione di framework di sviluppo (E++, Unidraw)
- ➡ Essi divennero noti come la Banda dei Quattro ("*Gang of four*", Go4).
- ➡ I design patterns da essi definiti adoperano un approccio di documentazione uniforme

# The Gang of Four



Gamma, Helm, Johnson, Vlissides at OOPSLA 1994

# Evoluzione dei Design Patterns

<b>Christopher Alexander</b> <i>The Timeless Way of Building</i> <i>A Pattern Language: Towns, Buildings, Construction</i>	Architecture	1970'
<b>Gang of Four (GoF)</b> <i>Design Patterns: Elements of Reusable Object-Oriented Software</i> 	Object Oriented Software Design	1995'
<b>Many Authors</b>	Other Areas: Middleware, HCI, Management, Education, ...	2000'

# Catalogo dei Design Patterns

☞ La ricerca del pattern da utilizzare è semplificata dall' utilizzo di un **catalogo** di pattern:

GoF, Design Patterns – Elements of Reusable Object-Oriented Software

Il volume descrive 23 pattern in un formato consistente per semplificarne la ricerca e l' apprendimento.

Pattern name and classification;

Also Known As;

Applicability;

Participants;

Consequences;

Sample Code;

Related Patterns.

Intent;

Motivation;

Structure;

Collaborations;

Implementation;

Known Uses;

# Classificazione – 1/3

☞ I design pattern possono essere classificati in base a due criteri:  
Scopo (***purpose***), che riflette cosa fa il pattern:

- *creational*: astraggono il processo di creazione (istanziamento) di oggetti;
- *structural*: trattano la composizione delle classi o oggetti per formare strutture più complesse;
- *behavioral*: si occupano del modo (algoritmi) in cui classi o oggetti interagiscono reciprocamente e distribuiscono fra loro le responsabilità;

Ambito (***scope***), specifica se il pattern è relativo a classi o ad oggetti:

- *class*: trattano la relazioni statiche, determinate a tempo di compilazione, tra classi e sottoclassi;
- *object*: trattano relazioni dinamiche, che variano a tempo di esecuzione, tra oggetti.

# Classificazione – 2/3

- ☞ Nei design pattern **creazionali** relativi a classi, la creazione di oggetti è affidata a sottoclassi mentre nei design pattern relativi a oggetti tale responsabilità è affidata ad altri oggetti.
- ☞ Nei pattern **strutturali** relativi a classi, la composizione di classi è realizzata attraverso l'ereditarietà, mentre i pattern strutturali relativi a oggetti descrivono come assemblare oggetti.
- ☞ I pattern **comportamentali** relativi a classi utilizzano l'ereditarietà per descrivere algoritmi e controllo di flusso, mentre i pattern comportamentali relativi a oggetti descrivono come cooperano gli oggetti per eseguire un compito che non può essere portato a termine da un singolo oggetto.

# Classificazione – 3/3

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

# Pattern di progettazione

---

## Design pattern creazionali Esempi

# Design pattern creazionali

- I design pattern creazionali astraggono il processo di istanziazione.
- Consentono di rendere il sistema indipendente da come gli oggetti sono creati, rappresentati e delle relazioni di composizione tra essi.
- Se basati su classi, utilizzano l' ereditarietà per modificare la classe istanziata.
- Se basati su oggetti, delegano l' istanziazione ad altri oggetti.
- Incapsulano la conoscenza relativa alle classi concrete utilizzate dal sistema.
- Nascondono come le istanze delle classi sono create e assemblate.

# Design pattern creazionali

☞ Alcuni *creational patterns* catalogati:

- **Abstract Factory**: un oggetto che serve a creare istanze di altri oggetti
- **Factory Method**: un oggetto che serve a creare istanze di diverse classi derivate
- **Prototype**: istanza completa di un oggetto che serve per essere clonato o copiato
- **Singleton**: un oggetto che restituisce una sola istanza di se stesso

# Pattern creazionali

☞ Esempio di *creational patterns* illustrato nel seguito:

- Singleton
- Factory Method
- Abstract Factory

# Singleton – Scopo

- ☞ Assicura che una classe abbia una sola istanza e fornisce un punto globale di accesso ad essa.

# Singleton – Motivazione

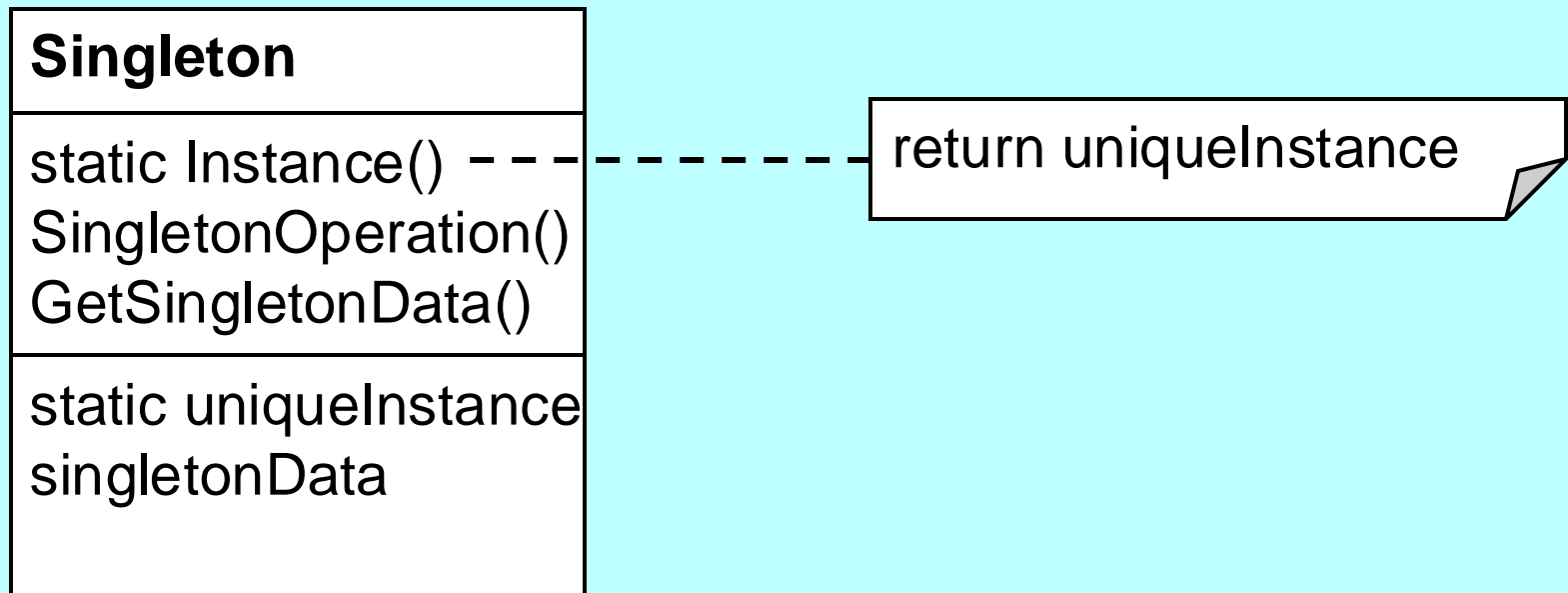
- ☞ In alcuni casi è importante che una classe abbia esattamente un'istanza.
- ☞ Una variabile esterna fa sì che l'oggetto sia accessibile, ma non impedisce che siano istanziati più oggetti.
- ☞ La classe stessa è responsabile di avere traccia della sua unica istanza e fornire un modo per accedere a tale istanza.

# Singleton – Applicabilità

☞ Il Singleton va usato quando:

- deve esserci esattamente una istanza di una classe e deve essere accessibile ai clients da un punto di accesso globale;
- l'unica istanza deve essere estensibile con subclassing e i clients devono essere capaci di usare una istanza estesa senza modificare il loro codice.

# Singleton – Struttura



# Singleton – Partecipanti



## Singleton

- Definisce un'operazione `Instance()` che consente ai clients di accedere alla sua unica istanza;
- `Instance()` è un metodo di classe (i.e. static);
- Può essere responsabile di creare la sua propria unica istanza.

# Singleton – Conseguenze

- ☞ Accesso controllato all'unica istanza.
- ☞ Un sostituto elegante per variabili esterne (consente di evitare l'“inquinamento” del name space con numerose variabili esterne).
- ☞ Il Singleton può essere esteso per consentire un numero variabile di istanze.

# Singleton – Implementazione 1/3

- ☞ Una tecnica per garantire che vi sia una sola istanza della classe consiste nel nascondere l'operazione di creazione in una operazione di classe (funzione *static*) e rendendo il costruttore *protected*.

# Singleton – Implementazione 2/3

```
class Singleton {  
    public:  
        static Singleton* Instance();  
    protected:  
        Singleton();  
    private:  
        static Singleton* _instance;  
};  
  
Singleton* Singleton::_instance = 0;  
  
Singleton* Singleton::Instance () {  
    if (_instance == 0) {  
        _instance = new Singleton;  
    }  
    return _instance;  
}
```

# Singleton – Implementazione 3/3

- ☞ Nel caso di sottoclassi, occorre fare in modo che l' unica istanza della classe derivata sia accessibile in modo che i client possano utilizzarla.
- ☞ Una possibile soluzione nel determinare il Singleton che si intende utilizzare nell' operazione Instance():

```
Singleton* Singleton::Instance() {  
    if(condition)  
        _instance= new particularInstance;  
}
```

# Factory Method – Scopo

- ➡ Offrire una interfaccia per creare un oggetto, lasciando che le sottoclassi decidano quale classe istanziare.
- ➡ Un Factory Method permette ad una classe di affidare l'istanziamento alle sottoclassi

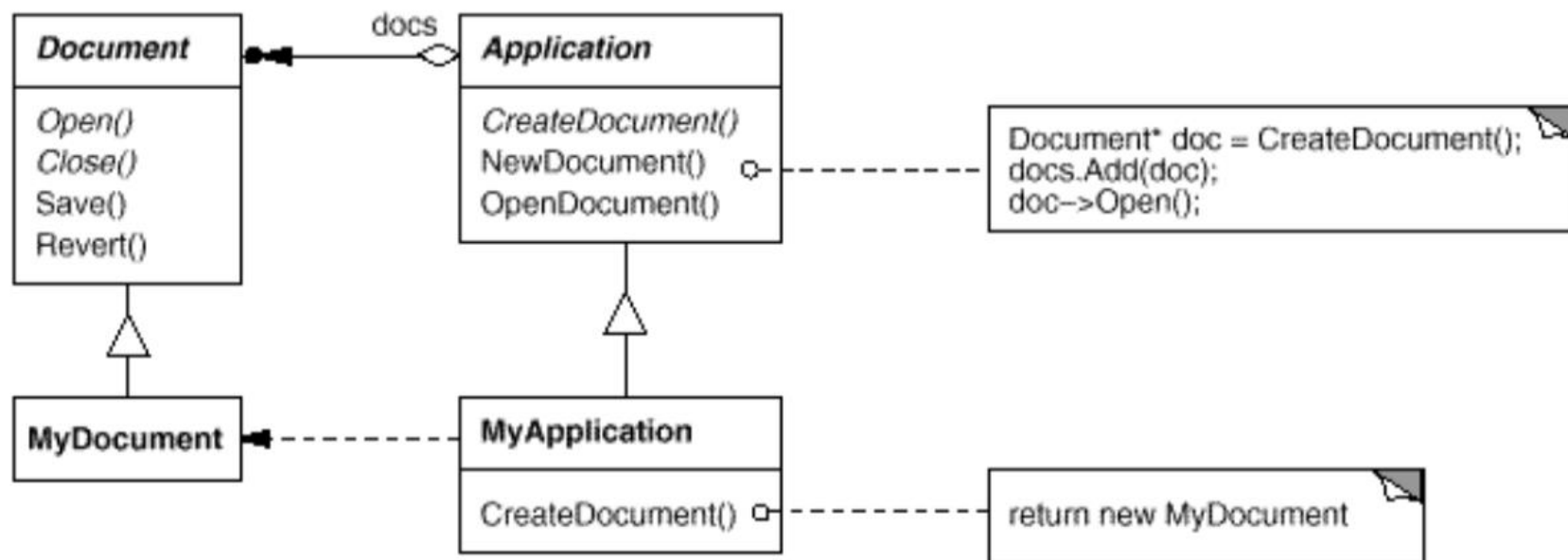
# Factory Method – Motivazione

- ☞ Si consideri un framework che manipola in maniera astratta documenti, Le due astrazioni chiave sono *l'Applicazione* e il *Documento*, entrambe sono astratte e i client devono creare sottoclassi per realizzare l'implementazione specifica.
- ☞ Poiché la sottoclasse di documento da istanziare è application-specific, l'Applicazione **sa quando** un documento deve essere creato, **ma non sa quale tipo** creare.
- ☞ Un FactoryMethod incapsula la conoscenza riguardo quale tipo di sottoclasse di documento creare, svincolandolo dal framework.

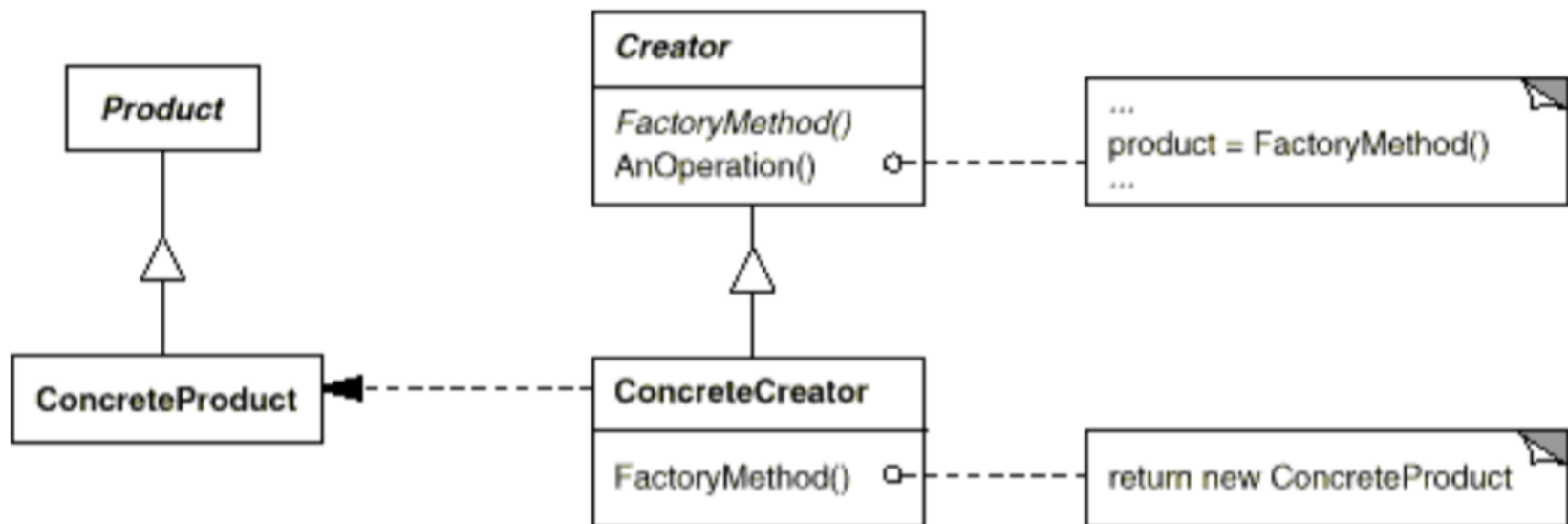
# Factory Method – Applicabilità

- ☞ Il factory method va usato quando:
- Una classe non può sapere quale tipo di classi di oggetti deve creare.
  - Una classe vuole lasciar decidere alle sottoclassi quali oggetti creare.

# Factory Method – Struttura (1/2)



# Factory Method – Struttura (2/2)

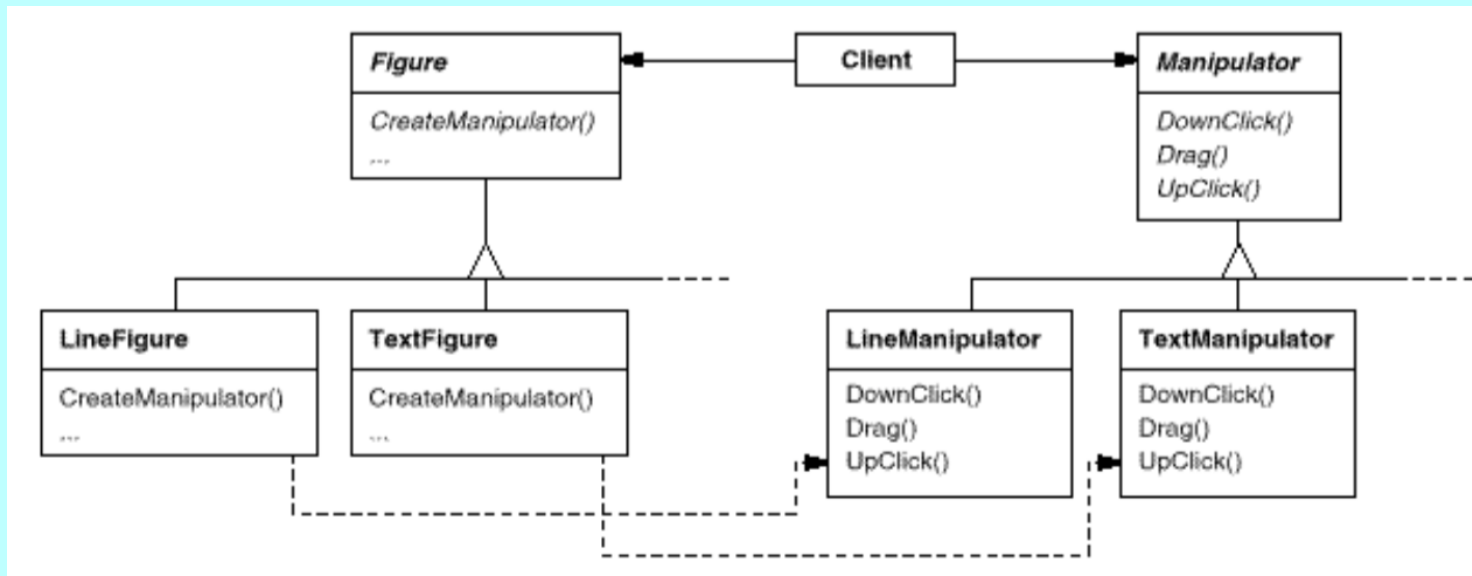


# Factory Method – Partecipanti

- ➡ Product (Document)  
Definisce l'interfaccia di oggetti che il metodo factory crea
- ➡ ConcreteProduct (MyDocument)  
Implementa l'interfaccia dell'oggetto Product
- ➡ Creator (Application)  
Dichiara il metodo factory, che restituisce un oggetto di tipo Product. Esso può anche definire una implementazione di *default*
- ➡ ConcreteCreator (MyApplication)  
Effettua l'override del factory method per restituire una istanza di un ConcreteProduct

# Factory Method – Conseguenze

- ➡ Rimuove il bisogno di legare classi application-specific al codice ma forza a creare sottoclassi anche quando non ce ne sia il bisogno.
- ➡ Fornisce agganci per creare oggetti (metodi create), in un modo più flessibile rispetto a creare oggetti direttamente, favorendo la ridefinizione.
- ➡ Usando più Factory Method, un client può connettere *gerarchie di classi parallele* (quando una classe delega parte delle sue responsabilità a classi separate. Es. il *Document* ha il suo *DocumentFormatter*)



# Factory Method – Implementazione

- ☞ La classe Creator può essere astratta, oppure può definire una implementazione di default.
- ☞ Si può parametrizzare il tipo dell'oggetto creato, in modo da favorire l'estensibilità del factory method.

```
class Creator {  
    public:  
        virtual Product* Create(ProductId);  
};  
Product* Creator::Create (ProductId id) {  
    if (id == MINE)    return new MyProduct;  
    if (id == YOURS)  return new YourProduct;  
    // repeat for remaining products...  
    return 0;  
}
```

# Abstract Factory – Scopo

- ☞ Offrire una interfaccia per creare famiglie di oggetti in relazione o dipendenti, senza specificare le loro classi concrete.

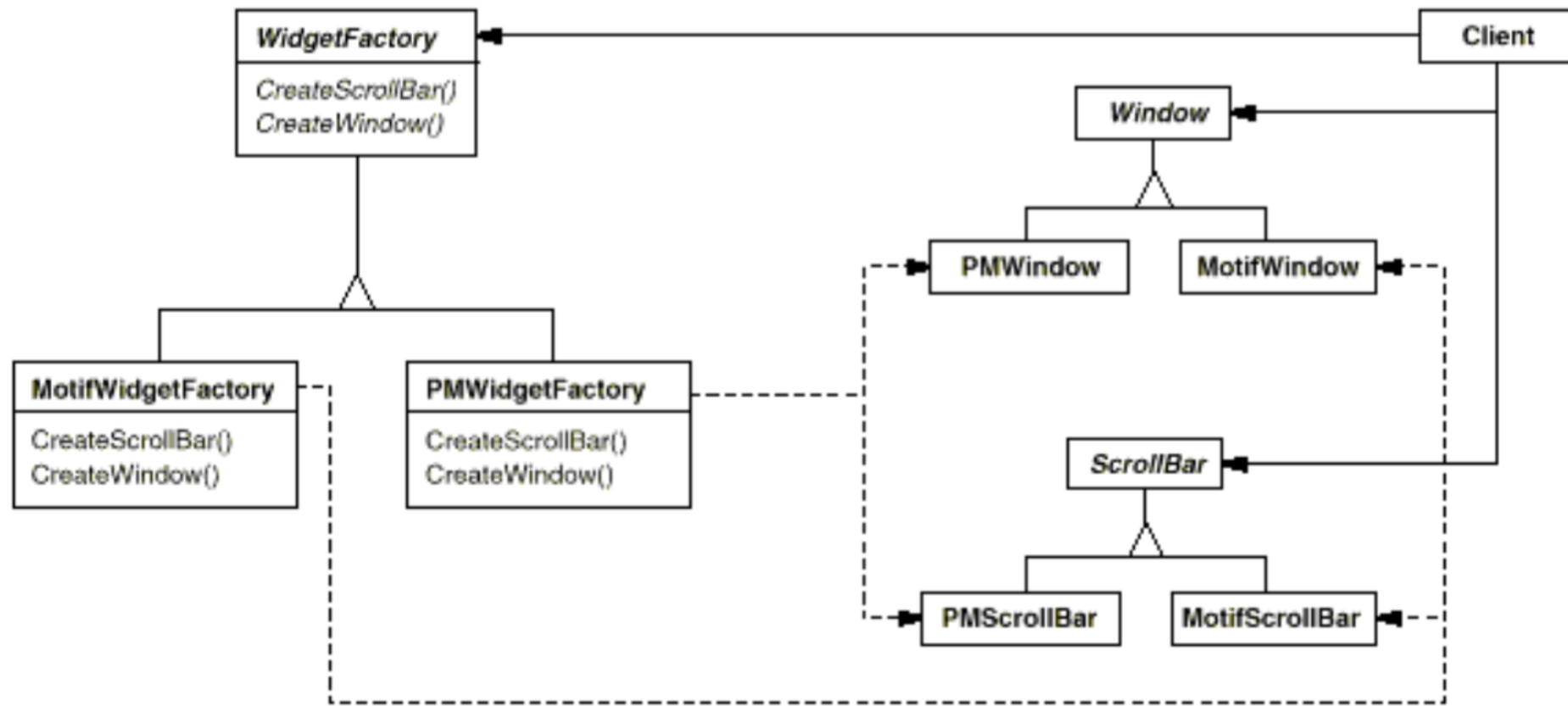
# Abstract Factory – Motivazione

- ☞ Ci sono casi in cui vogliamo creare differenti elementi tra loro in relazione senza legarci alle loro istanze concrete
- ☞ Ad esempio, un toolkit di interfacce grafiche deve supportare più standard look-and-feel, che definiscono l'estetica e il comportamento dei “widget” (finestre, scrollbar, pulsanti, etc.)
- ☞ I client usano una WidgetFactory astratta, che dichiara una interfaccia per creare ogni tipo base di widget, definiti a loro volta come classi astratte. Quindi, esiste una sottoclasse concreta che implementa ogni tipo di widget per uno specifico look-and-feel.

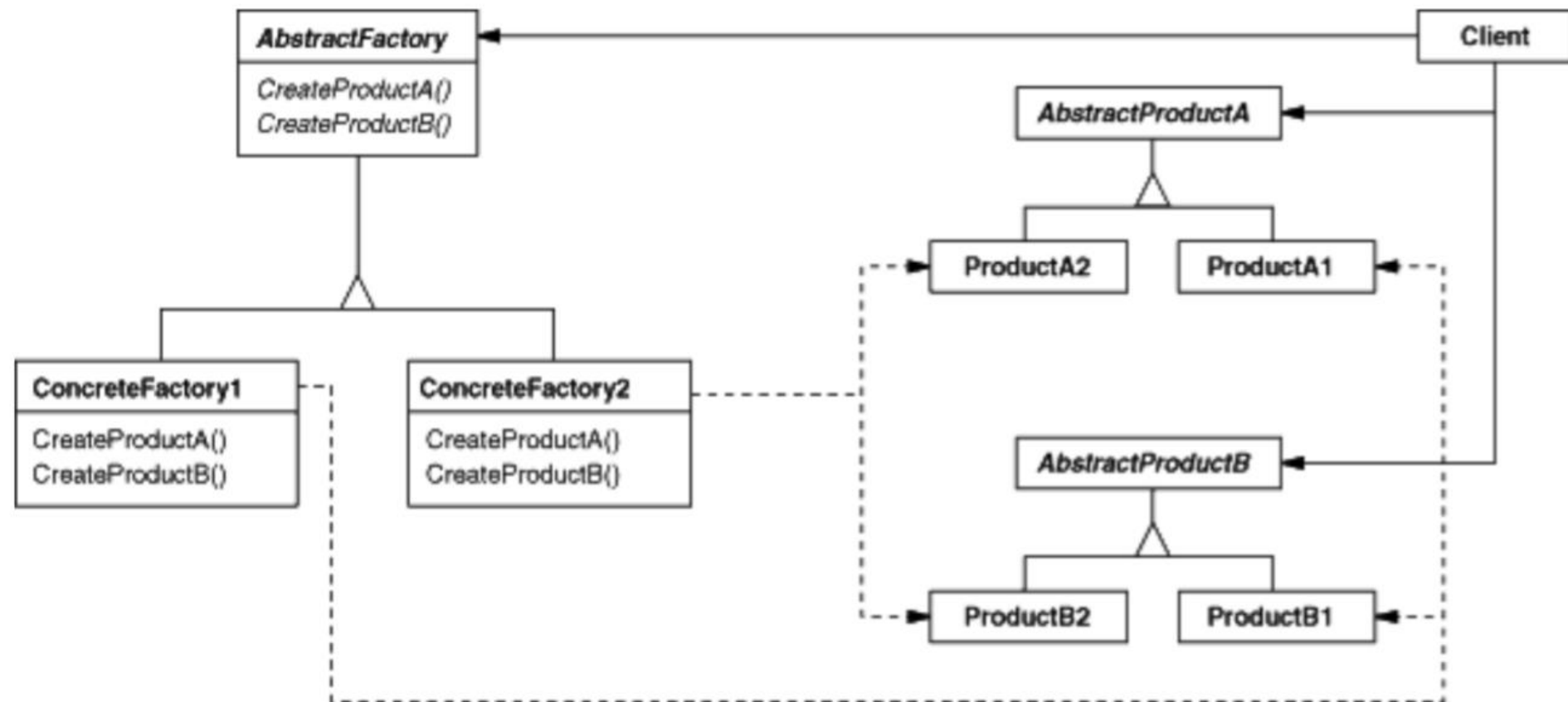
# Abstract Factory – Applicabilità

- ☞ L'abstract factory va usato quando:
- Un sistema deve essere indipendente da come i suoi prodotti sono creati, composti e rappresentati.
  - Un sistema deve essere configurato per più *famiglie di prodotti*
  - Una famiglia di prodotti è progettata per servirsi di un solo insieme di classi per volta, ed occorre garantire questo vincolo
  - Occorre offrire una libreria di prodotti, ma si vuole esporre solo le loro interfacce e non le loro implementazioni.

# Abstract Factory – Struttura (1/2)



# Abstract Factory – Struttura (2/2)



# Abstract Factory – Partecipanti

- ➡ AbstractFactory (WidgetFactory)  
Dichiara una interfaccia per le operazioni che creano oggetti astratti
- ➡ ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)  
Implementa le operazioni per creare oggetti concreti
- ➡ AbstractProduct (Window, ScrollBar)  
Dichiara una interfaccia per un tipo di oggetto prodotto
- ➡ ConcreteProduct (MotifWindow, MotifScrollBar)  
Definisce un oggetto prodotto che deve essere creato da la corrispondente concrete factory
- ➡ Client  
Usa solo interfacce dichiarate da AbstractFactory e AbstractProduct

# Abstract Factory – Conseguenze

- ☞ Isola le classi concrete: aiuta a controllare le classi di oggetti che una applicazione crea, isolandole nelle concrete factory piuttosto che nel codice del client.
- ☞ Rende semplice cambiare famiglie di prodotti scegliendo una concrete factory.
- ☞ Promuove la coerenza tra gli oggetti creati quando gli oggetti prodotti in una famiglia sono progettati per lavorare insieme.
- ☞ Supportare nuovi tipi di prodotti è difficile, perché necessita di cambiare la Factory Interface e tutte le sottoclassi.

# Abstract Factory – Implementazione

- ☞ Siccome una applicazione richiede solamente una istanza di ConcreteFactory, si può usare il pattern **Singleton**.
- ☞ AbstractFactory dichiara solo interfacce per creare prodotti, ed è compito delle sottoclassi crearli: si può usare il **Factory Method**.
- ☞ Si può parametrizzare il tipo degli oggetti creati in una abstract factory, in modo da favorire l'estensibilità.

# Abstract Factory – Code 1/3

```
/* An AbstractFactory */  
class MazeFactory {  
    public:  
        MazeFactory();  
        virtual Maze* MakeMaze() const  
            { return new Maze; }  
        virtual Wall* MakeWall() const  
            { return new Wall; }  
        virtual Room* MakeRoom(int n) const  
            { return new Room(n); }  
        virtual Door* MakeDoor(Room* r1, Room* r2) const  
            { return new Door(r1, r2); }  
};
```

# Abstract Factory – Code 2/3

```
/* A client: MazeGame */
Maze* MazeGame::CreateMaze (MazeFactory& factory)
{
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);
    aMaze->AddRoom(r1);
    . . . . .
}
```

# Abstract Factory – Code 3/3

```
/* A ConcreteFactory */  
  
class EnchantedMazeFactory : public MazeFactory {  
    public:  
        EnchantedMazeFactory();  
        virtual Room* MakeRoom(int n)    const  
            { return new EnchantedRoom(n, CastSpell()); }  
        virtual Door* MakeDoor(Room* r1, Room* r2) const  
            { return new DoorNeedingSpell(r1, r2); }  
    protected:  
        Spell* CastSpell() const;  
};
```

# Builder: Overview

## ☞ Intent

- Separare la costruzione di un oggetto complesso dalla sua rappresentazione così che lo stesso processo di costruzione può creare differenti rappresentazioni

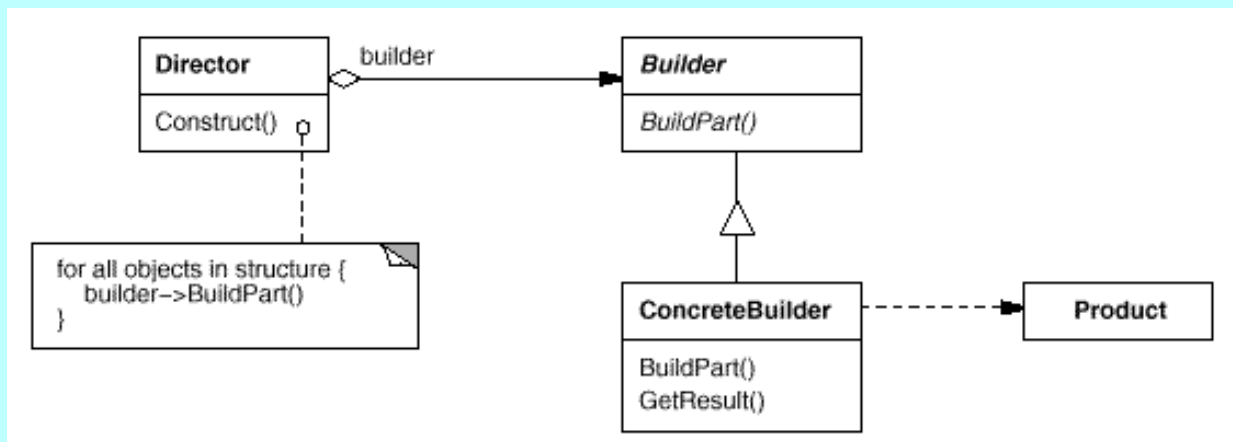
## ☞ Esempio: una fabbrica di automobili

- I robots costruiscono ogni parte della macchina
- Le parti vengono aggiunte alla macchina in costruzione



# Builder: Participants

- ➡ **Builder**  
Specifica un'interfaccia astratta per creare parti di un oggetto Product
- ➡ **ConcreteBuilder**  
Costruisce e assembla le parti di Product implementando l'interfaccia Builder
- ➡ **Director**  
Costruisce un oggetto usando l'interfaccia Builder
- ➡ **Product**  
Rappresenta l'oggetto complesso in costruzione



# Builder: Collaborations

- ☞ Il Client crea l'oggetto Director a lo configura con un Builder
- ☞ Il Director notifica il Builder di costruire ogni parte del prodotto
- ☞ Il Builder gestisce le richieste del Director e aggiunge parti al prodotto
- ☞ Il Client recupera il prodotto dal Builder

# Builder: Applicability

- ☞ Usiamo il pattern Builder quando:
- L' algoritmo per creare un oggetto complesso dovrebbe essere indipendente dalle parti che costituiscono l'oggetto e da come sono assemblate
  - Il procedimento di costruzione deve consentire differenti rappresentazione per l'oggetto che viene costruito
  - Il processo di costruzione può essere separato in passi discreti (differenza tra Builder e Abstract Factory)

# Builder: Consequences

---

- ☞ Consente di variare la rappresentazione interna di un prodotto usando differenti Builders
- ☞ Isola il codice per la costruzione e la rappresentazione
- ☞ Consente un controllo a grana fine sul processo di costruzione

# Builder: Implementation

☞ Argomenti da considerare

- Per ogni Product è necessaria una classe astratta ?
  - Solitamente Products non hanno un' interfaccia comune
- Solitamente c'è una classe Builder astratta che definisce un'operazione per ogni componente che un Director può chiedere di creare
  - Queste operazioni non fanno nulla per default
  - Il ConcreteBuilder effettua l' override delle operazioni selettivamente

# Pattern di progettazione

---

## Design pattern strutturali Esempi

# Pattern strutturali

- I pattern strutturali sono relativi a come classi e oggetti sono **composti** per formare strutture più grandi.
- I design pattern strutturali basati su **classi** utilizzano l'ereditarietà per generare classi che combinano le proprietà di classi base.
- I design pattern strutturali basati su **oggetti** mostrano come comporre oggetti per realizzare nuove funzionalità. Danno flessibilità alla composizione che viene modificata a run-time, cosa impossibile con la composizione statica (con classi).

# Design pattern strutturali



Alcuni *structural patterns* catalogati:

- **Adapter:** Unisce le interfacce di differenti classi
- **Bridge:** separa l'interfaccia di un oggetto dalla sua implementazione
- **Composite:** una struttura ad albero per rappresentare oggetti complessi
- **Decorator:** aggiunge funzionalità (responsabilità) ad oggetti dinamicamente
- **Façade:** una singola classe che rappresenta un'intero sistema
- **Proxy:** un oggetto che rappresenta un altro oggetto

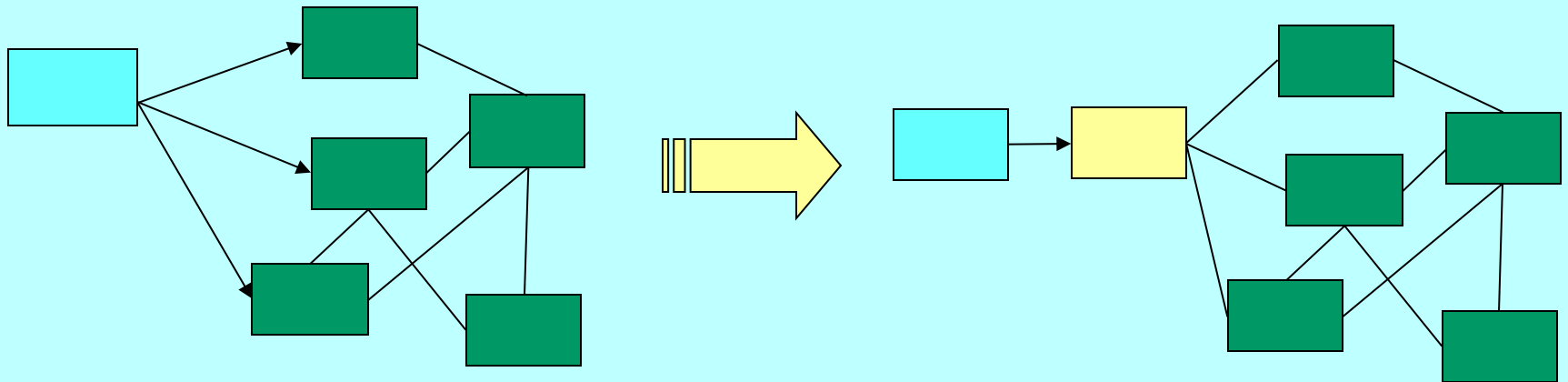
# Design pattern strutturali

☞ Esempi di *structural patterns* illustrati in dettaglio nel seguito:

- Façade;
- Adapter;
- Decorator.

# Façade – Scopo

- ➡ Rendere più semplice l'uso di un (sotto)sistema.
- ➡ Fornire un'unica interfaccia per un insieme di funzionalità “sparse” su più interfacce/classi.



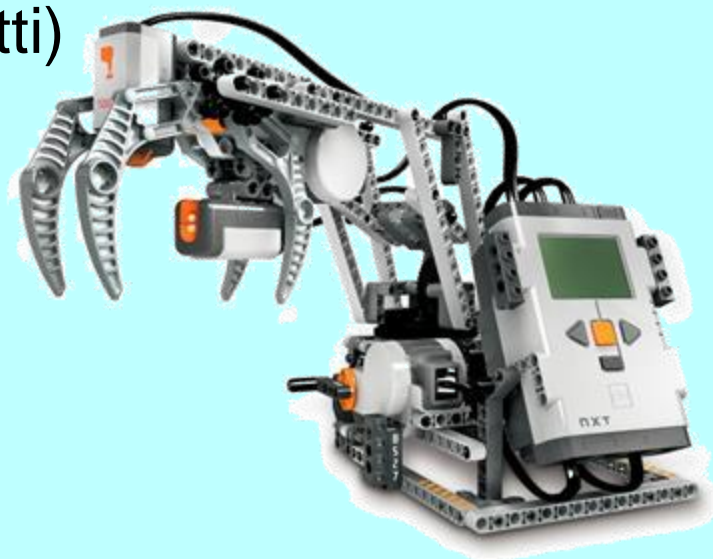
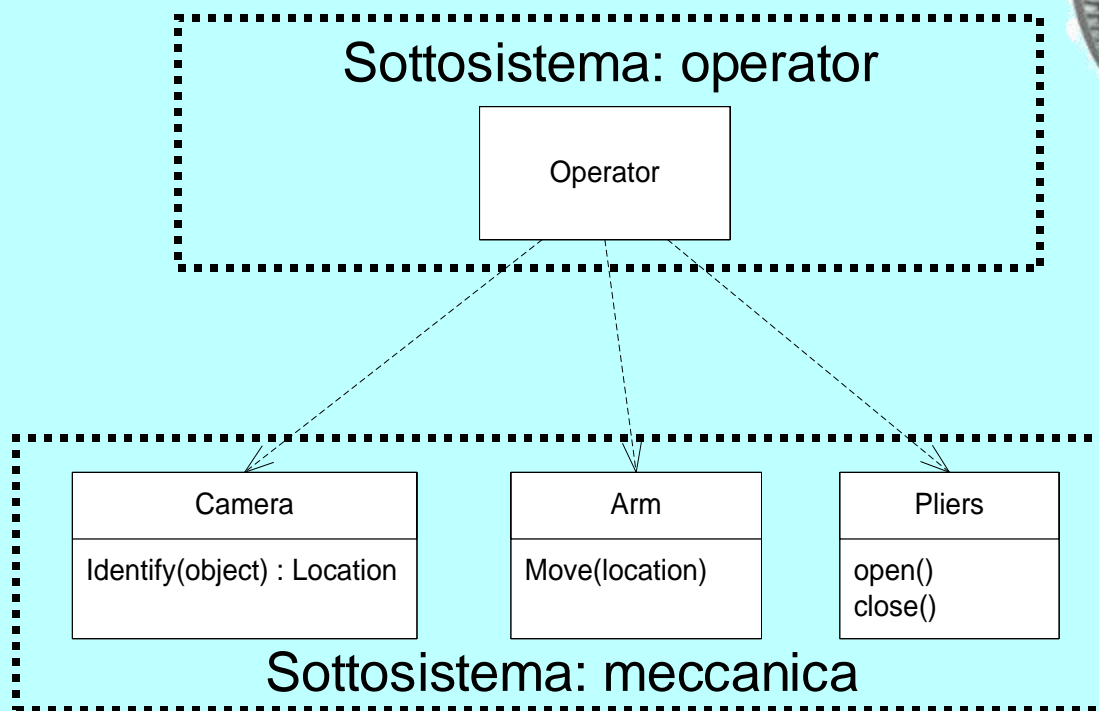
# Façade – Motivazione 1/4

- ☞ La suddivisione di un sistema in sottosistemi aiuta a ridurre la complessità.
- ☞ Tuttavia, occorre diminuire comunicazione e dipendenze tra i sottosistemi.
- ☞ L' utilizzo di un oggetto façade fornisce un' unica e semplice interfaccia alle funzionalità del sottosistema.

# Façade – Motivazione 2/4

☞ Robot con quattro classi:

- Camera (per identificare gli oggetti)
- Arm (mobile)
- Pliers (per afferrare)



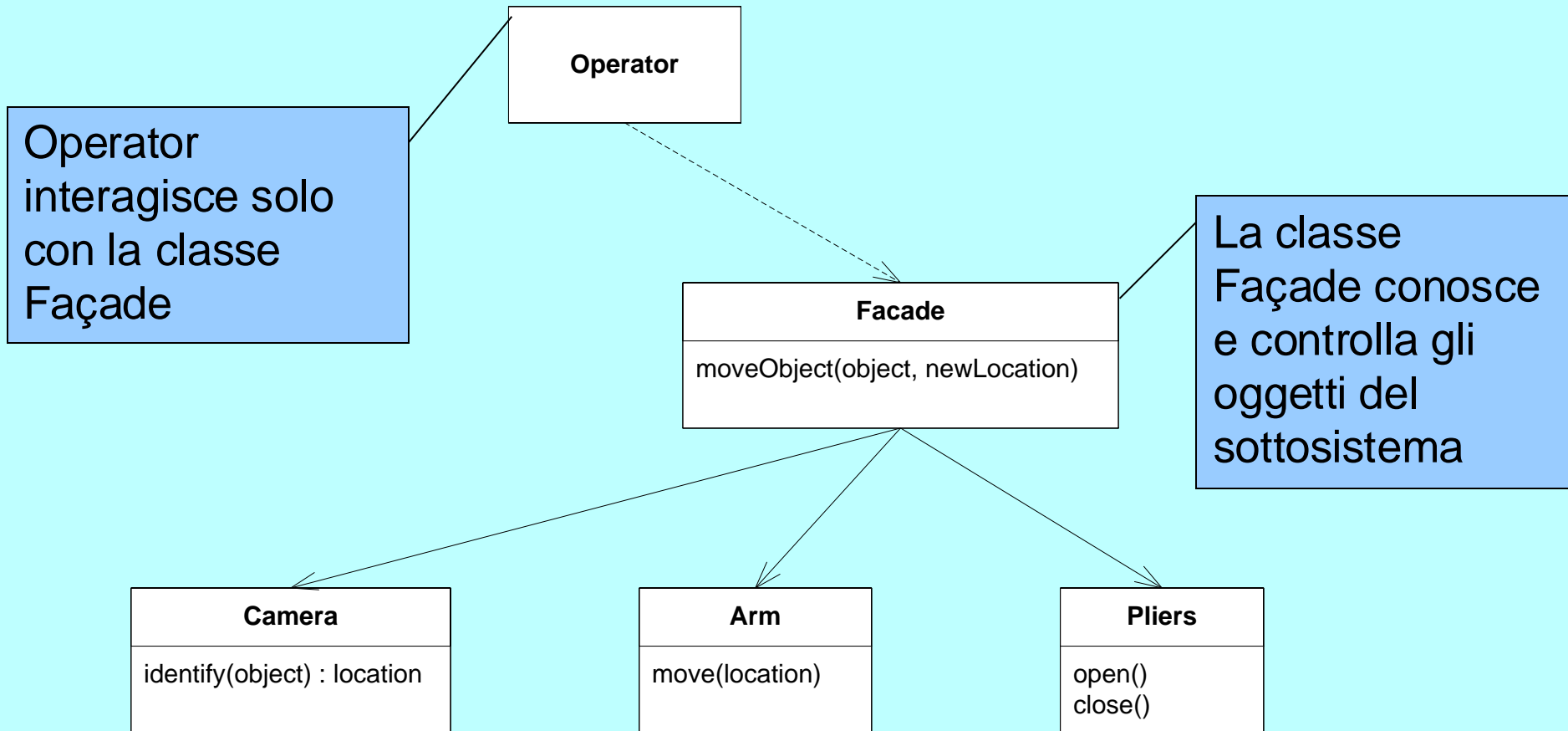
# Façade – Motivazione 3/4

- 👉 Quanto deve conoscere della meccanica l'operatore?
- 👉 Si supponga di voler individuare un oggetto e spostarlo in una locazione predefinita:

```
oldLocation = Camera.identify(object);  
Arm.move(oldLocation);  
Pliers.close();  
Arm.move(newLocation);  
Pliers.open();
```

- 👉 Problema: non c'è incapsulamento:
  - L'operatore deve conoscere la struttura ed il comportamento.

# Façade – Motivazione 4/4

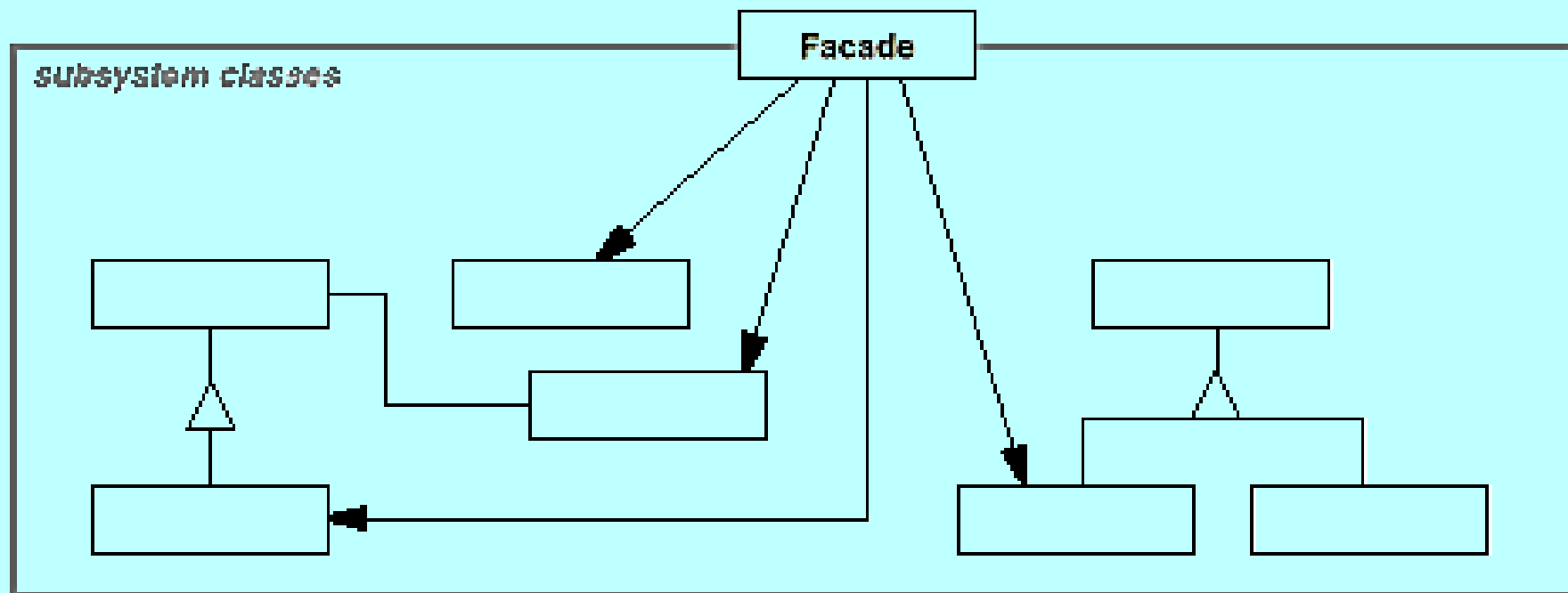


# Façade – Applicabilità

☞ Utilizzare il pattern Façade:

- Per fornire una vista semplice e di default di un sottosistema complesso;
- Nel caso di sottosistema stratificato, è possibile utilizzare façade come entry point per ciascun livello.

# Façade – Struttura



# Façade – Partecipanti

## Façade:

- Conosce quali classi di un sottosistema sono responsabili per una richiesta;
- Delega le richieste del client agli oggetti appropriati del sottosistema.

## Classi del sottosistema:

- Implementano le funzionalità del sottosistema;
- Gestiscono il lavoro assegnato da Façade;
- Non hanno alcun riferimento della facciata.

# Façade – Esempi

- Classi per disegno 3D per disegnare in 2D;
- Compilatore:
  - Classi Parser, Scanner, Token, SyntacticTree, CodeGenerator, ecc....;
  - Classe Compiler con metodo compile().

# Façade – Conseguenze

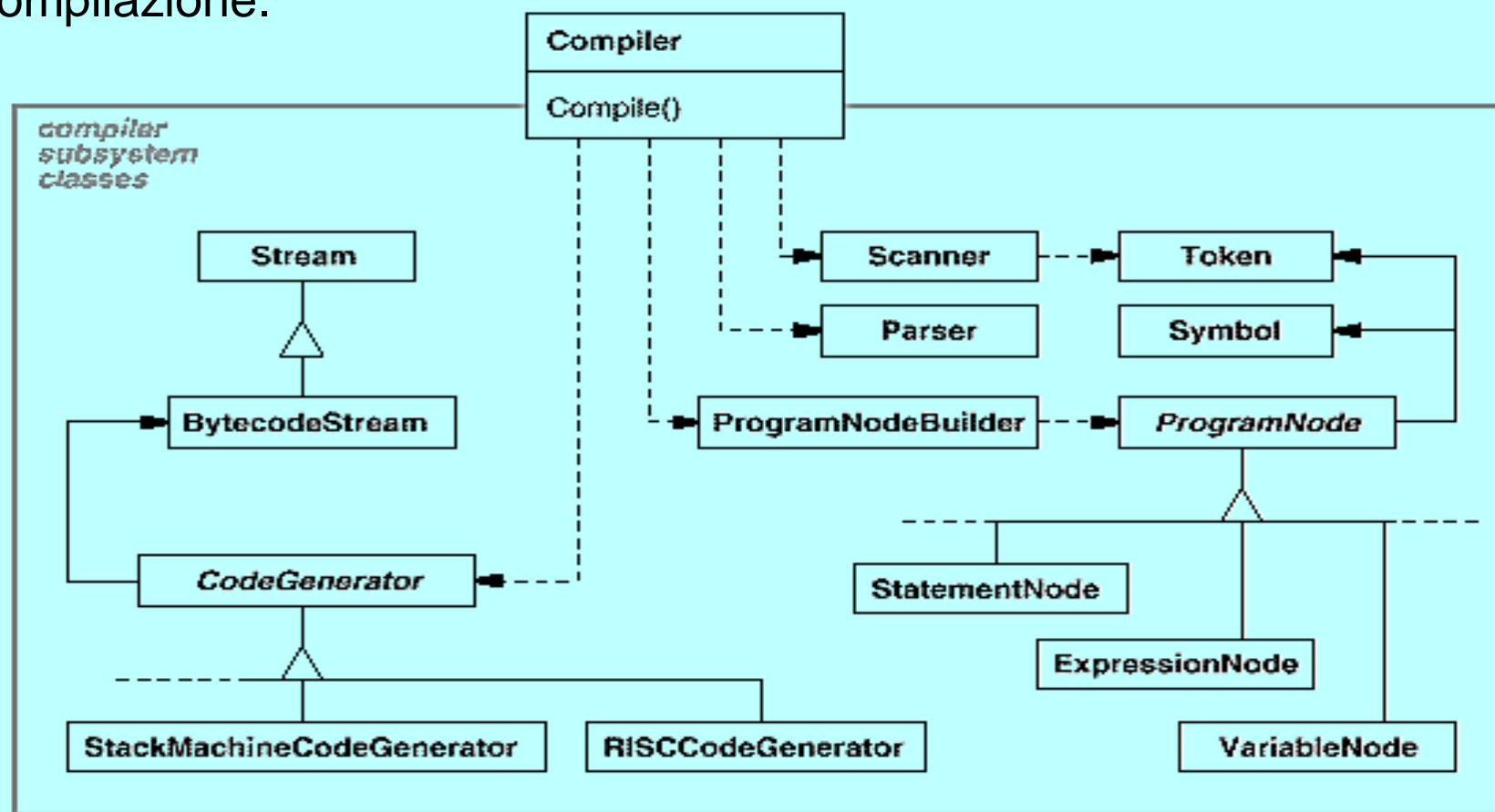
- ➡ Promuove un accoppiamento debole fra cliente e sottosistema.
- ➡ Nasconde al cliente le componenti del sottosistema.
- ➡ Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema.

# Façade – Implementazione

- ☞ L'accoppiamento tra i client ed il sottosistema può essere ridotto anche rendendo Façade una classe astratta con sottoclassi concrete per diverse implementazioni di un sottosistema.
  - In tal caso, i client possono comunicare con il sottosistema attraverso l'interfaccia della classe astratta Façade.

# Façade – Esempio 1/8

Si consideri il class diagram seguente relativo ad un ambiente di programmazione che da alle applicazioni accesso al sottosistema per la compilazione.



# Façade – Esempio 2/8

- ☞ Il sottosistema di compilazione definisce una classe `BytecodeStream` che implementa un flusso di oggetti `Bytecode`. Ciascun oggetto `Bytecode` incapsula un bytecode che specifica le istruzioni macchina.
- ☞ La classe `Scanner` del sottosistema riceve un flusso di caratteri e genera un flusso di token, un token per volta.
- ☞ Il `Parser` effettua l'analisi dei token generati dallo `Scanner` e genera un parse tree utilizzando un `ProgramNodeBuilder`.
- ☞ Il parse tree è fatto di istanze di sottoclassi di `ProgramNode`.
- ☞ `ProgramNode` utilizza un oggetto `CodeGenerator` per generare codice macchina nella forma di oggetti `Bytecode` in un flusso `BytecodeStream`.
- ☞ La classe `Compiler` è una “facciata” che mette tutti i pezzi insieme fornendo una semplice interfaccia per la compilazione del codice sorgente e la generazione del codice per una particolare macchina.

# Façade – Esempio 3/8

```
class Scanner {  
    public:  
        Scanner(istream&);  
        virtual ~Scanner();  
        virtual Token& Scan();  
    private:  
        istream& _inputStream;  
};
```

```
class Parser {  
    public:  
        Parser();  
        virtual ~Parser();  
        virtual void Parse(Scanner&, ProgramNodeBuilder&);  
};
```

# Façade – Esempio 4/8

```
class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();
    virtual ProgramNode* NewVariable(
        const char* variableName) const;
    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression) const;
    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value) const;
    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart) const;
    ...
    ProgramNode* GetRootNode();
private:
    ProgramNode* _node;
};
```

# Façade – Esempio 5/8

👉 Il metodo `Traverse` genera il codice macchina.

```
class ProgramNode {
public:
    //program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
    //...
    //child manipulation
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    //...
    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

# Façade – Esempio 6/8

```
class CodeGenerator {  
    public:  
        virtual void Visit(StatementNode*);  
        virtual void Visit(ExpressionNode*);  
        //...  
    protected:  
        CodeGenerator(BytecodeStream&);  
    protected:  
        BytecodeStream& _output;  
};
```

# Façade – Esempio 7/8

☞ Ogni sottoclasse di ProgramNode (StatementNode, ExpressionNode, ...) implementa il metodo Traverse da richiamare sui propri oggetti figli.

```
void ExpressionNode::Traverse (CodeGenerator& cg) {  
    cg.Visit(this);  
    ListIterator i(_children);  
    for (i.First(); !i.IsDone(); i.Next()) {  
        i.CurrentItem()->Traverse(cg);  
    }  
}
```

# Façade – Esempio 8/8

```
class Compiler {
public:
    Compiler();
    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;
    parser.Parse(scanner, builder);
    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

# Adapter – Scopo

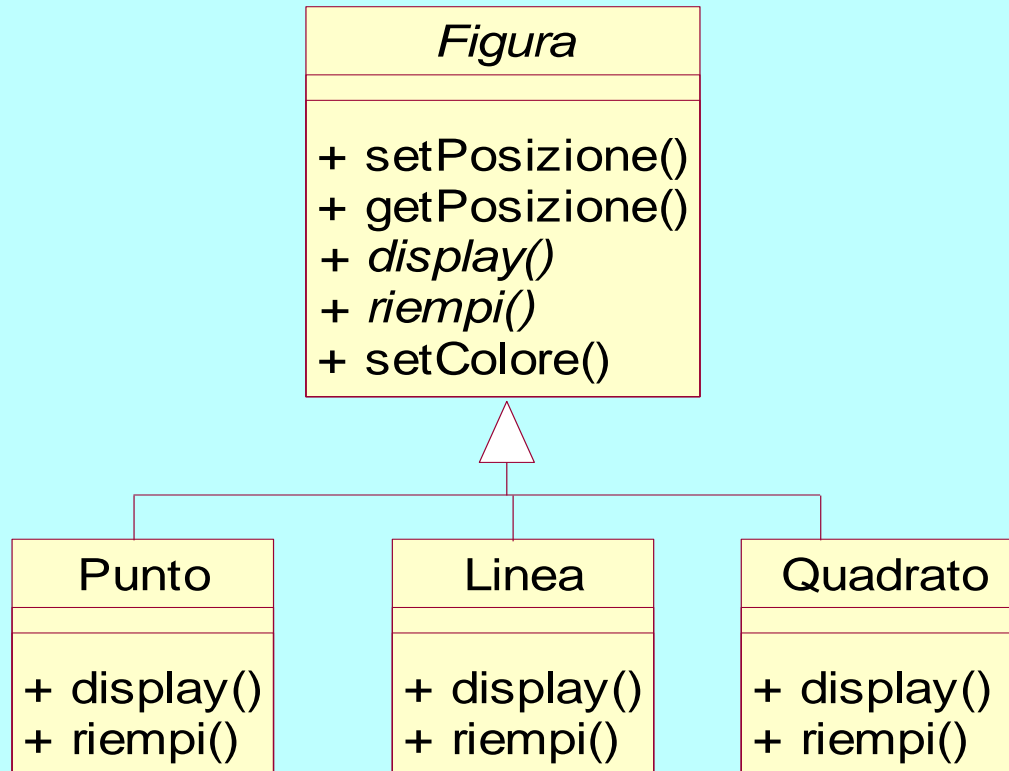
- ☞ Adattare l'interfaccia di una classe già pronta all'interfaccia che il cliente si aspetta
  - “Adapter” come l’ “adattatore” per prese di corrente.

# Adapter – Motivazione 1/4

- ☞ Talvolta una classe progettata per il riuso non è riusabile solo perché la sua interfaccia non coincide con quella del dominio specifico di un' applicazione.

# Adapter – Motivazione 2/4

- ☞ Si consideri un editor di disegno che fornisce l'interfaccia Figura i cui metodi riempi e display sono implementati dalle classi Punto, linea e Quadrato.

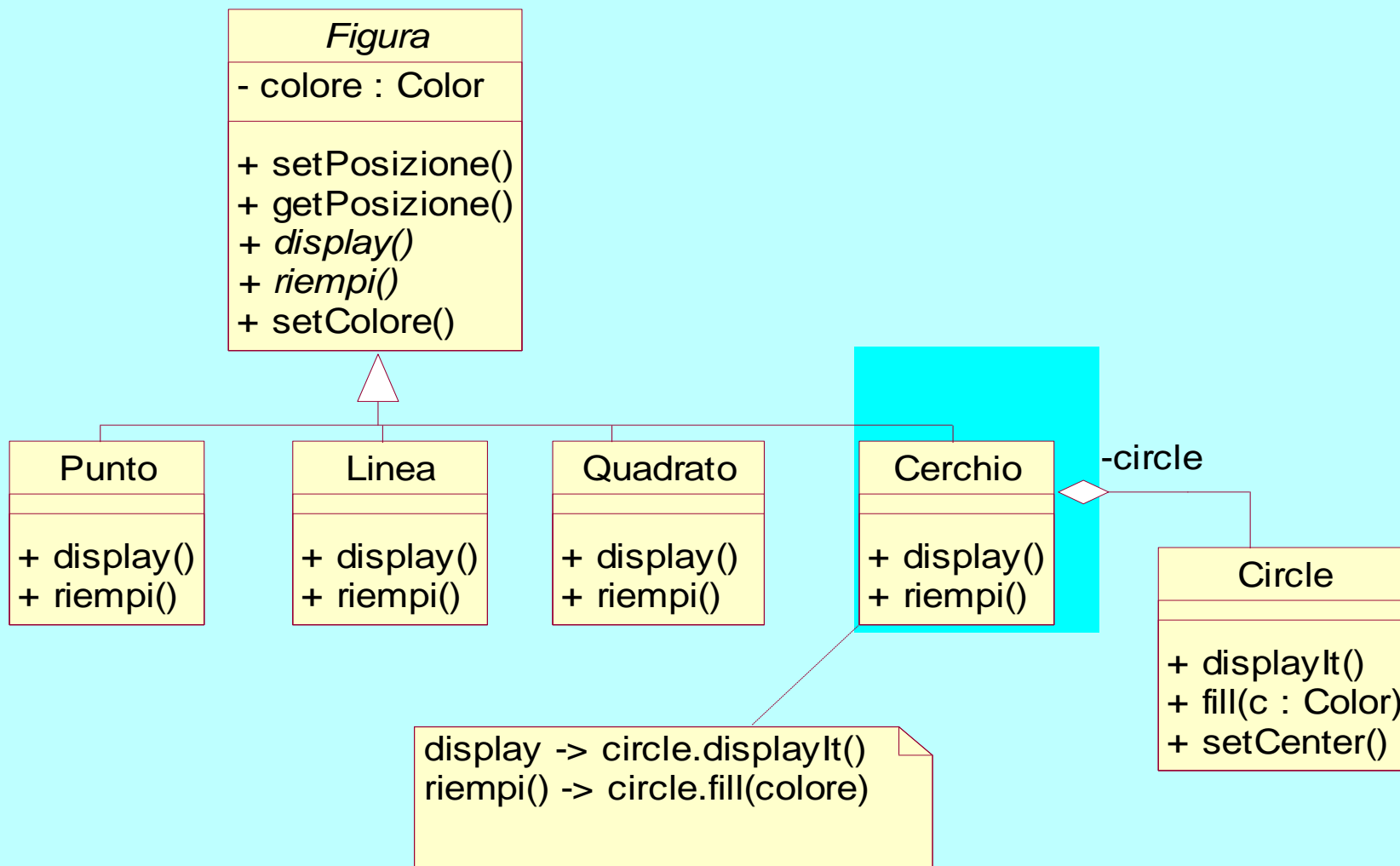


# Adapter – Motivazione 3/4

- ☞ Vogliamo aggiungere Cerchio
  - Lo implementiamo da zero?
    - Fatica inutile...
  - Usiamo la classe Circle?
    - Ha un'interfaccia diversa che non possiamo modificare.
- ☞ Creiamo un adattatore.

Circle
+ displayIt() + fill(c : Color) + setCenter()

# Adapter – Motivazione 4/4



# Adapter – Applicabilità

☞ Usiamo Adapter quando:

- Vogliamo utilizzare una classe esistente la cui interfaccia non risponde alle nostre necessità;
- Si vuole realizzare una classe riusabile che coopera con classi che non necessariamente hanno un' interfaccia compatibile;
- Occorre utilizzare sottoclassi esistenti le cui interfacce non possono essere adattate sottoclassando ciascuna di esse. Un object adapter può adattare l' interfaccia della classe base.

# Due tipi di Adapter

## Object Adapter

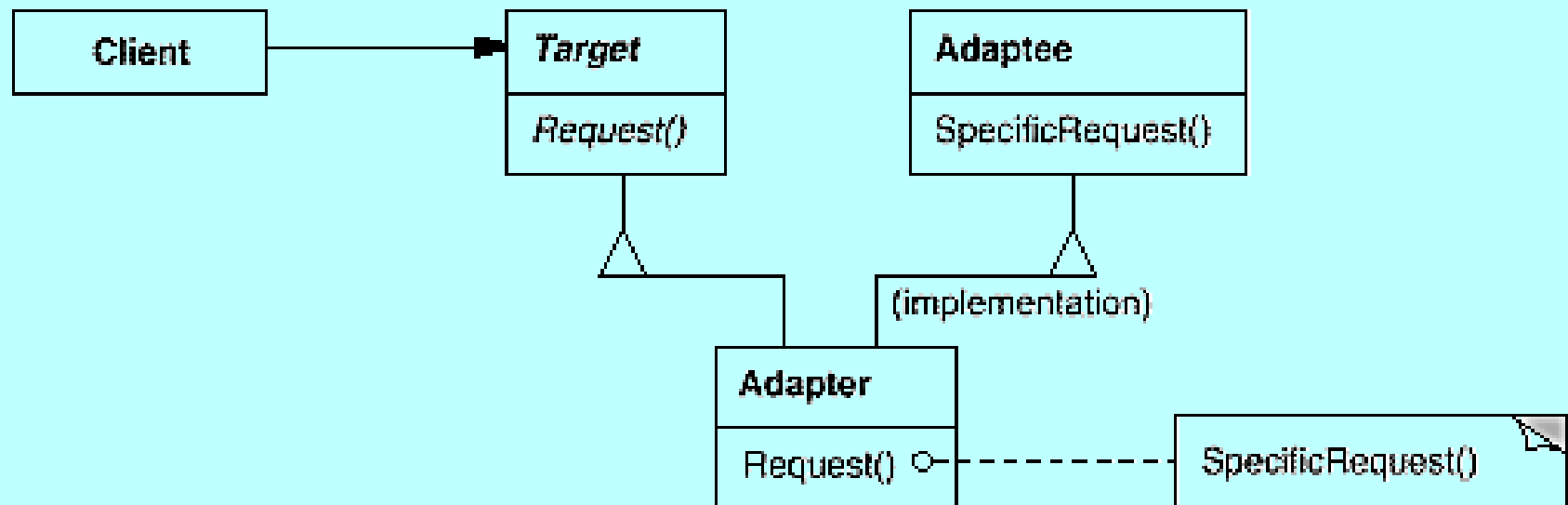
- Basato su delega/composizione.

## Class Adapter

- Basato su ereditarietà;
- L' adattatore eredita sia dall' interfaccia attesa sia dalla classe adattata;
- No eredità multipla: l' interfaccia attesa deve essere un' interfaccia, non una classe.

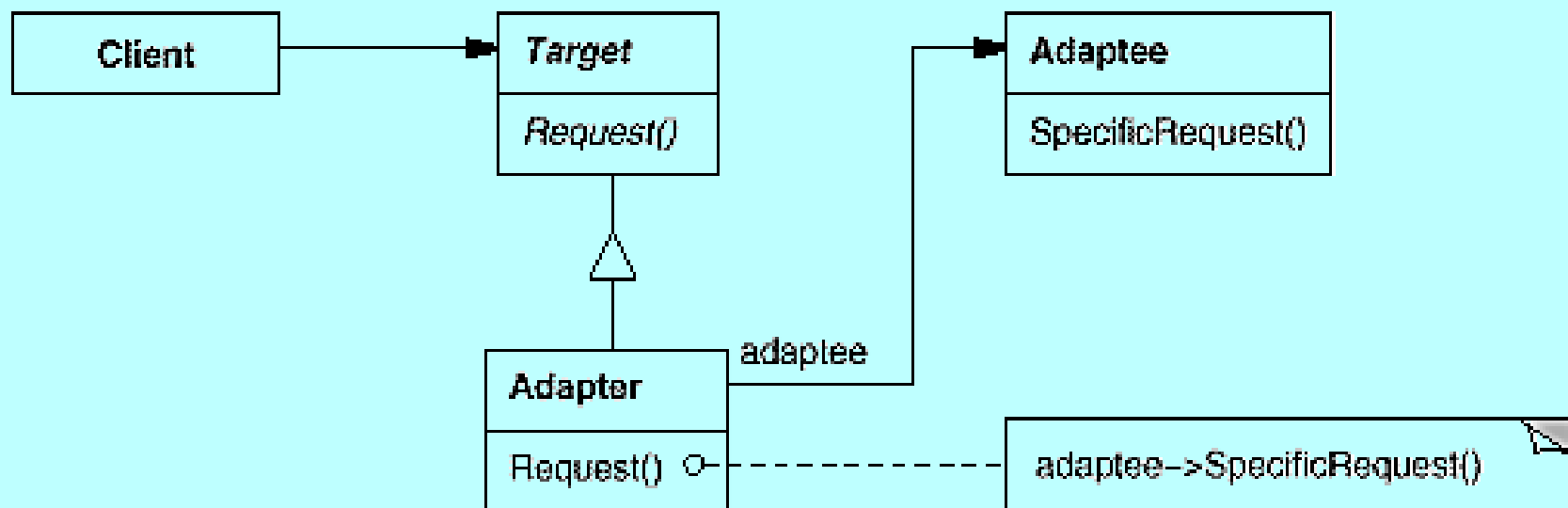
# Adapter – Struttura 1/2

## 👉 Class Adapter



# Adapter – Struttura 2/2

## 👉 Object Adapter



# Adapter – Partecipanti

## Target

- Definisce l'interfaccia specifica che il client utilizza.

## Client

- Collabora con gli oggetti conformi all'interfaccia Target.

## Adaptee

- L'interfaccia esistente che deve essere adattata.

## Adapter

- Adatta l'interfaccia di Adaptee all'interfaccia Target.

# Adapter – Conseguenze

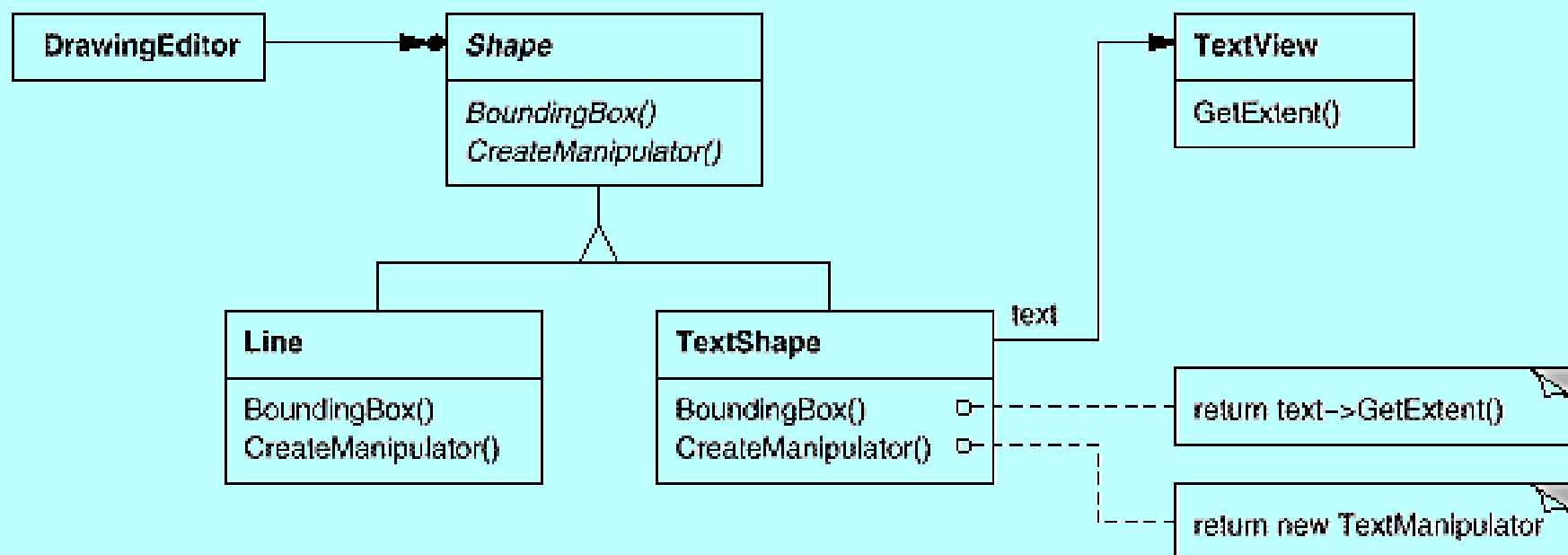
- ☞ Class Adapter:
  - Non va bene se vogliamo adattare anche le sottoclassi;
  - Adapter potrebbe sovrascrivere dei comportamenti di Adaptee;
- ☞ Object Adapter:
  - Un singolo Adapter funziona con gli oggetti Adaptee e quelli delle sottoclassi;
  - È difficile sovrascrivere il comportamento di Adaptee.
- ☞ In alcuni linguaggi (Smalltalk) è possibile realizzare classi che includono un adattamento dell'interfaccia (*pluggable adapter*).
- ☞ Se due client richiedono di vedere un oggetto in maniera diversa è possibile definire dei two-way adapters.

# Adapter – Implementazione

- ☞ In C++, un class adapter può essere implementato facendo ereditare ad Adapter pubblicamente da Target e privatamente da Adaptee:
- In tal modo Adapter è un sottotipo di Target ma non di Adaptee.

# Adapter – Esempio 1/9

- ➡ Vediamo come implementare il diagramma sottostante sia per Class Adapter che per Object Adapter.
- ➡ Shape e TextView sono uguali, ovviamente, in entrambi i casi.



# Adapter – Esempio 2/9

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

# Adapter – Esempio 3/9

- Class Adapter

```
class TextShape : public Shape, private TextView {  
public:  
    TextShape();  
    virtual void BoundingBox(  
        Point& bottomLeft, Point& topRight  
    ) const;  
    virtual bool IsEmpty() const;  
    virtual Manipulator* CreateManipulator() const;  
};
```

# Adapter – Esempio 4/9

- La funzione BoundingBox converte l'interfaccia di TextView per renderla conforme a quella di Shape.

```
void TextShape::BoundingBox (  
    Point& bottomLeft, Point& topRight  
) const {  
    Coord bottom, left, width, height;  
    GetOrigin(bottom, left);  
    GetExtent(width, height);  
    bottomLeft = Point(bottom, left);  
    topRight = Point(bottom + height, left + width);  
}
```

# Adapter – Esempio 5/9

```
bool TextShape::IsEmpty () const {  
    return TextView::IsEmpty();  
}
```

- Si supponga che esista una classe TextManipulator per la manipolazione di un TextShape.

```
Manipulator* TextShape::CreateManipulator () const {  
    return new TextManipulator(this);  
}
```

# Adapter – Esempio 6/9

- Object Adapter
  - Utilizza la composizione di oggetti per combinare classi con interfacce diverse.
  - In questo approccio l'adattatore TextShape mantiene un puntatore a TextView.

# Adapter – Esempio 7/9

```
class TextShape : public Shape {
public:
    TextShape(TextView*);
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

# Adapter – Esempio 8/9

- TextShape deve inizializzare il puntatore all'istanza di TextView.

```
TextShape::TextShape (TextView* t) {  
    _text = t;  
}  
  
void TextShape::BoundingBox (  
    Point& bottomLeft, Point& topRight  
) const {  
    Coord bottom, left, width, height;  
    _text->GetOrigin(bottom, left);  
    _text->GetExtent(width, height);  
    bottomLeft = Point(bottom, left);  
    topRight = Point(bottom + height, left + width);  
}
```

# Adapter – Esempio 9/9

```
bool TextShape::IsEmpty () const {  
    return _text->IsEmpty();  
}
```

- CreateManipulator è uguale al caso del Class Adapter.

# Facade vs. Adapter

- ➡ Entrambi sono “wrapper” (involucri).
- ➡ Entrambi si basano su un’ interfaccia, ma:
  - Façade la semplifica;
  - Adapter la converte.

# Composite (Composto)

## Scopo

*Comporre oggetti in strutture ricorsive ad albero per rappresentare gerarchie parte-tutto e consentire ai clienti di trattare in modo uniforme oggetti singoli/semplici e composizioni di oggetti*

# Esempio (1/5)

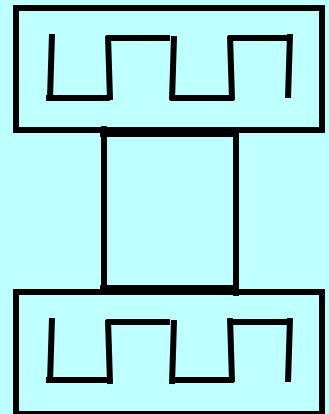
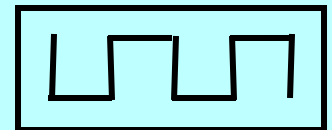
☞ Programma di grafica

☞ Deve consentire di

- Creare oggetti semplici (Punto, Linea, Cerchio,...)
- Raggruppare dinamicamente oggetti semplici in oggetti composti (4 linee in un rettangolo, ...)
- **Trattare gli oggetti composti come se fossero oggetti semplici**

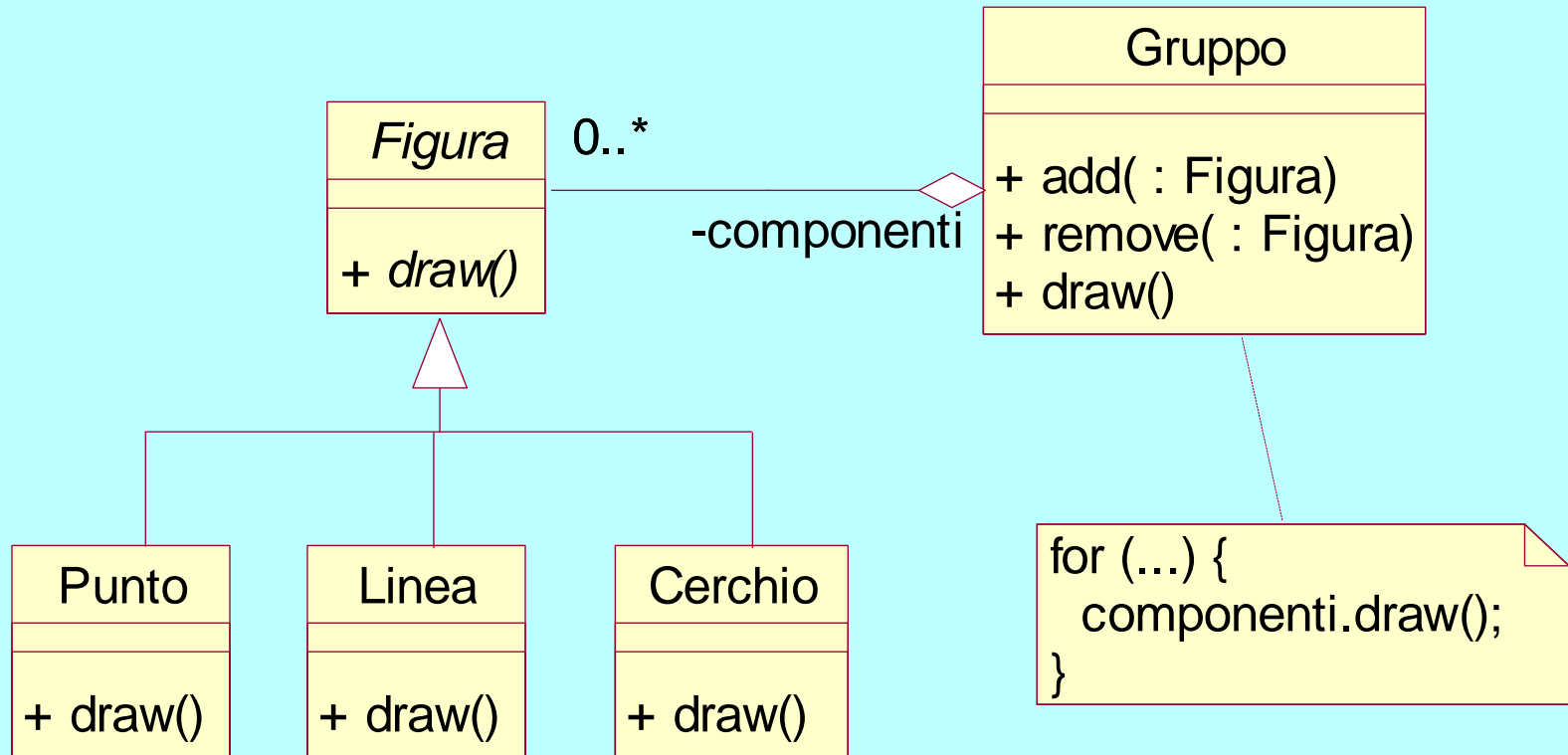
# Esempio (2/5)

- ➡ Raggruppare quattro linee per fare un rettangolo
- ➡ Raggruppare N linee per fare una serpentina
- ➡ Raggruppare un rettangolo e una serpentina...
- ➡ ...
- ➡ ...Proviamo a fare un diagramma UML...



# Esempio (3/5)

1° tentativo: cos'è che non va?

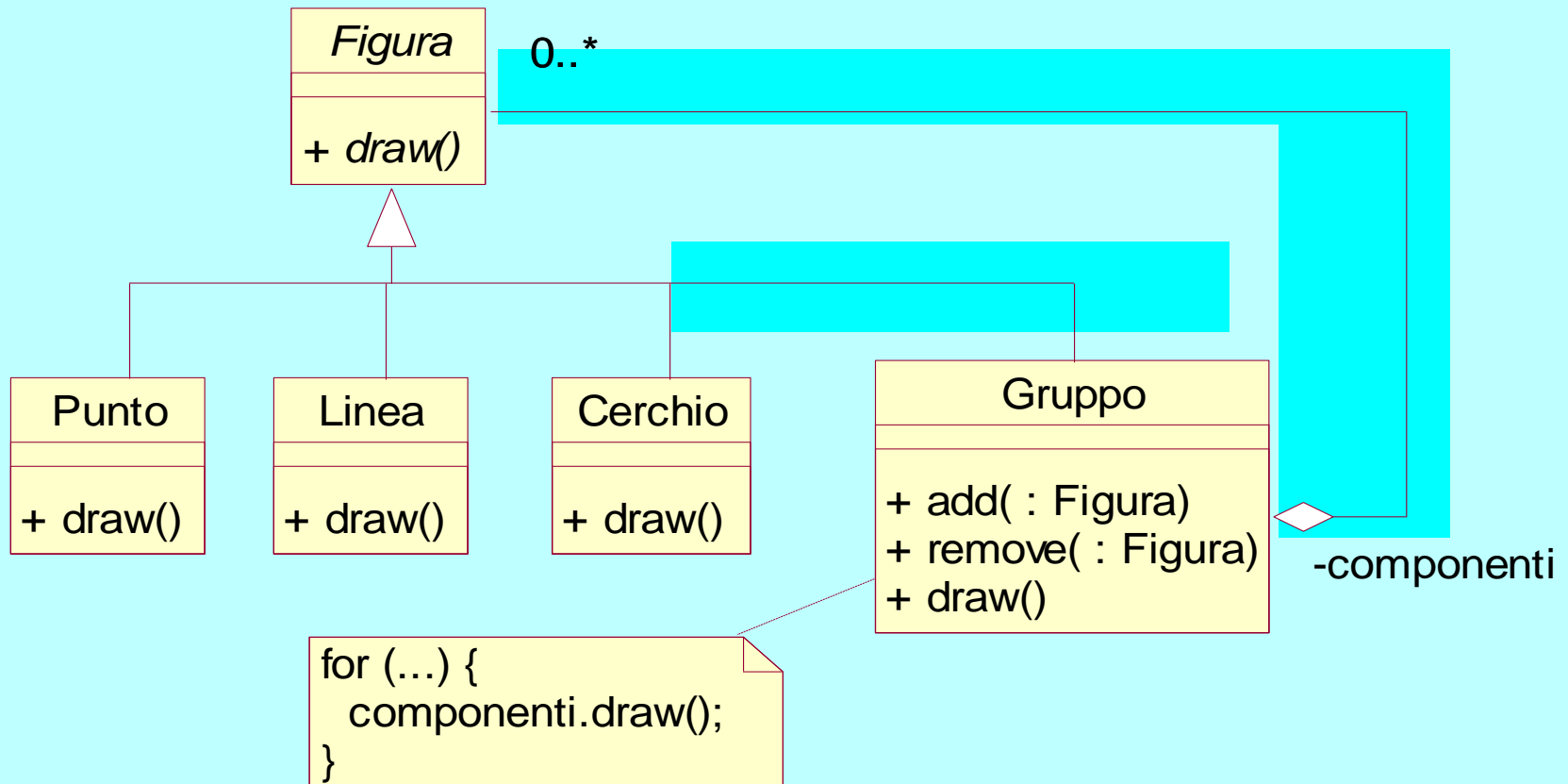


Risp.: gestione Gruppo  $\neq$  gestione Figura...

# Esempio (4/5)

2° tentativo: **Gruppo** sottoclasse di **Figura**!

Questa è l'idea base del Composite



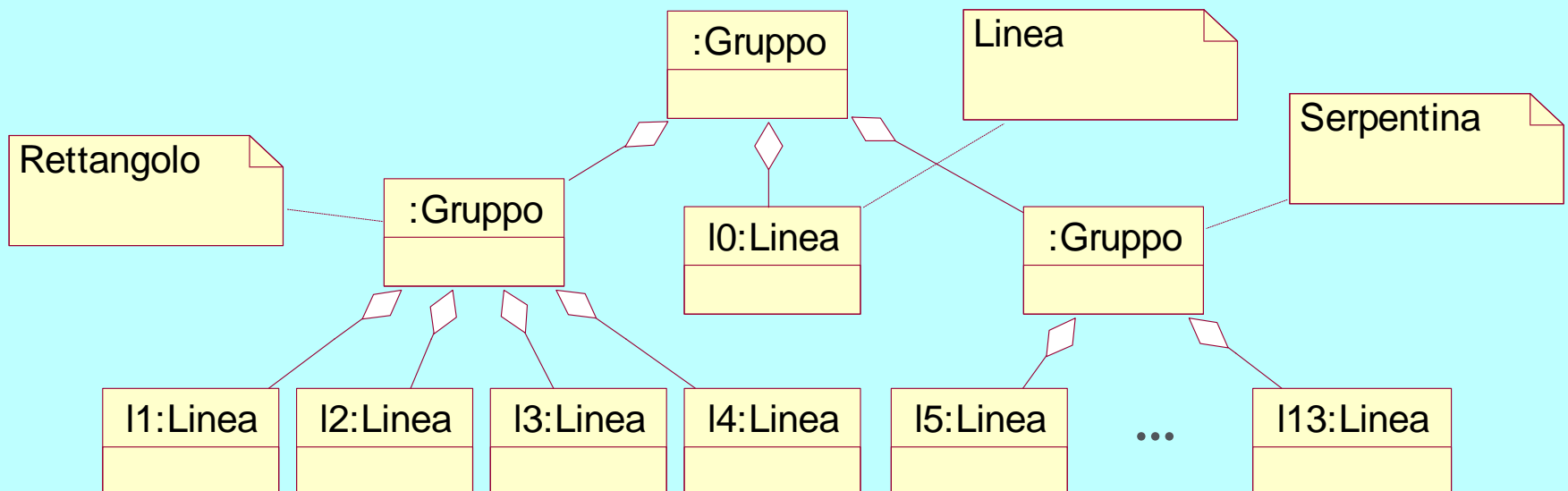
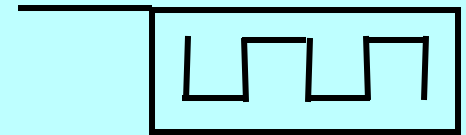
# Esempio (5/5)

- ☞ Un punto/cerchio/linea è una figura...
- ☞ ...“Trattare gli oggetti composti come se fossero oggetti semplici” ...
- ☞ Un gruppo **è una** figura!
- ☞ La ricorsione è in:
  - **Gruppo** contiene **Figura**,
  - e siccome alcune figure sono gruppi (**Gruppo** sottoclasse di **Figura**)
  - un gruppo contiene gruppi...

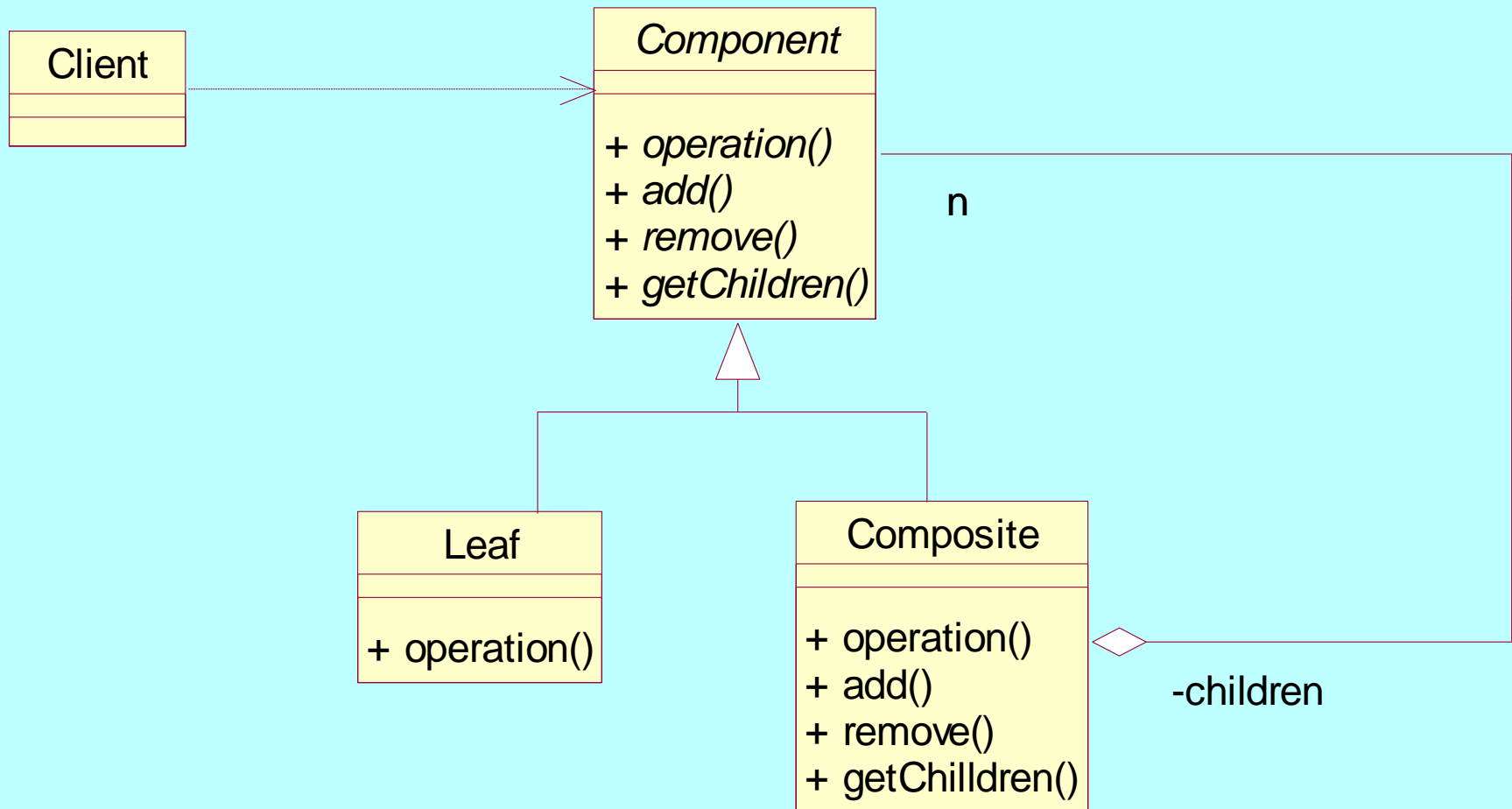
# Diagramma oggetti: albero!

☞ Per rappresentare il gruppo rettangolo+serpentina+linea:

- 3 istanze di **Gruppo**
- 14 istanze di **Linea**



# Diagramma Composite



# Observer

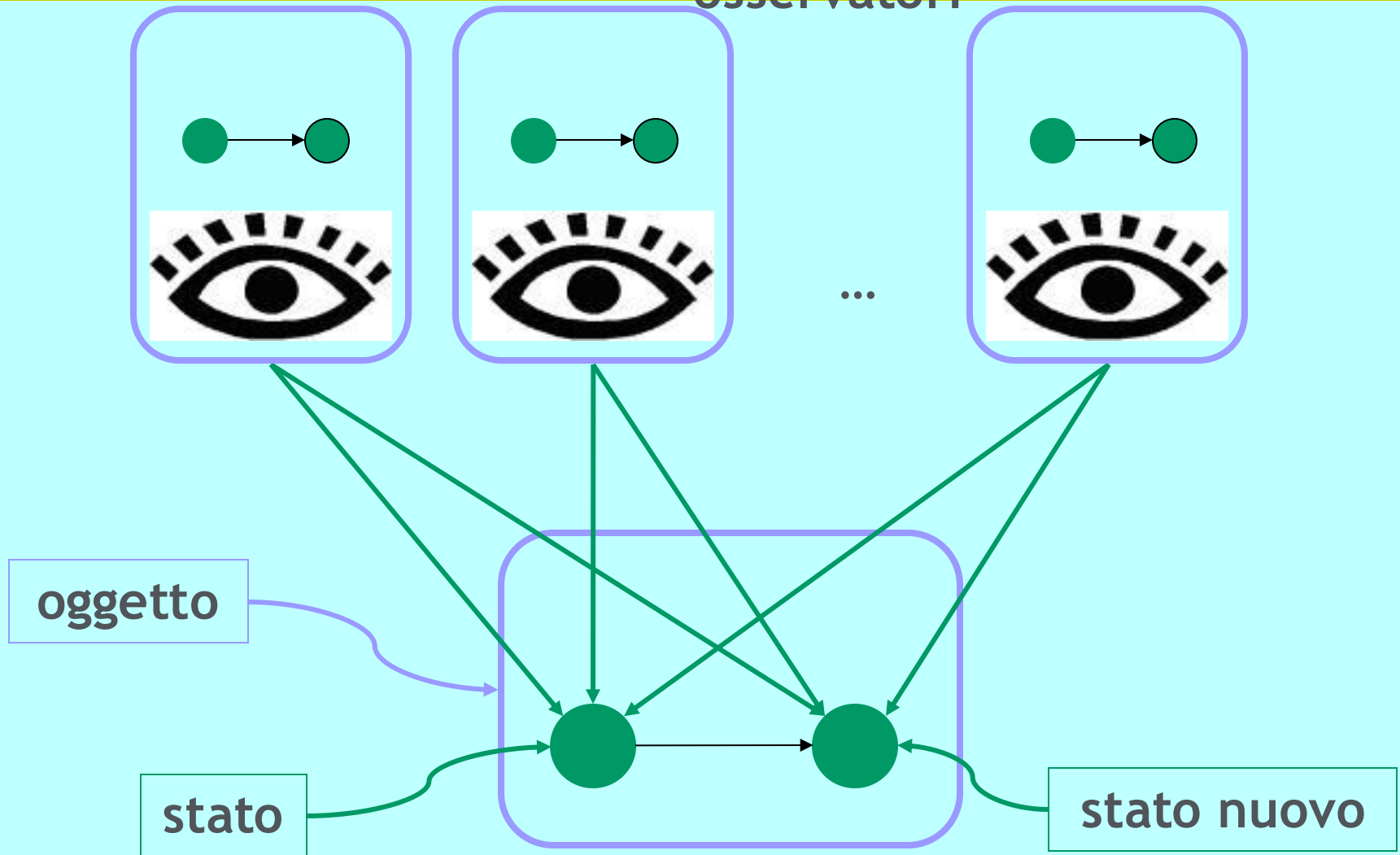
## Scopo

Definire una dipendenza “uno a molti” tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da lui sono notificati e aggiornati automaticamente

*“ in effetti l'osservatore non osserva, rimane in attesa che gli venga detto di andare a guardare...”*

# Motivazione

osservatori



# Motivazioni

- ☞ Una possibile soluzione potrebbe essere quella di utilizzare attributi pubblici
- ☞ Oppure gli osservatori potrebbero invocare continuamente un metodo dell'oggetto osservato?
- ☞ **Ma queste soluzioni presentano vari problemi**
  - scoprire la variazione troppo tardi
  - “perdersi” la prima di due variazioni “veloci”
  - E se ci sono tanti osservatori? Poco scalabile!
  - L'osservato passa il tempo a interrogare lo stato dell'oggetto (magari senza nessuna variazione)

# Quindi

- ☞ Gli osservatori si **registrano** presso l'oggetto osservato
- ☞ Quando l'oggetto osservato cambia stato, **notifica** tutti gli osservatori (ne invoca un metodo)
- ☞ Quando notificato, ogni osservatore decide cosa fare
  - Niente
  - **Richiedere** all'osservato informazioni sullo stato
  - ...

# Quindi / 2

- ☞ Gli osservatori potrebbero essere varie “viste” sullo stesso oggetto (ad esempio una rappresentazione tabellare e a grafico che “osservano” gli stessi dati)
- ☞ L’oggetto osservato non conosce l’identità precisa degli osservatori
- ☞ Gli osservatori si aggiungono/tolgono a runtime

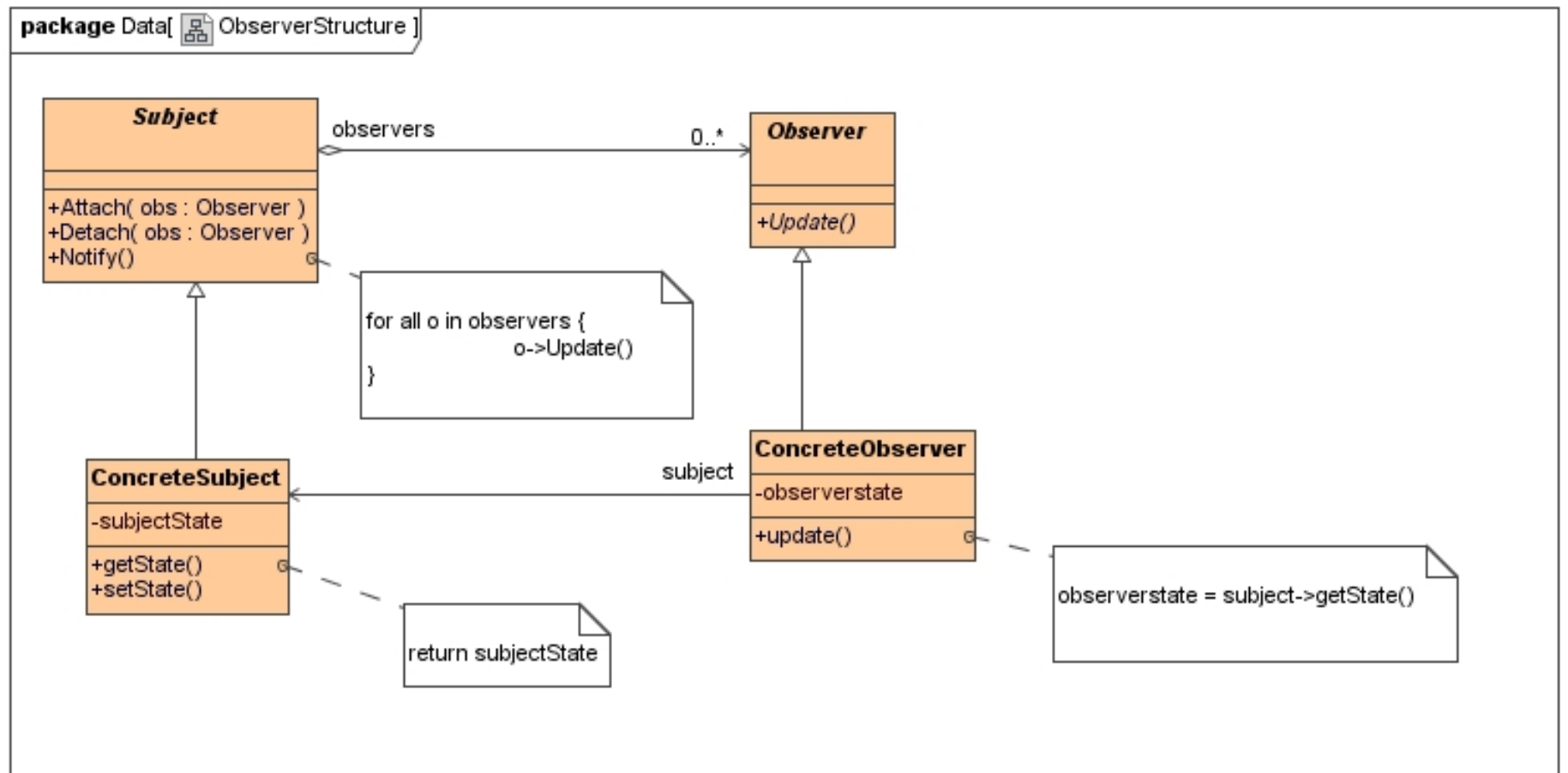
# Il pattern

- ☞ Gli osservatori devono potersi registrare
  - L'oggetto osservato deve fornire un'interfaccia standard per la registrazione
  - **register()** e **remove()**
- ☞ L'oggetto osservato deve poter notificare
  - Gli osservatori devono fornire un'interfaccia standard per la notifica : **update()** (e l'oggetto osservato ha **notify()** che chiama tutti gli update dei registrati)

# Applicabilità

- Quando un' astrazione ha due aspetti, uno dipendente dall' altro. Incapsulando questi aspetti in oggetti separati ci consente di variarli e riutilizzarli indipendentemente
- Quando un cambiamento ad un oggetto comporta il cambiamento di altri e non si sa quanti oggetti siano
- Quando un oggetto dovrebbe essere capace di notificare altri oggetti senza fare assunzioni su chi siano questi oggetti. In pratica non vogliamo che questi oggetti siano fortemente accoppiati.


# Diagramma Observer

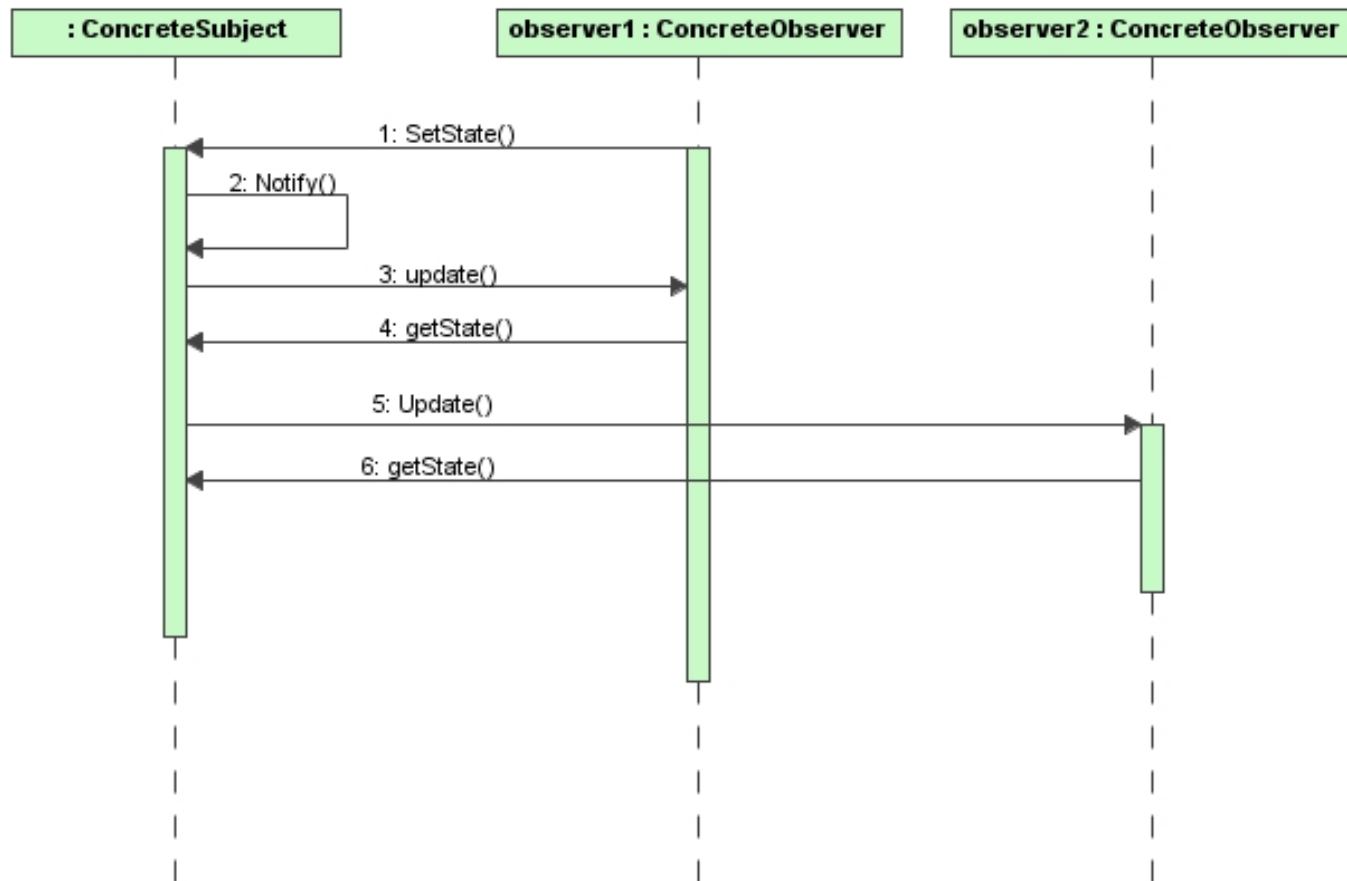


# Partecipanti

- **Subject**
  - *Conosce i suoi osservatori. Un qualunque numero di oggetti Observer può osservare un subject*
- **Observer**
  - *Definisce un interfaccia per aggiornare gli oggetti che dovrebbero essere notificati dei cambiamenti in un subject*
- **ConcreteSubject**
  - *Conserva lo stato di interesse per gli oggetti ConcreteObserver*
  - *Invia una notifica ai suoi observers quando lo stato cambia*
- **ConcreteObserver**
  - *Mantiene un riferimento a un oggetto ConcreteSubject*

# Collaborazioni

interaction ObserverSequence[  ObserverSequence ]



# Commenti

☞ Chi invoca **setState()** ?

- Chiunque
- Uno degli osservatori

☞ Chi invoca **notify()** ?

- Ogni metodo dell'oggetto osservato che modifica lo stato
- Il cliente, dopo che ha terminato una sequenza di modifiche

# Varianti

- ☞ E se un osservatore osserva più oggetti, come fa a sapere chi ha variato lo stato?
  - Parametro di **update()** (ad es., **this**)
- ☞ Push vs. Pull
  - Pull: **update()** non passa nessuna info sullo stato, l'osservatore usa **getState()**
  - Push: **update()** passa anche lo stato
- ☞ ...

# Observer in Java

- ☞ In `java.util`
- ☞ Interfaccia `Observer`
  - `void update(Observable o, Object arg)`
- ☞ Classe `Observable`
  - `addObserver(Observer o)`
  - `deleteObserver(Observer o)`
  - `notifyObservers(Observer o)`
- ☞ Nomi di metodi differenti

# Observer in Java

```
/* File Name : EventSource.java */

package OBS;
import java.util.Observable;           //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable
{
    public void run()
    {
        try
        {
            final InputStreamReader isr = new InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true )
            {
                final String response = br.readLine();
                setChanged();
                notifyObservers( response );
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

# Observer in Java

```
/* File Name: ResponseHandler.java */

package OBS;

import java.util.Observable;
import java.util.Observer; /* this is Event Handler */

public class ResponseHandler implements Observer
{
    private String resp;
    public void update (Observable obj, Object arg)
    {
        if (arg instanceof String)
        {
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}
```

# Observer in Java

```
/* Filename : myapp.java */
/* This is main program */

package OBS;

public class myapp
{
    public static void main(String args[])
    {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver( respHandler );

        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
}
```

# State

## Scopo

- Permettere a un oggetto di cambiare il suo comportamento al variare del suo stato interno

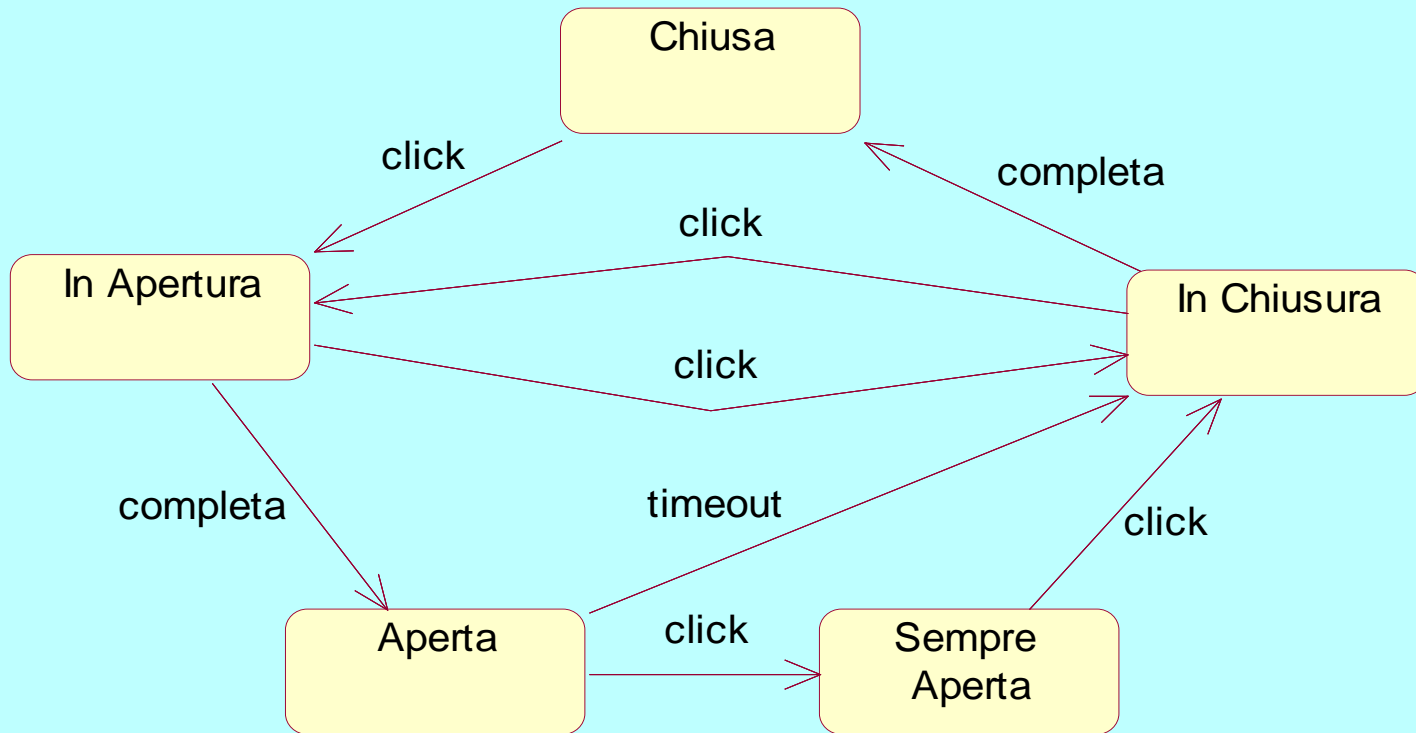
## Come funziona

- Si estrae la rappresentazione dello stato in classi esterne...
- ... organizzate in una gerarchia...
- ... e si sfrutta il polimorfismo per variare il comportamento

## Esempio

👉 Porta che si apre/chiude

- Click su pulsante (e tempi di attesa...)



# Implementazione ingenua

Porta
- stato
+ click()

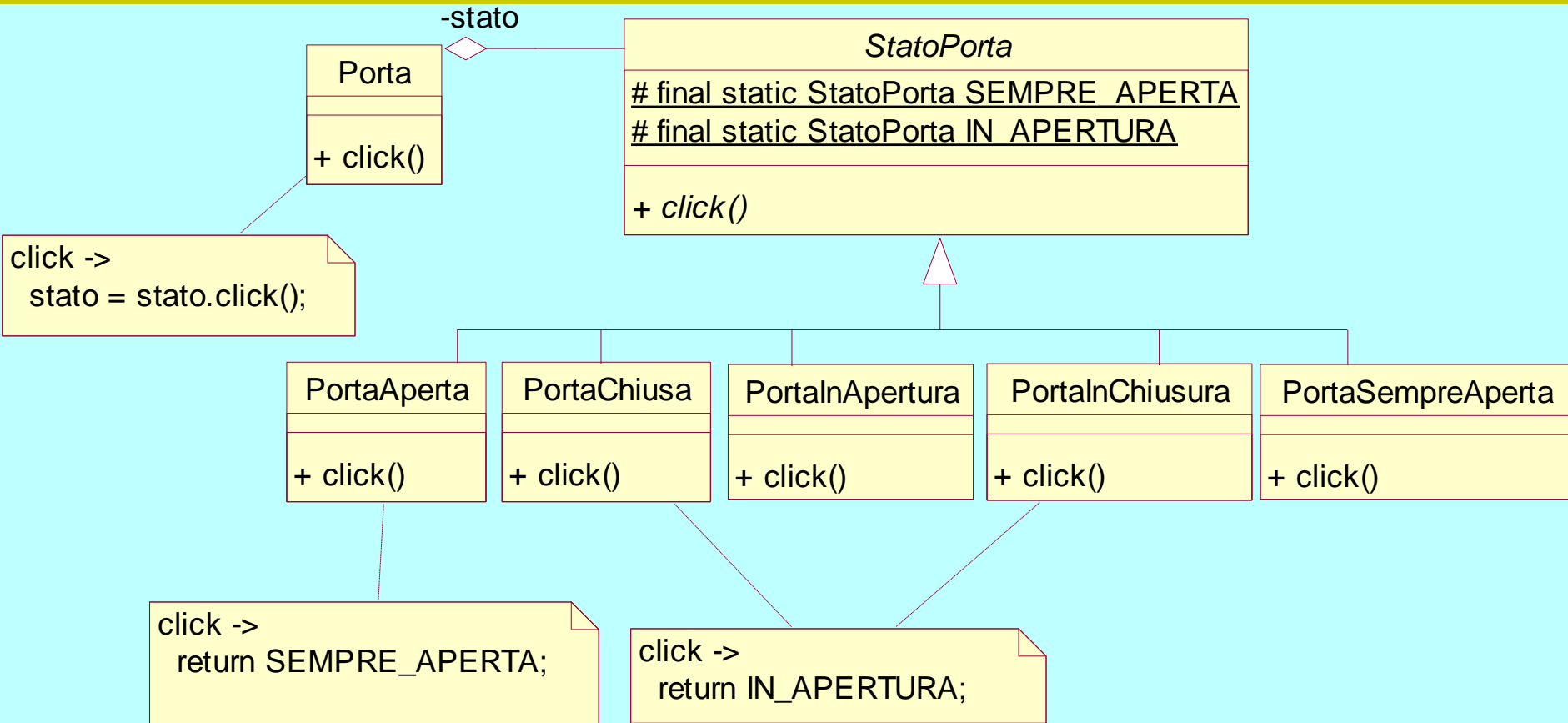
click ->

```
...  
if (stato==APERTA)  
    stato=SEMPRE_APERTA;  
else if (stato==CHIUSA || stato=IN_CHIUSURA)  
    stato=IN_APERTURA;  
else if (stato==SEMPRE_APERTA)  
    stato=IN_CHIUSURA;  
else if (stato==IN_APERTURA)  
    stato=IN_CHIUSURA;  
...
```

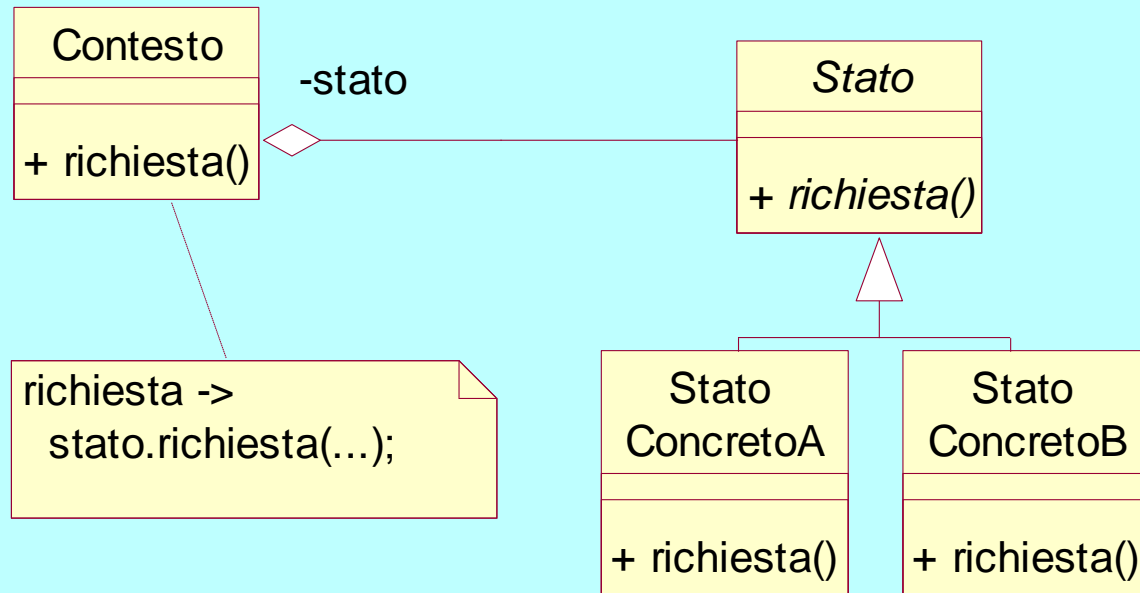
# Commenti

- ☞ Comportamento “cablato”
- ☞ Tanti if... (campanello d’ allarme)
- ☞ Difficile da comprendere...
- ☞ Alternativa
  - Usiamo una gerarchia di classi per rappresentare gli stati della porta
  - Ogni (sotto)classe uno stato
  - Ogni classe ha il suo **click**

# Implementazione con State



# Diagramma State



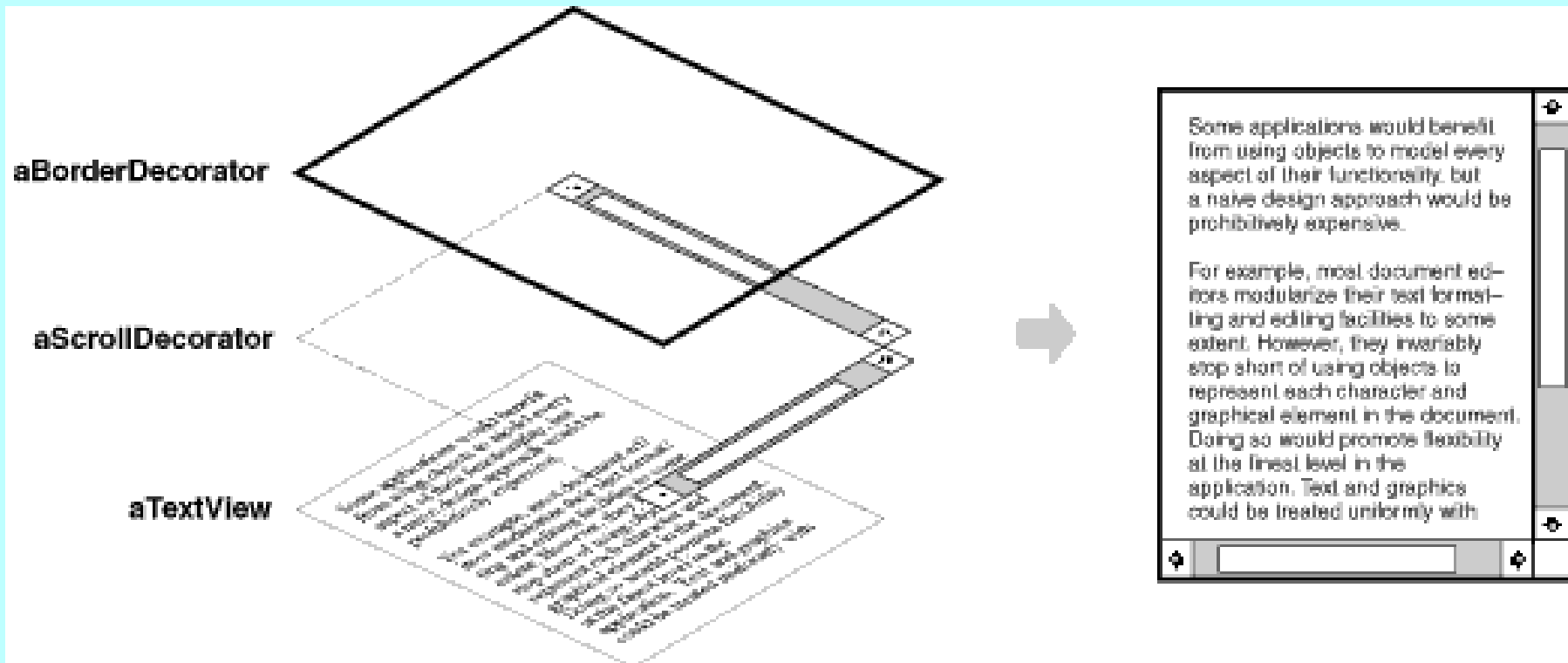
# Decorator (Decoratore)

## Scopo

*Aggiungere dinamicamente a un oggetto  
responsabilità/funzionalità/operazioni*

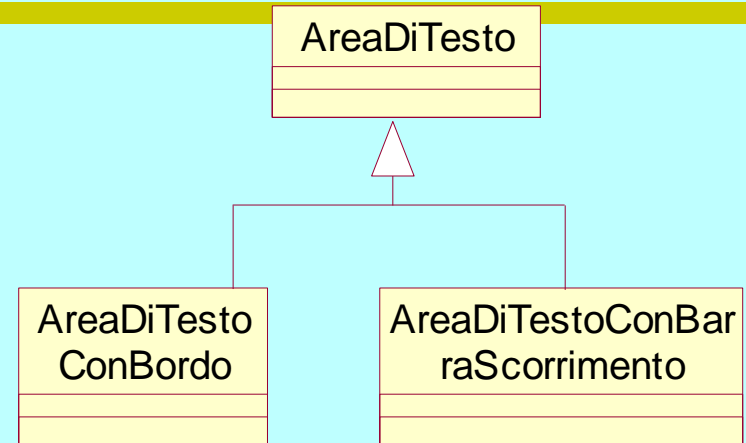
- ☞ Alternativa (più flessibile) alla creazione di sottoclassi per l'estensione di funzionalità
- ☞ (anche la specializzazione può aggiungere responsabilità/funzionalità/operazioni)

# Esempio



👉 Come ottenerlo?

# Aggiungere responsabilità staticamente?



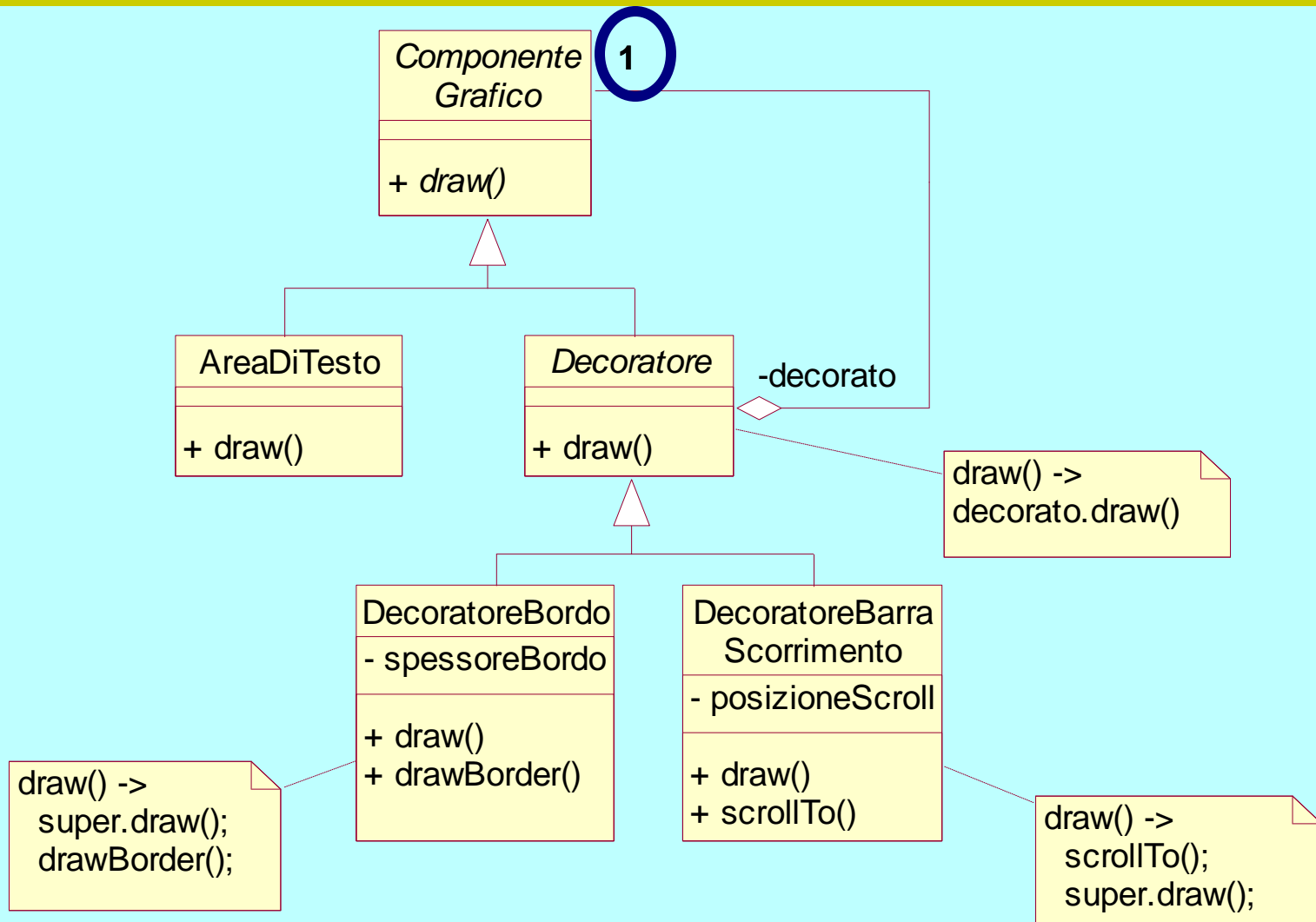
## ✎ Ereditarietà

- Creo istanze della classe voluta
- Contro
  - **AreaDiTestoConBordoEBarraScorrimento?**  
**AreaDiTestoConBarraScorrimentoEBordo?**  
Altre 2 sottoclassi?
  - Statico: non posso aggiungere funzionalità dopo la creazione
  - Altre componenti? (**Finestra**) Duplicazione!

# Aggiungere responsabilità dinamicamente

- ➡ Racchiudere l'oggetto da “decorare” (l'area di testo) in un altro oggetto, responsabile di gestire la “decorazione” (il bordo, la scrollbar): il “decoratore”
- ➡ Il decoratore trasferisce/delega e fa qualcosa in più (prima o dopo)
- ➡ Il decoratore ha l'interfaccia conforme all'oggetto decorato, e quindi è trasparente al cliente

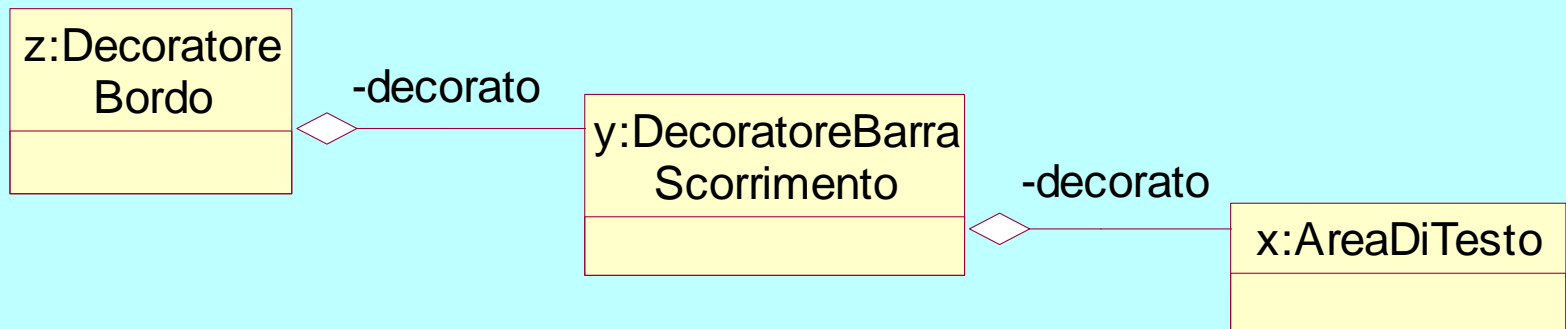
# Di nuovo l'esempio



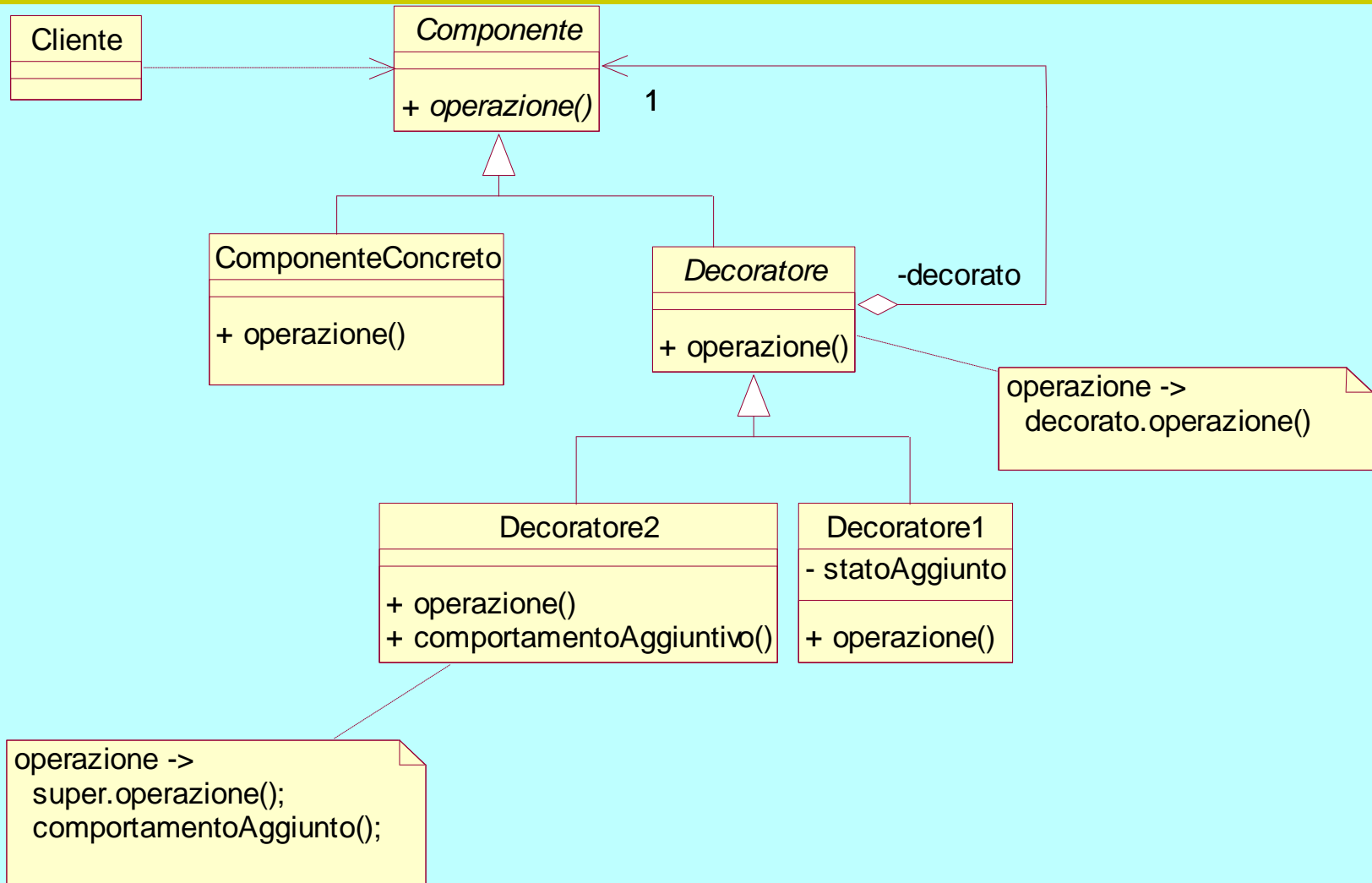
# Diagramma degli oggetti

☞ Catena con

- Varie decorazioni
- Alla fine, un'istanza di **AreaDiTesto**
- `new DecoratoreBordo (`  
    `new DecoratoreBarraScorrimento (`  
        `new AreaDiTesto() ) )`
- `new DecoratoreBarraScorrimento (`  
    `new DecoratoreBordo (`  
        `new AreaDiTesto() ) )`



# Diagramma Decorator



# Decoratore vs. ereditarietà

- Dinamico (run time)
- Senza vincoli per il cliente (può inventarsi combinazioni non previste)
- Aggiunge responsabilità a un singolo oggetto
- Evita esplosione combinatoria
- Statico (compile time)
- Con vincoli per il cliente (non può inventarsi combinazioni non previste)
- Aggiunge responsabilità a (tutte le istanze di) una classe
- Può causare esplosione combinatoria

# Usi tipici del Decorator

☞ Stream Java

☞ Interfacce utente grafiche

- Componente grafica
  - area di testo, finestra, panel, ...
- decorata con
  - bordo, barra di scorrimento, ...
  - (eventualmente più di uno!)

# Bridge (Ponte)

## Scopo

Separare un'astrazione dalla sua implementazione per far sì che possano variare indipendentemente

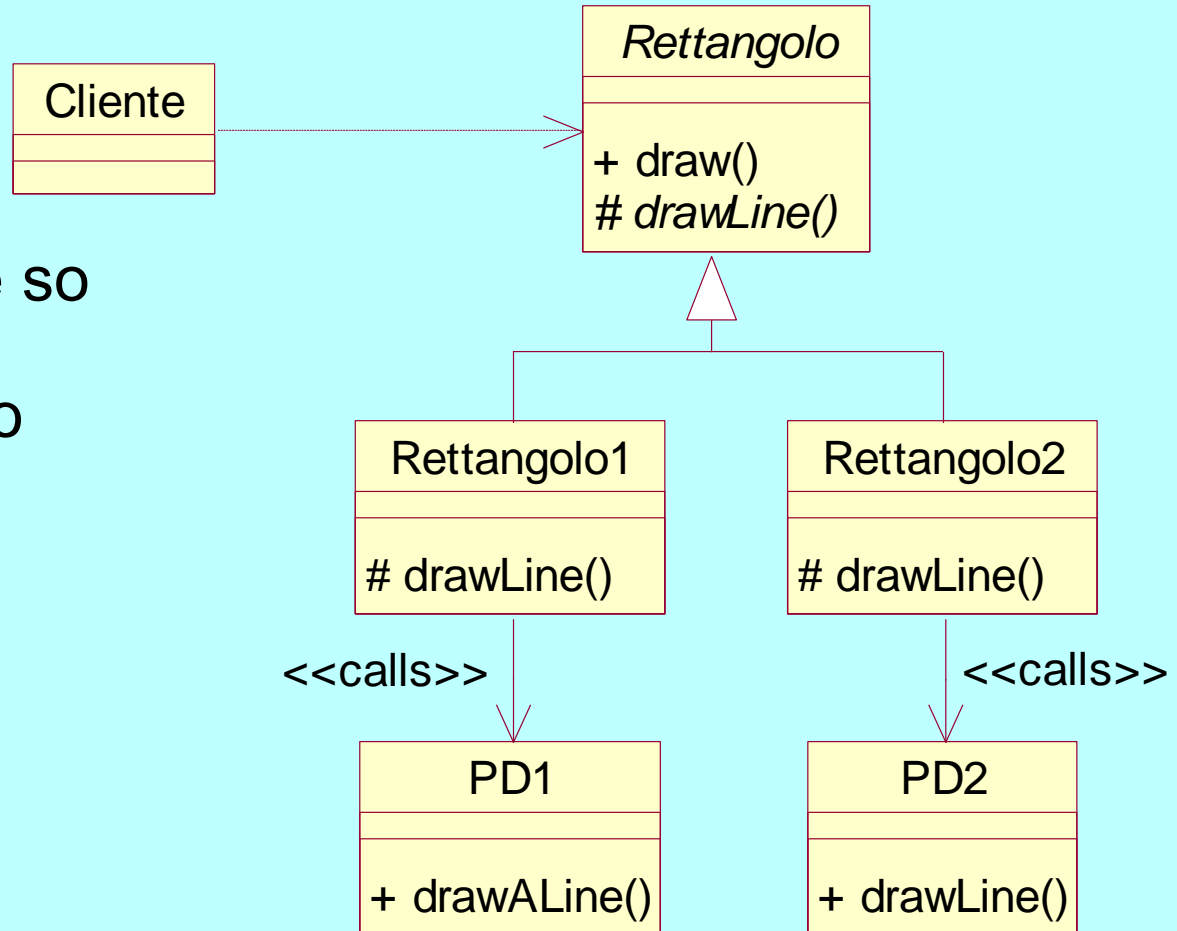
- ☞ Bel pattern, ma difficile
- ☞ Ricaviamolo, lavorando su un esempio

# Esempio

- ☞ Dobbiamo scrivere un programma che disegna rettangoli
- ☞ usando, in alternativa, uno di due programmi di disegno
  - PD1: `drawALine(x1, y1, x2, y2)`
  - PD2: `drawLine(x1, x2, y1, y2)`
- ☞ So quale dei due programmi usare al momento dell'istanziamento dei rettangoli
  - Il cliente non deve preoccuparsene

# Prima soluzione (ingenua...)

- 2 tipi di rettangoli, uno usa PD1 e l'altro PD2

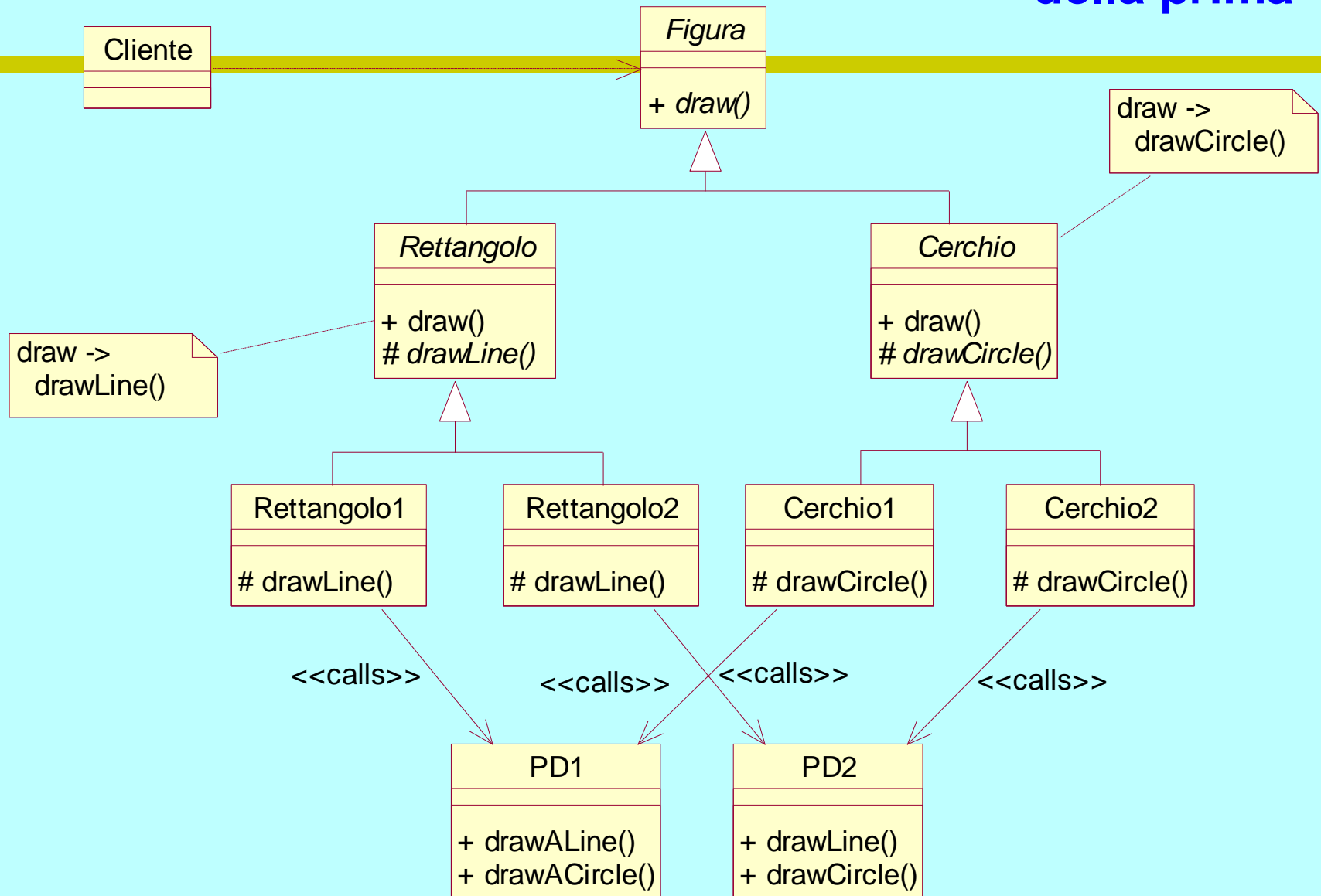


- All'istanziatura se istanziare **Rettangolo1** o **Rettangolo2**

# Mi ero dimenticato...

- ☞ ... che voglio disegnare anche cerchi,
  - Per fortuna DP1 e DP2 hanno
    - PD1: `drawACircle(x, y, r)`
    - PD2: `drawCircle(x, y, r)`
- ☞ e sempre consentendo al cliente di astrarre
  - Cerchi come rettangoli → **Figura**

# Seconda soluzione... ... revisione della prima



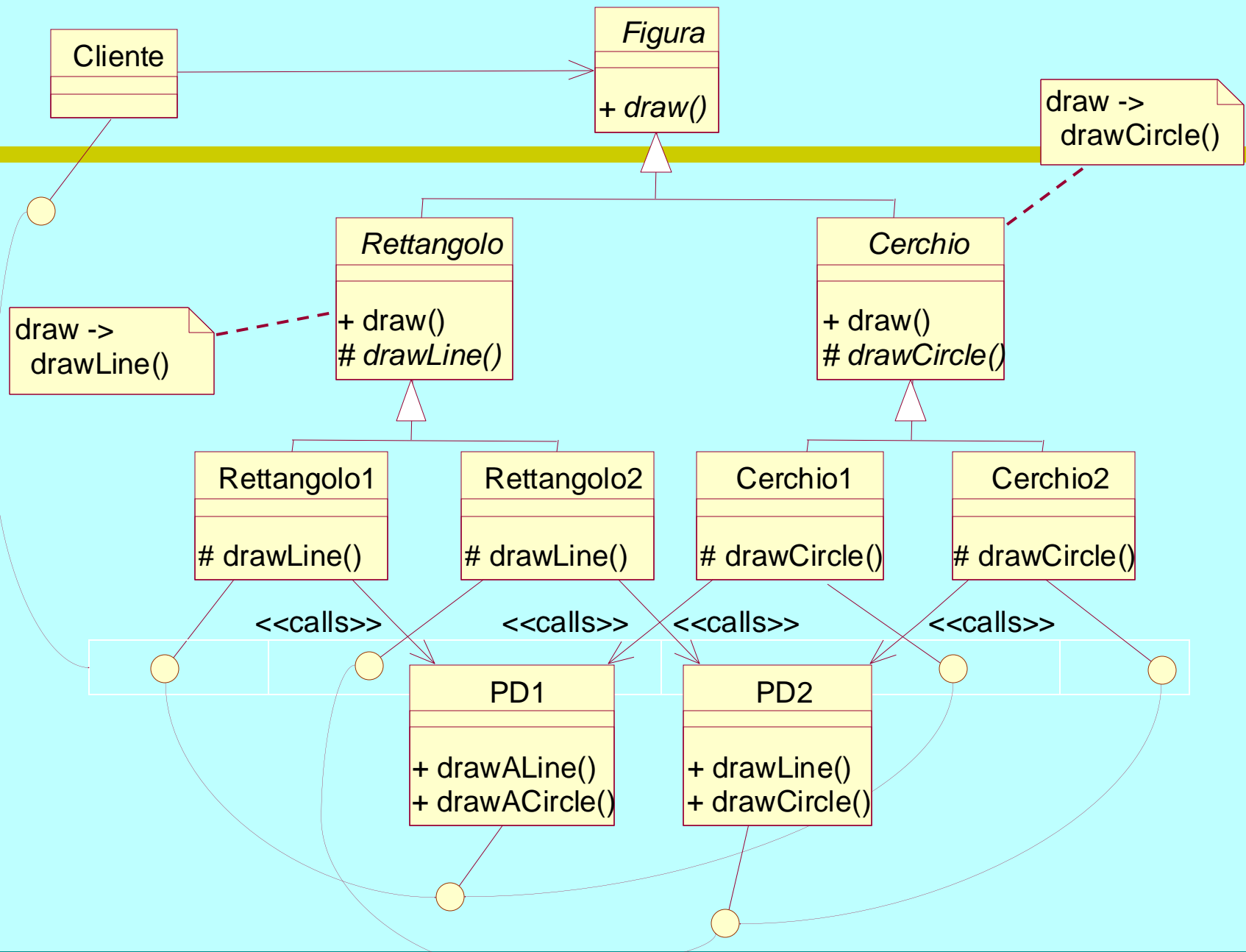
# Come funziona

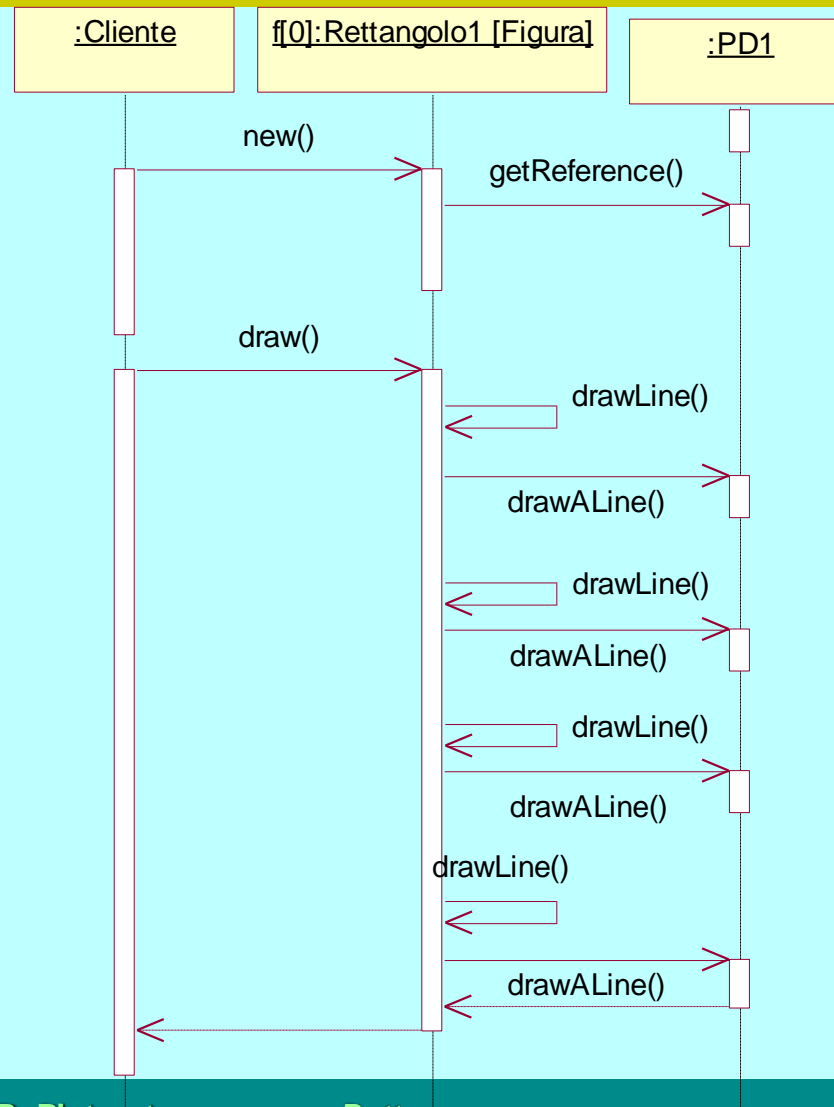
- Codice del **C**liente:

- Vediamo le istanze

- Vediamo il  
diagramma di  
sequenza

```
Figura[] f = new Figura[4];  
f[0] = new Rettangolo1(...);  
f[1] = new Rettangolo2(...);  
f[2] = new Cerchio1(...);  
f[3] = new Cerchio2(...);  
for (i = ...)   
    f[i].draw();
```

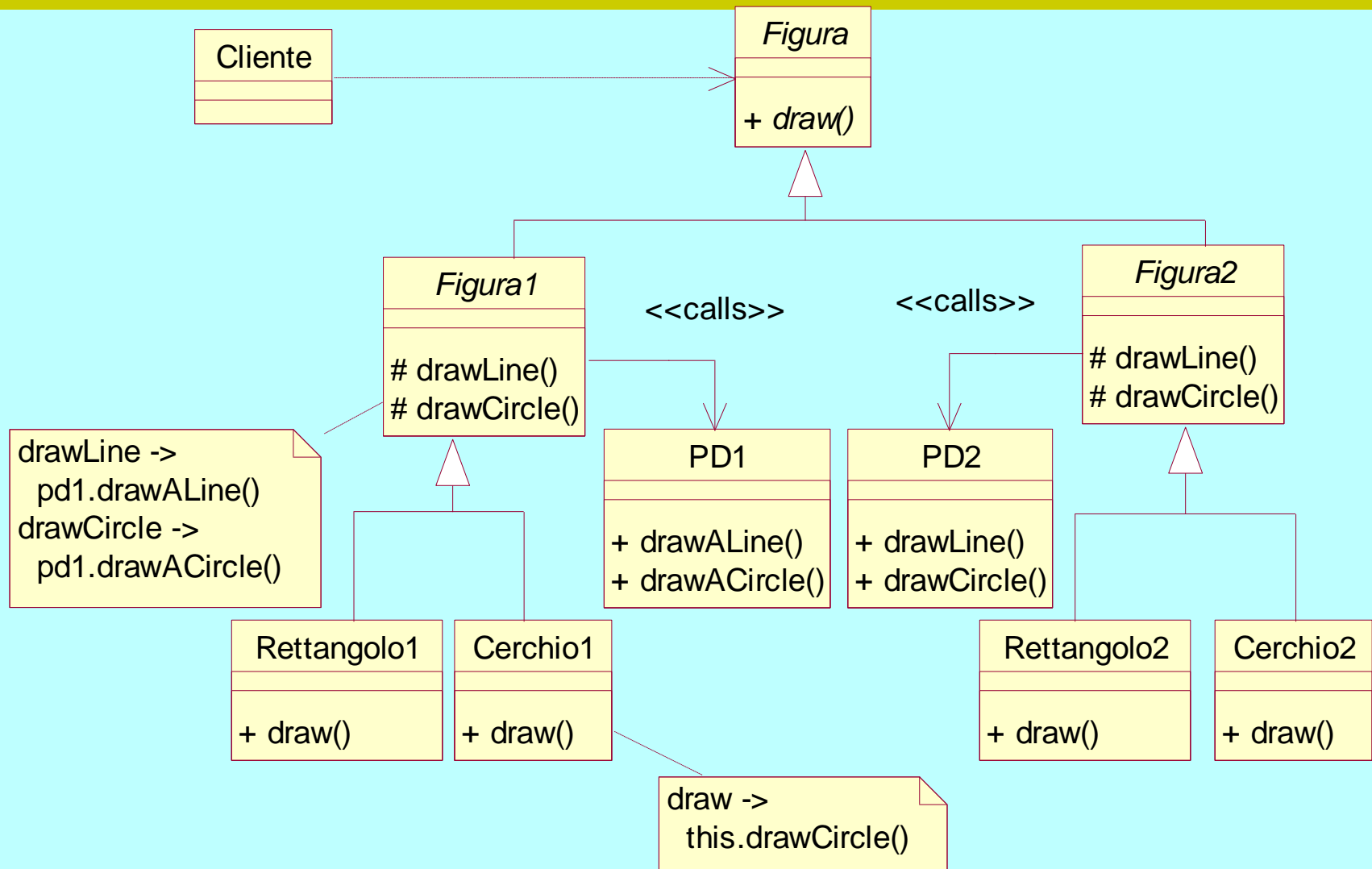




# Commenti

- ☞ E se devo gestire altre figure?
  - Per ogni ulteriore figura aggiungo una piccola gerarchia (3 classi)
    - Classe astratta
    - Una sottoclasse per ogni PD
- ☞ E se per caso devo gestire/usare anche altri programmi di disegno (**PD3**, **PD4**, ...)?
  - Per ogni ulteriore PD aggiungo una classe **PD<sub>i</sub>** e una classe in ogni gerarchia
- ☞ Sembra funzionare... ma non ci piace molto... Con N figure e M PD ho  $N \cdot M$  classi a “livello 3”

# Alternativa



# Commenti

- ☞ Ho scambiato “l’ordine”:
  - Da: Prima tipo di figura poi versione di PD
  - A: Prima versione poi tipo di figura
- ☞ Mah, non è cambiato molto
- ☞ Sempre  $N * M$ ...

# Uhm...

## ☞ Scopo del Bridge

- Separare un'astrazione dalla sua implementazione
- per far sì che possano variare indipendentemente

## ☞ Adesso capiamo! (?)

- Astrazione: le figure
- Implementazione: i programmi di disegno

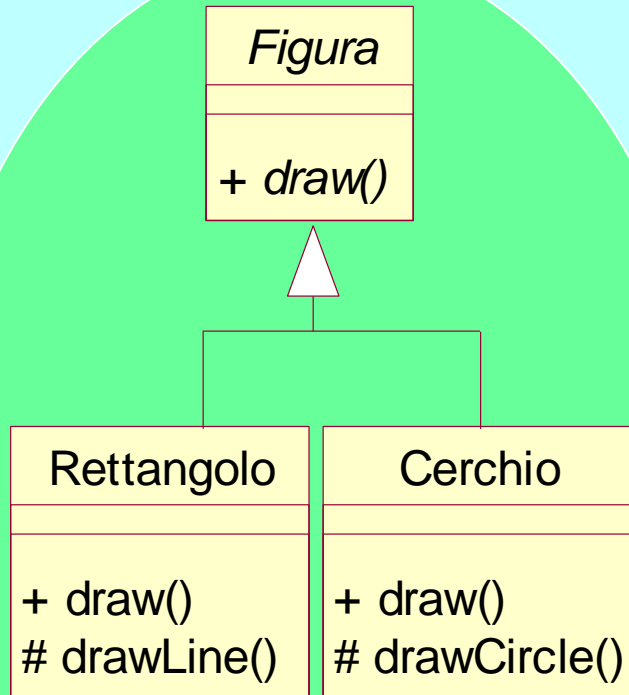
## ☞ Con la nostra soluzione

- Alto accoppiamento fra figure (astrazione) e PD (implementazioni)  $\Rightarrow$
- Non riesco a farle variare indipendentemente
- C'è anche ridondanza...

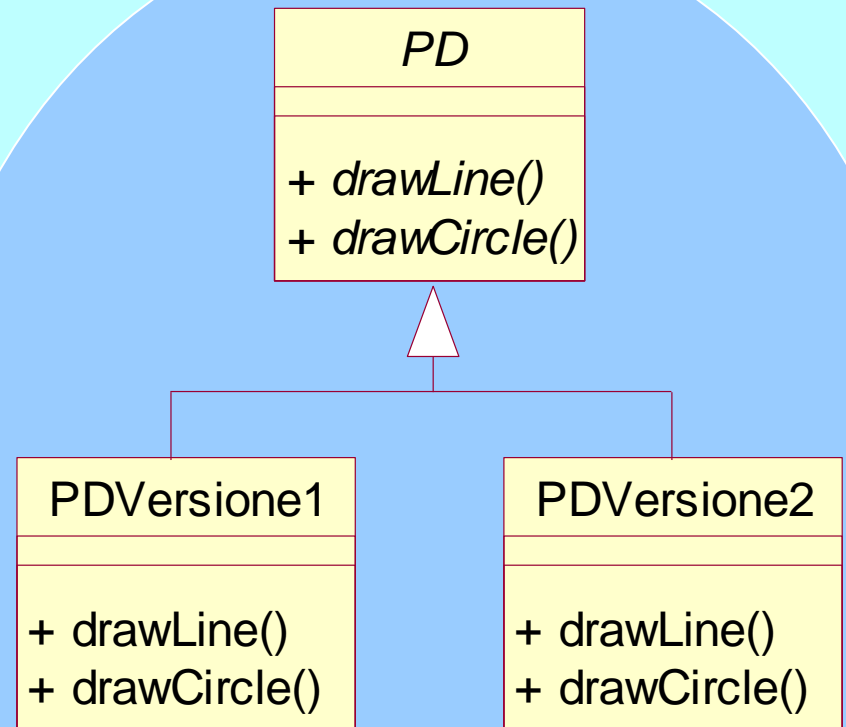
# È di qualità?

- ☞ Separiamo astrazione e implementazioni
  - Astrazione: **Figura**
    - Ci sono 2 tipi di **Figura**
  - Implementazioni: Programmi di disegno (**PD**)
    - Anche qui ne abbiamo 2 versioni

# Separiamo astrazione e implementazione



Astrazione

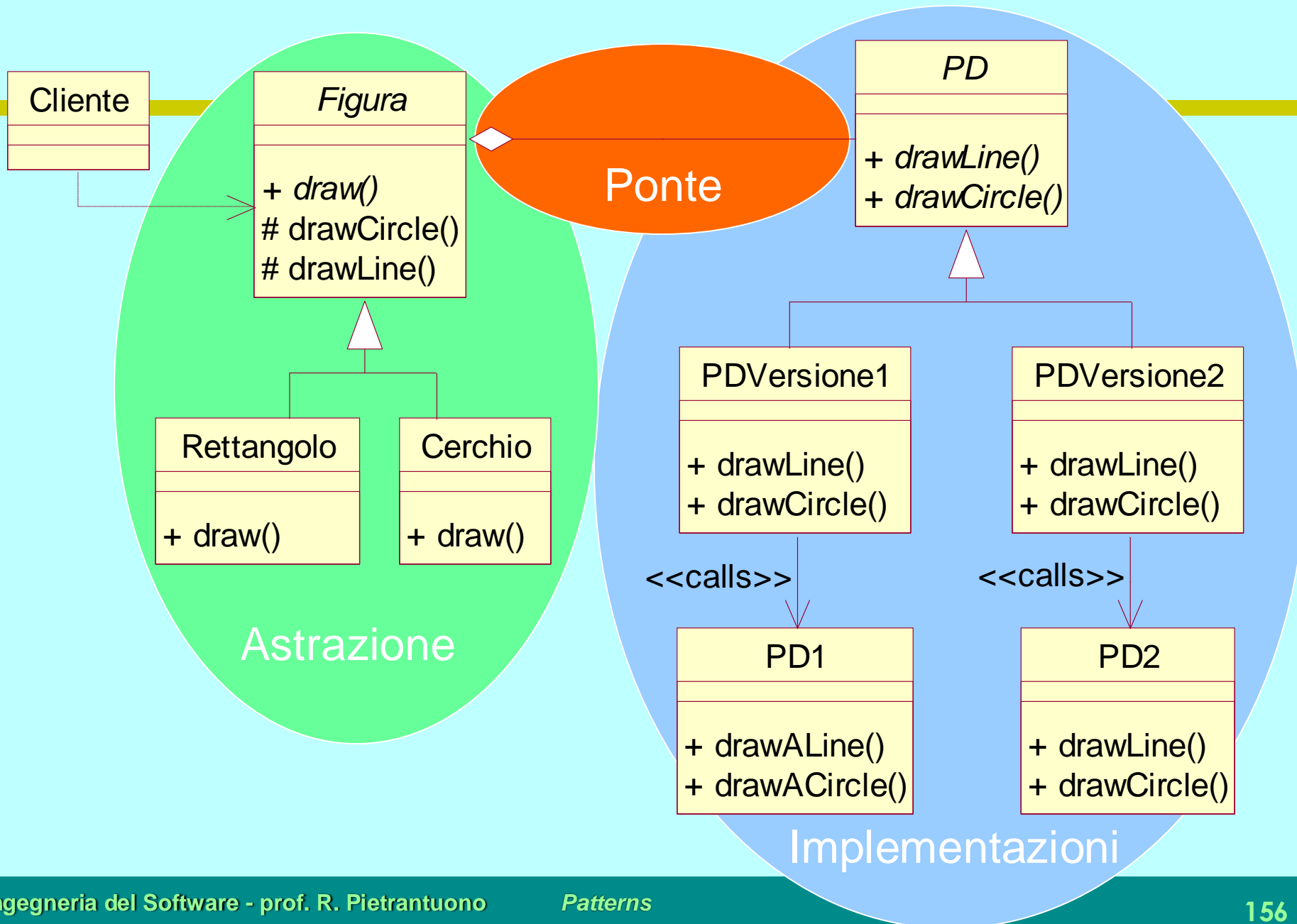


Implementazioni

# Come li colleghiamo?

- ☞ Con un **ponte** (“bridge”)...
- ☞ Una **Figura** delega la responsabilità del disegno (**draw**) a un (PD)
- ☞ Già che ci siamo, un dettaglio:
  - Rifattorizziamo **#drawLine** e **#drawCircle** nella sopraclasse per evitare duplicazioni...
  - Sono **protected** ⇒
    - Ereditati
    - Non visibili nell’interfaccia di **Figura**

# Diagramma finale



# Come funziona

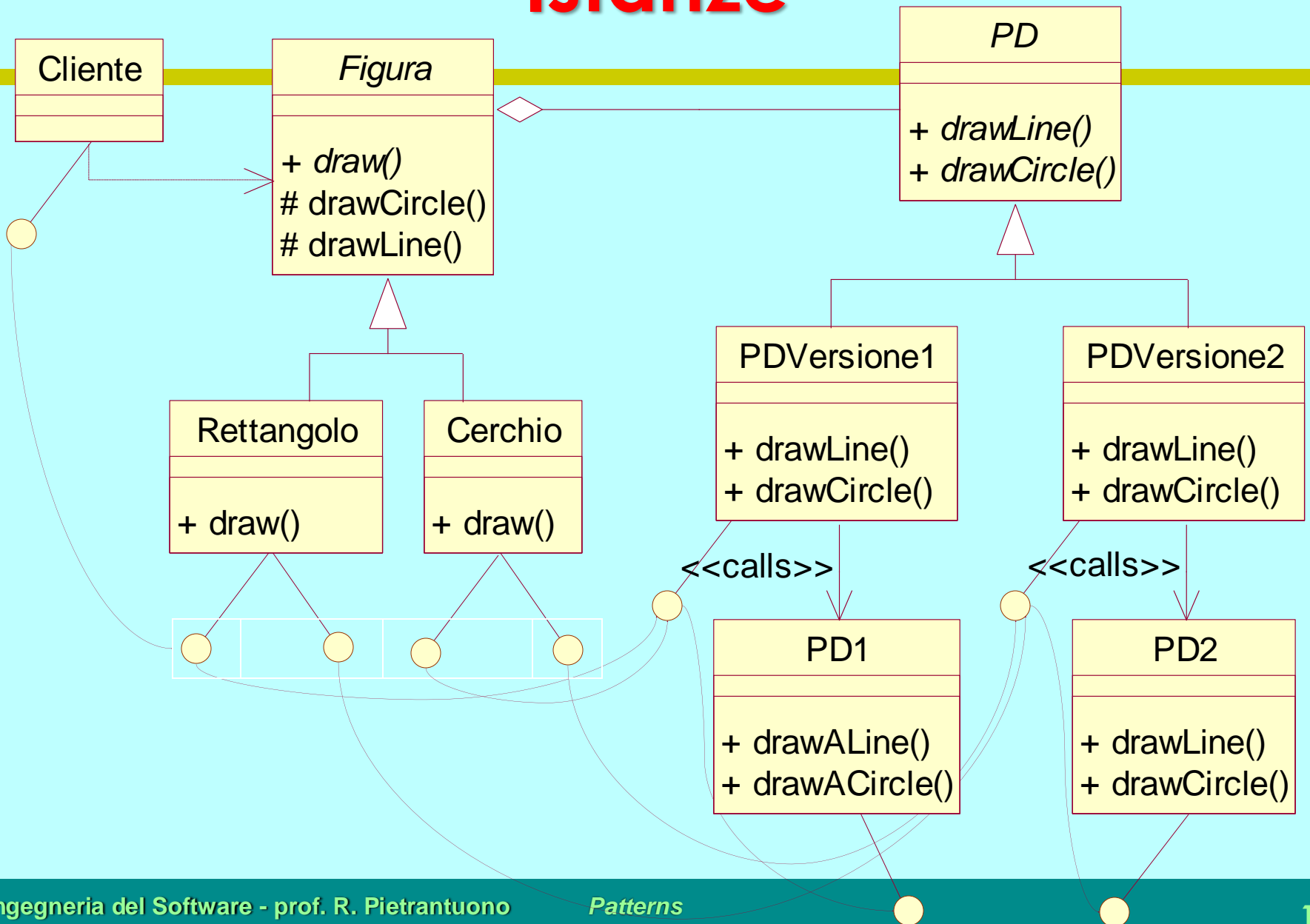
- Codice del **C**liente:

- Vediamo le istanze

- (Diagramma di sequenza: Ex.)

```
Figura[] f = new Figura[4];  
PD p1 = new PDVersione1();  
PD p2 = new PDVersione2();  
f[0] = new Rettangolo(p1);  
f[1] = new Rettangolo(p2);  
f[2] = new Cerchio(p1);  
f[3] = new Cerchio(p2);  
for (i = ...)  
    f[i].draw();
```

# Istanze



# Commenti



Abbiamo

- eliminato un po' di ereditarietà
- aggiunto composizione/delega



Non c'è più esplosione combinatoria

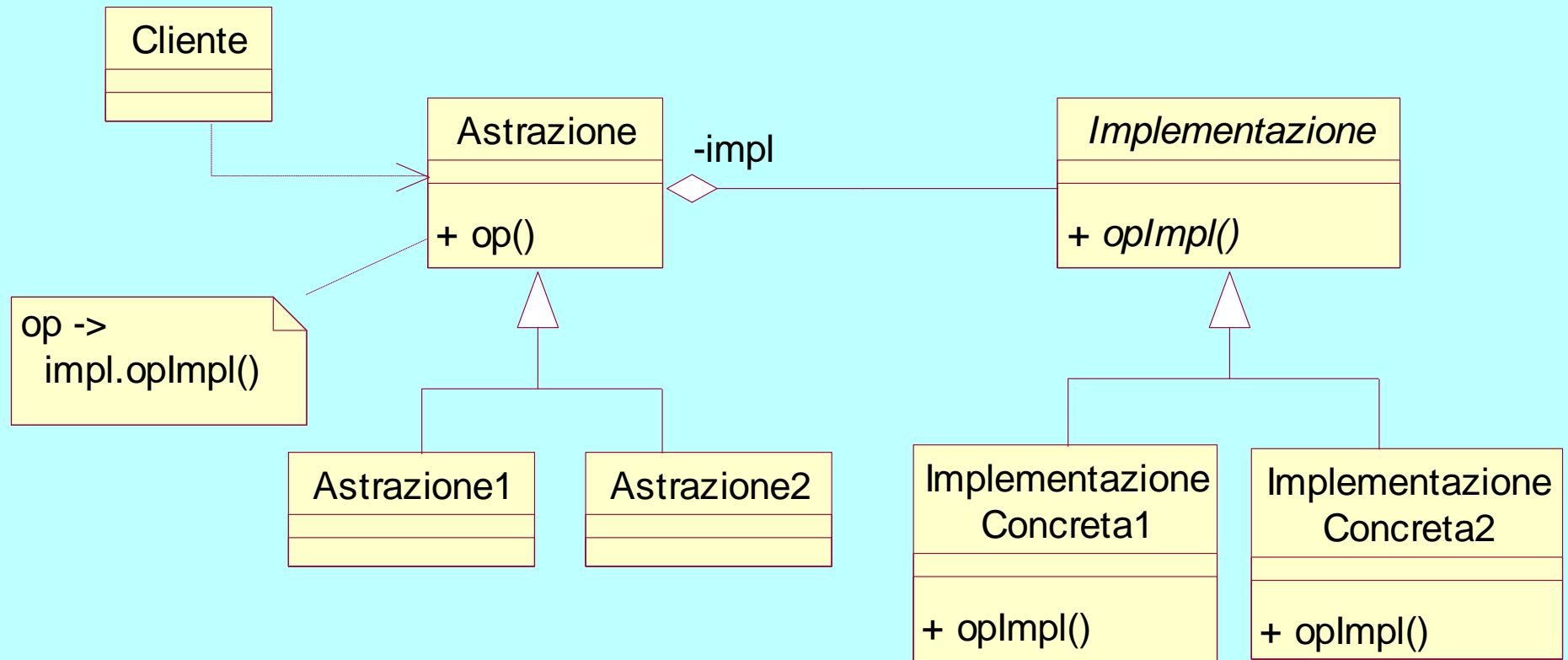
- $N + M$  classi, non  $N * M$



Come ci siamo arrivati?

- Scopo del Bridge (finalmente chiaro...)
  - **Separare** un'astrazione dalla sua implementazione
  - per far sì che possano **variare indipendentemente**

# Diagramma del Bridge



# Commenti

- ☞ Spesso interfaccia e implementazione
  - sono nella stessa classe,
  - o vengono separate in sopraclasse e sottoclasse
- ☞ Bridge le separa “di più”
- ☞ Evita anche ricompilazioni
- ☞ Morale più generale:
  - Mai accontentarsi della prima versione...
  - ... però evitare la “paralysis by analysis”...

# Il Bridge

- ☞ Bridge è uno strumento per il programmatore:
  - gli permette di organizzarsi il codice
  - per poter aggiungere nuove astrazioni (front-end) e nuove implementazioni (back-end) in modo semplice
- ☞ Si potrebbe usare delegazione invece di ereditarietà nella variazione di astrazioni
- ☞ Pro
  - Astrazioni con interfacce diverse
- ☞ Contro
  - Cliente non può usare il polimorfismo

# Bridge vs. Adapter

- ➡ Obiettivi diversi
- ➡ Adapter adatta, Bridge separa interfaccia e implementazione per farle variare indipendentemente
- ➡ Bridge si usa tipicamente all'inizio di un progetto,
- ➡ Adapter viene usato dopo che le classi coinvolte sono state progettate/implementate e vengono riusate
- ➡ Comunque le classi **PDVersione1** e **PDVersione2** sono due adattatori oggetto...
  - Soluzione frequente

# Bridge vs. Decoratore

- ➡ Obiettivo in comune: evitare la proliferazione (esplosione combinatoria) di classi
- ➡ Struttura diversa