Introduzione ai Design Pattern

Corso di Ignegneria del Software e Sistemi Informativi
Prof. Beniamino Di Martino

Lucidi a cura di Antonio Esposito

Cos'è un “pattern”?

- Una forma o un modello proposti per essere imitati (Dizionario Webster)
- Una regola costituita da tre elementi, che esprime una relazione tra un contesto, un problema e una soluzione (Alexander-1977)
- Una soluzione generale e riutilizzabile ad un problema comune in un dato contesto (Wikipedia)
- Descrizione di un problema comune e di una sua possibile soluzione, basata sull'esperienza maturata in situazioni simili

Un po' di storia

- **Christopher Alexander** (architetto): prima definizione di “pattern” e “pattern language” (1977)
  - Descrizione di una serie di pattern per la costruzione di fattorie in Svizzera.
- **Beck, Cunningham**: “Using Pattern Languages for Object-Oriented Programs” (1987)
  - Sviluppo di interfacce software guidato da pattern
- **Gamma, Helm, Johnson, Vlissides** (**GoF**): “Design Patterns” (1994)
  - Catalogo di 23 pattern ricorrenti nella progettazione object-oriented

Classificazione

- Analysis Pattern Servono a capire come funziona sistema
- Architectural Pattern Dicono come va a organizzare l'architettura del sistema
  → Più ad alto livello
- Design Pattern by GOF
  - Creational pattern Come vanno creati oggetti
  - Structural pattern Come sono fatti dalle gli oggetti
  - Behavioral pattern Come si comportano tra loro oggetti
  - Concurrency pattern Come lavorano oggetti che lavorano in parallelo
- Implementation pattern

Come implementare le classi? Dipendono dal linguaggio di programmazione

Descrizione dei pattern

- Non esiste uno standard per la definizione dei pattern
  - Ogni autore elabora un template personale
  - Molti elementi comuni possono essere rintracciati
- Descrizioni spesso accompagnate da grafici di supporto
- Focus della descrizione dipende da quale aspetto risulta più importante:
  - Struttura
  - Comportamento
- Le descrizioni fanno generalmente riferimento al template usato dalla GoF in “Design patterns: Elements of reusable object-oriented software”

Descrizione dei pattern: template

- **Pattern Name and Classification:** Un nome unico e descrittivo che aiuta nell'identificare e riferirsi al pattern.
- **Intent:** Una descrizione dell'obbiettivo che ha spinto alla creazione del pattern e le ragioni per utilizzarlo.
- **Also Known As:** Altri nomi per il pattern.
- **Motivation (Forces):** Uno scenario costituito da un problema e un contesto in cui il pattern può essere usato.
- **Applicability:** Situazioni in cui il pattern può essere usato (contesto).
- **Structure:** Una rappresentazione grafica. Spesso sono usati i diagrammi delle classi e delle iterazioni proposti da UML.
- **Participants:** Una lista delle classi e degli oggetti usati nel pattern.

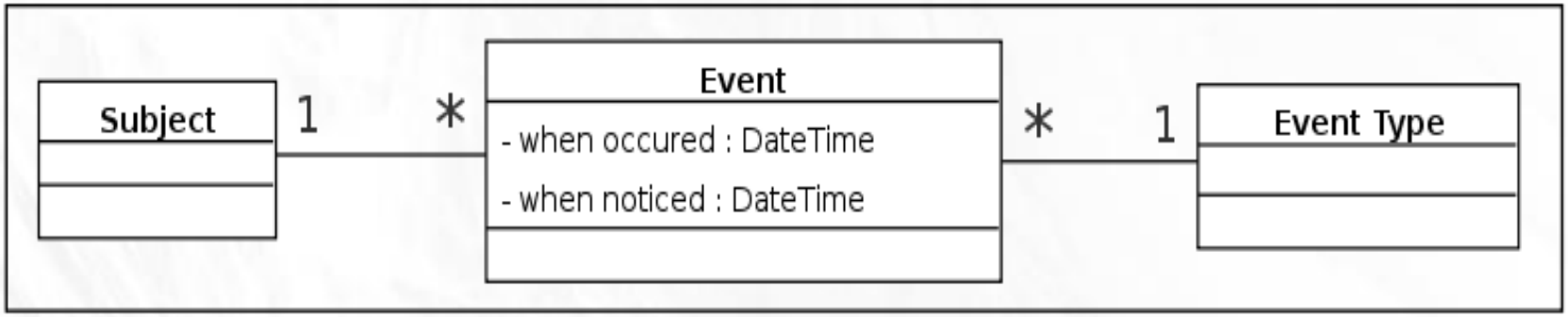
Descrizione dei pattern: template

- **Collaboration:** Una descrizione di come classi e oggetti interagiscono tra loro.
- **Consequences:** Una descrizione dei risultati, degli effetti secondari e delle limitazioni conseguenti all'uso del pattern.
- **Implementation:** Una descrizione di una implementazione del pattern.
- **Sample Code:** Un esempio di come il pattern possa essere usato in un linguaggio di programmazione.
- **Known Uses:** Esempi di usi reali del pattern.
- **Related Patterns:** Altri pattern che possono essere legati a quello corrente; in questo ambito possono essere introdotte eventuali differenze tra pattern simili.

Analysis pattern

- “Insieme di oggetti generici collegati, con attributi stereotipati, comportamenti e interazioni descritti in maniera neutrale” (Haitham et al.)
- Astrazione di alto livello
- Non legato ad un problema di linguaggio o di implementazione, ma a un dominio di business.
- Analisi della struttura di un sistema e dei legami tra le sua parti
- Individuazione di pattern architetturali e di progetto

Analysis pattern

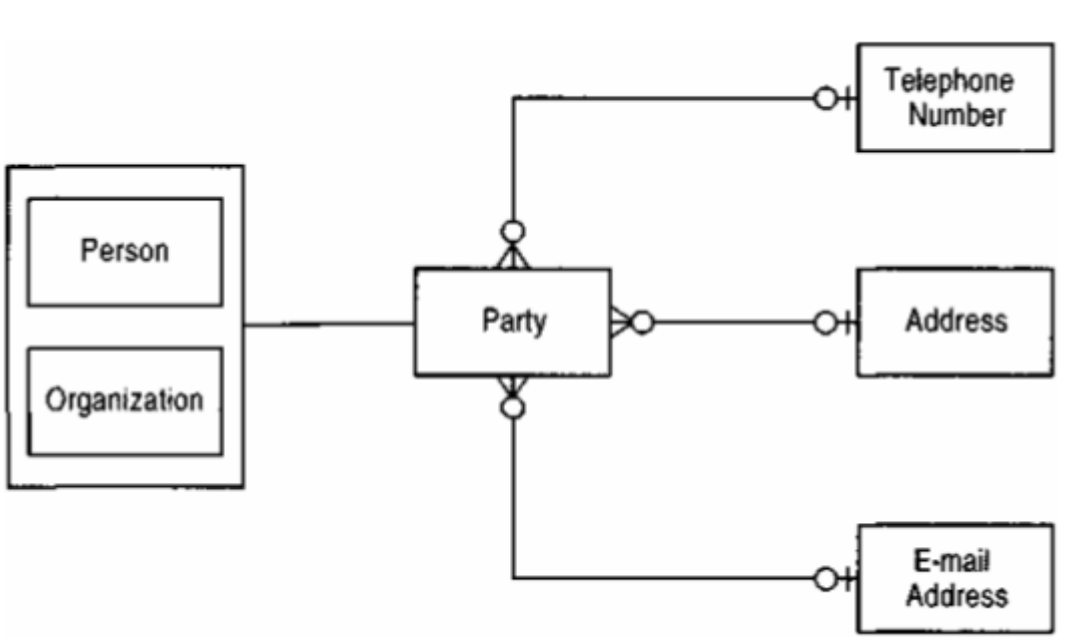


Event analysis pattern

"Catturare il ricordo di qualcosa di interessante per il dominio" (Martin Fowler)

- Assenza di metodi
- Uso di attributi stereotipati
- Dominio applicativo non indicato
- Possibile collegamento al pattern Observer

Analysis pattern

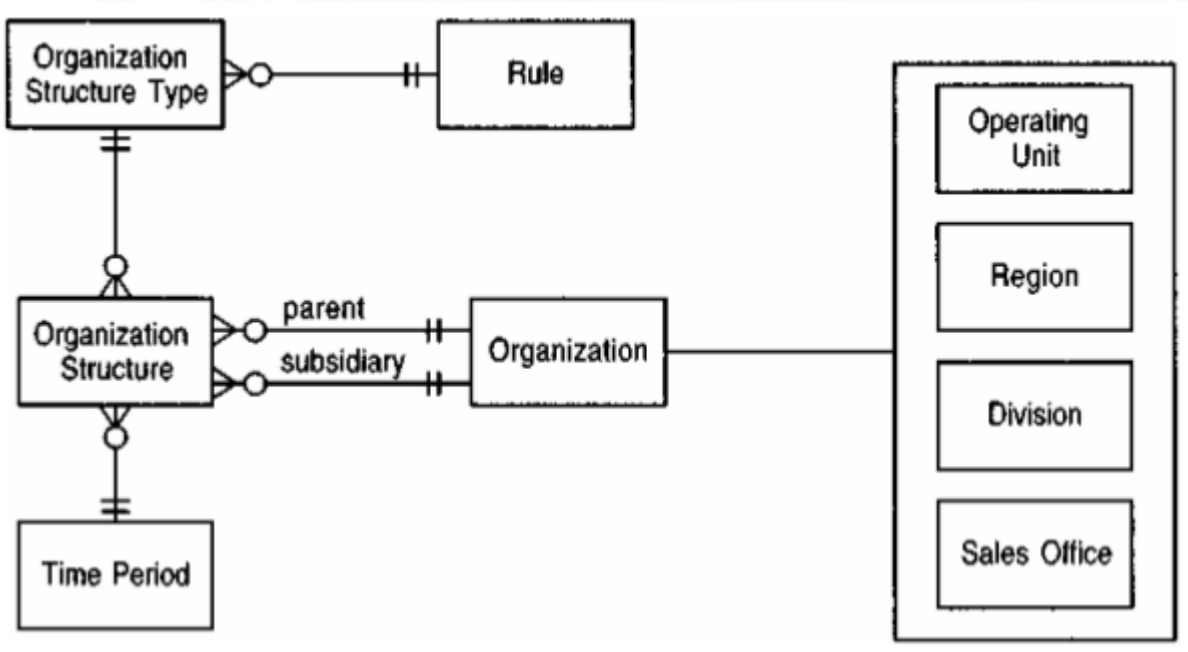


Party pattern

- Definisce relazioni tra soggetti diversi
- Esempio applicato ad un note-book

- Party come generalizzazione di Persona e organizzazione
- Consente di associare più elementi a concetti diversi

Analysis pattern



Organization Structure pattern

- Definisce il concetto di organizzazione
- Impone regole alla struttura dell'organizzazione
- Impone limiti temporali alle relazioniversi
- Analizza organizzazioni gerarchicamente strutturate

Analysis pattern

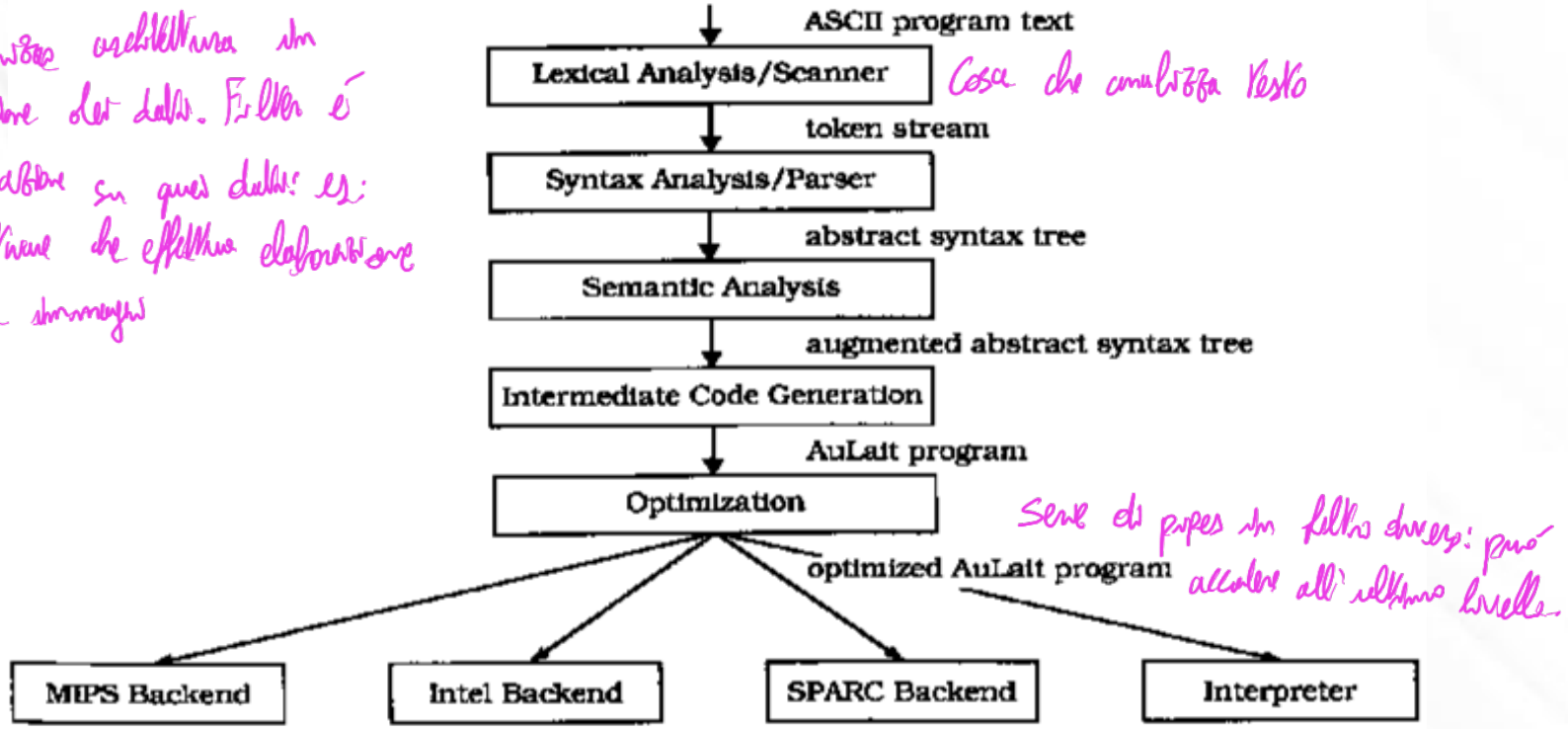
- Pro
  - Un singolo pattern può essere riutilizzato per sistemi con obbiettivi completamente differenti
  - Facile associazione con esempi concreti
- Contro
  - L'estrema semplificazione dei modelli può causare problemi in software complessi
  - Non sempre migliorano l'efficienza

Architectural pattern

- Livello più ampio rispetto ai design pattern
- Descrivono modelli per impostare l'organizzazione strutturale di un sistema software.
- Definizione di sottosistemi predefiniti, ruoli assunti e relazioni reciproche.
- Non rappresentano una semplice architettura
  - Risolve e sottolinea l'importanza di elementi di coesione in un'architettura
  - Più architetture possono implementare lo stesso pattern e condividerne le caratteristiche

Pipe and filters

- Fornisce una struttura per sistemi che elaborano uno stream di dati
- Ogni passo di elaborazione è incapsulato in un componente filtro
- I dati sono passati attraverso le pipe tramite filtri adiacenti.
- Ricombinando i filtri si ottengono famiglie di sistemi collegati



Pipe and filters: Structure

Card CRC: come si chiama, cosa fa, a chi è collegato

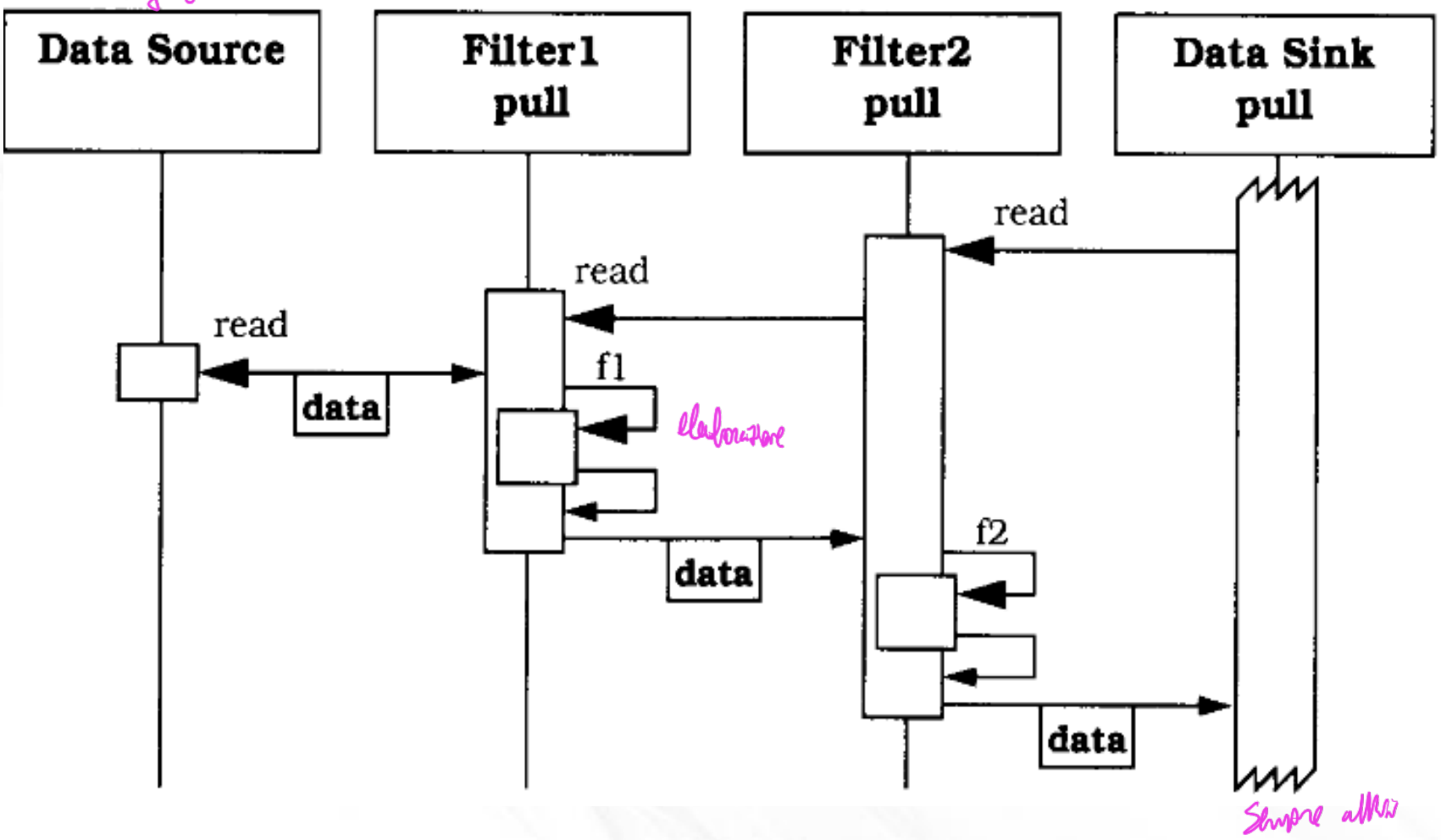
Class	Collaborators
Filter	• Pipe
Responsibility	
• Gets input data. • Performs a function on its input data. • Supplies output data.	

Class	Collaborators
Pipe	• Data Source • Data Sink • Filter
Responsibility	
• Transfers data. • Buffers data. • Synchronizes active neighbors.	

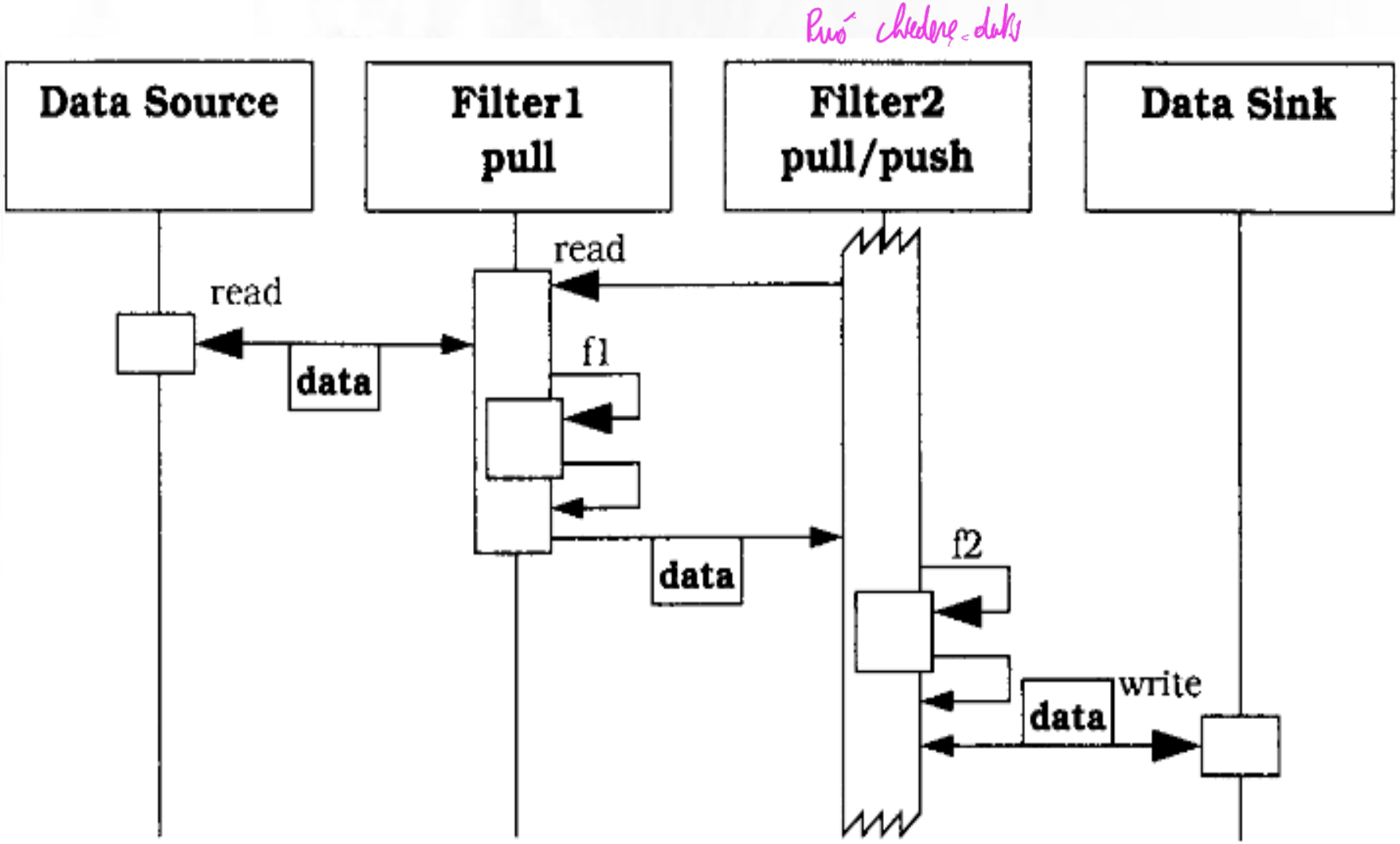
Class	Collaborators
Data Source	• Pipe
Responsibility	
• Delivers input to processing pipeline.	

Class	Collaborators
Data Sink	• Pipe
Responsibility	
• Consumes output.	

Pipe and filters: Behavior



Pipe and filters: Behavior



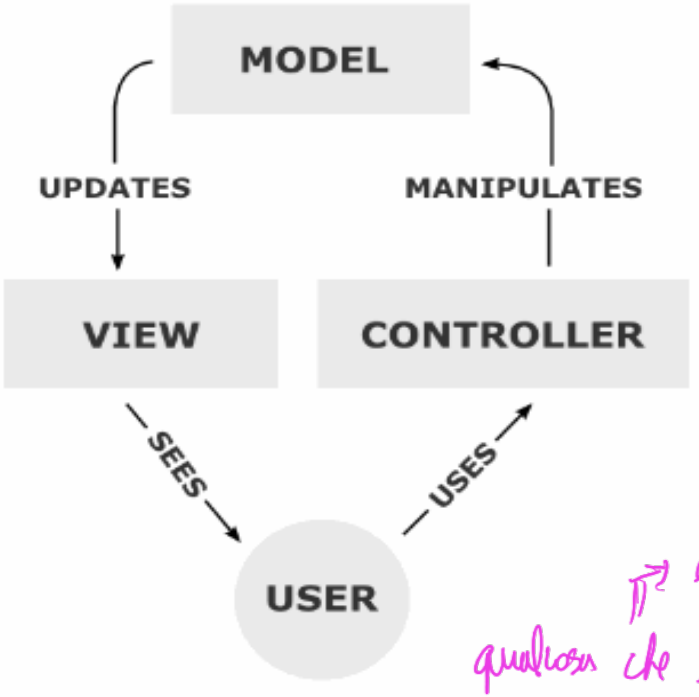
Pipe and filters: Known Uses

- Unix
- CMS Pipelines (extension IBM mainframes)
- LASSPTools (Numerical Analysis)
  - Graphical input devices (knobs or sliders)
  - Filters for numerical analysis and data extraction
  - Data sinks to produce animation from numerical
- Data streams
- Khoros : Image recognition...

Pipe and filters: Benefits

- No intermediate file necessary (but possible)
- Flexibility by filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping of pipeline
- Efficiency by parallel processing

Architectural pattern



Model-View-Controller (MVC) pattern

Separa la rappresentazione delle informazioni dagli usi che un utente può farne

- Un **controller** può inviare dei comandi ad una **view** associata per modificarne la presentazione o lo stato attraverso un **model**.
- Un **model** notifica le sue **view** associate e controlla quando esse subiscono dei cambiamenti. Le notifiche possono essere ignorate (modello passivo)
- Una **view** richiede al **model** informazioni necessarie a generare una rappresentazione corretta per l'utente

Model aggiorna view sulla base del controller.
Ci sono molte estensioni che collegano controller e view.

Model: ciò che rappresenta i dati: modello dei dati che sta manipolando sulla UI sistema.
Stato del sistema fa parte del modello
Ciascuno di questi pezzi (no model) può essere rappresentato da sottoclassi
Vista: qualsiasi modo che ho di visualizzare i dati da model.

Altri architectural pattern

- **Broker**: Fornire funzionalità a nodi diversi usando oggetti remoti
- **Client-Server** Api o web socket di zyppark
- **Naked Objects**: Generazione automatica delle interfacce di sistema (rafforzamento di incapsulamento e information hiding)
- **Event-driven architecture (EDA):** promuove la produzione, il rilevamento, il consumo e la reazione ad eventi
- **Blackboard**: usato nei sistemi di intelligenza artificiale, identifica un repository centrale della conoscenza (la blackboard) su cui i partecipanti possono scrivere.
- **Multilayer**: suddivisione dei componenti in base ai rispettivi scopi o caratteristiche peculiari.

Design pattern

- Descrizione o template, utilizzabile in più situazioni differenti, di come risolvere un problema
- Formalizzazione di “best-practice” che i programmatori devono implementare nelle loro applicazioni
- Mostrano relazioni e iterazioni tra classi o oggetti, senza specificare le classi o gli oggetti che saranno coinvolti in concreto.
- Difficilmente applicabili in linguaggi di programmazione funzionali.

In genere in linguaggi ad oggetti

↳ Creo solamente funzioni

Design pattern: tipologie

- **Algorithm strategy**: descrivono come sfruttare le caratteristiche di un'applicazione rispetto ad una piattaforma computazionale
- **Computational design**: identificazione di concetti chiave per la computazione
- **Execution**: supportano l'esecuzione di applicazioni, definendo strategie per la gestione di task e thread
- **Implementation strategy:** forniscono indicazioni sull'implementazione del codice e la gestione delle strutture dati.
- **Structural design**: supportano lo sviluppo delle strutture di alto livello delle applicazioni

Classificazione standard

- **Pattern creazionali**: nascondono i costruttori delle classi e creano interfacce per il loro uso. Supporto all'incapsulamento e all'information hiding
- **Pattern strutturali**: consentono di riutilizzare degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze
- **Pattern comportamentali**: forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti
- **Pattern di concorrenza**: usati nel caso di processi che operano contemporaneamente su dati condivisi, indicando metodi di sincronizzazione

Creational pattern

- **Abstract factory**: fornisce un'interfaccia per creare famiglie di oggetti connessi tra loro, in modo che non ci sia necessità da parte degli utilizzatori di specificare i nomi delle classi concrete all'interno del codice
- **Builder**: separa la costruzione di un oggetto complesso dalla sua rappresentazione
- **Factory method**: fornisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale istanziare.
- **Lazy initialization**: istanzia un oggetto solo nel momento in cui deve essere usato per la prima volta.
- **Prototype**: permette di creare nuovi oggetti clonando un oggetto iniziale, o prototipo.
- **Singleton**: assicura che di una classe possa esistere una sola istanza.

Abstract Factory

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Senza dover le classi concrete a cui vado ad accedere. Classi diverse, ma che lavorano a livello di interfaccia.

Alternative Name:

- Kit

Abstract Factory

Problem:

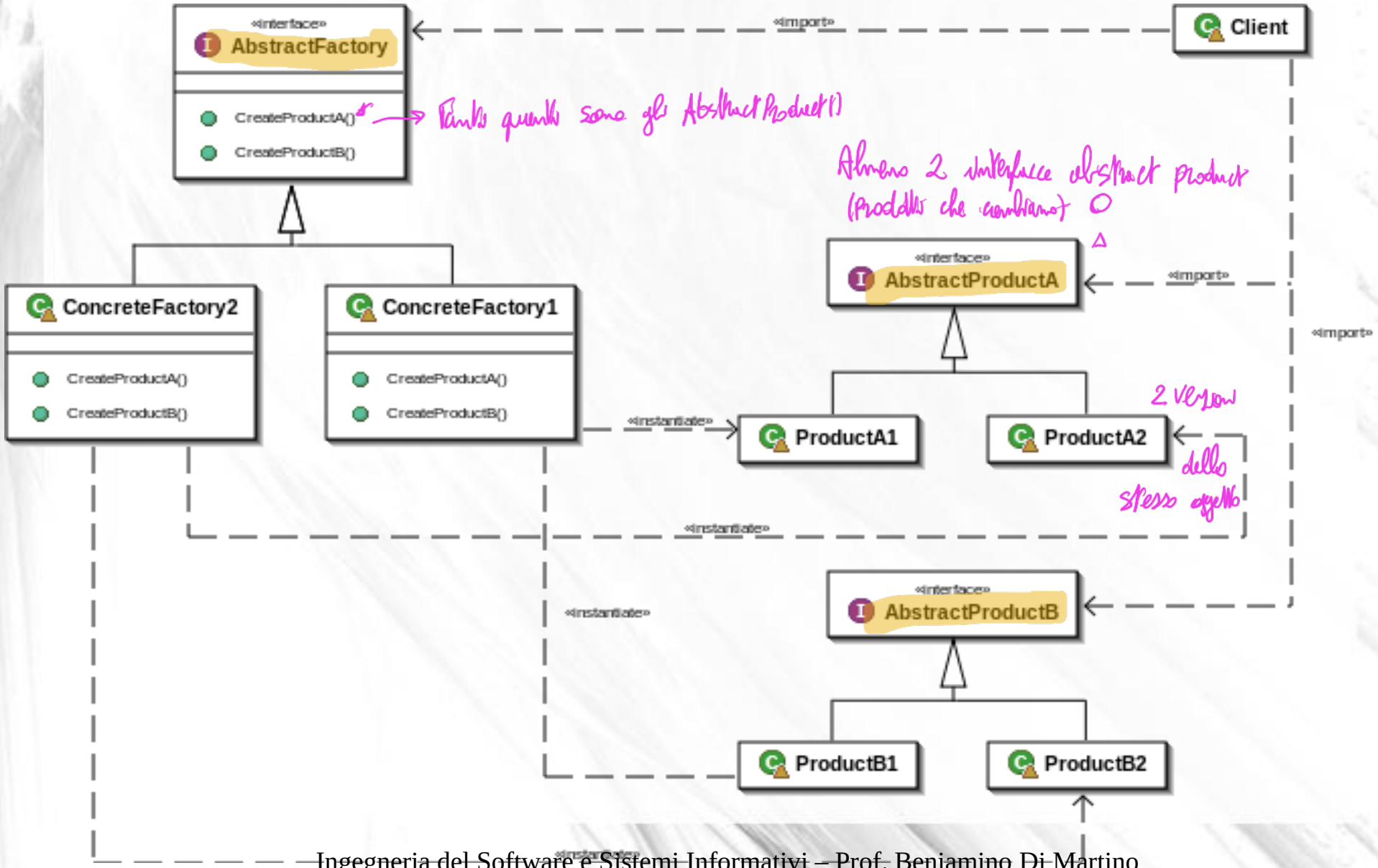
- How should one go about designing applications which have to be adopted to different Look and feel standards? Look and feel differente a secondo del sistema

Solution:

- One could solve this problem by defining abstract widgets factory class that declares an interface for creating each basic kind of widgets (control) .

- There is also an abstract class for each kind of widget, and concrete sub classes implement widgets for specific look and standards.

Abstract Factory



* Può creare più tipologie di prodotto. ProductA1 e A2.
Per ciascuna una concrete Factory, che implementa i metodi dell'interfaccia.

Δ A è JButton, B JTextArea
1 per Unix, 2 Windows.

Factory dice crea JButton / crea JArea. A runtime si sceglie cosa creare.

AbstractFactory sceglie una tra i prodotti A, B ... numerati.

o Se avessi solo un prodotto, non serve questo sistema. Implementato con classi diverse.

Abstract Factory

Applicability

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.

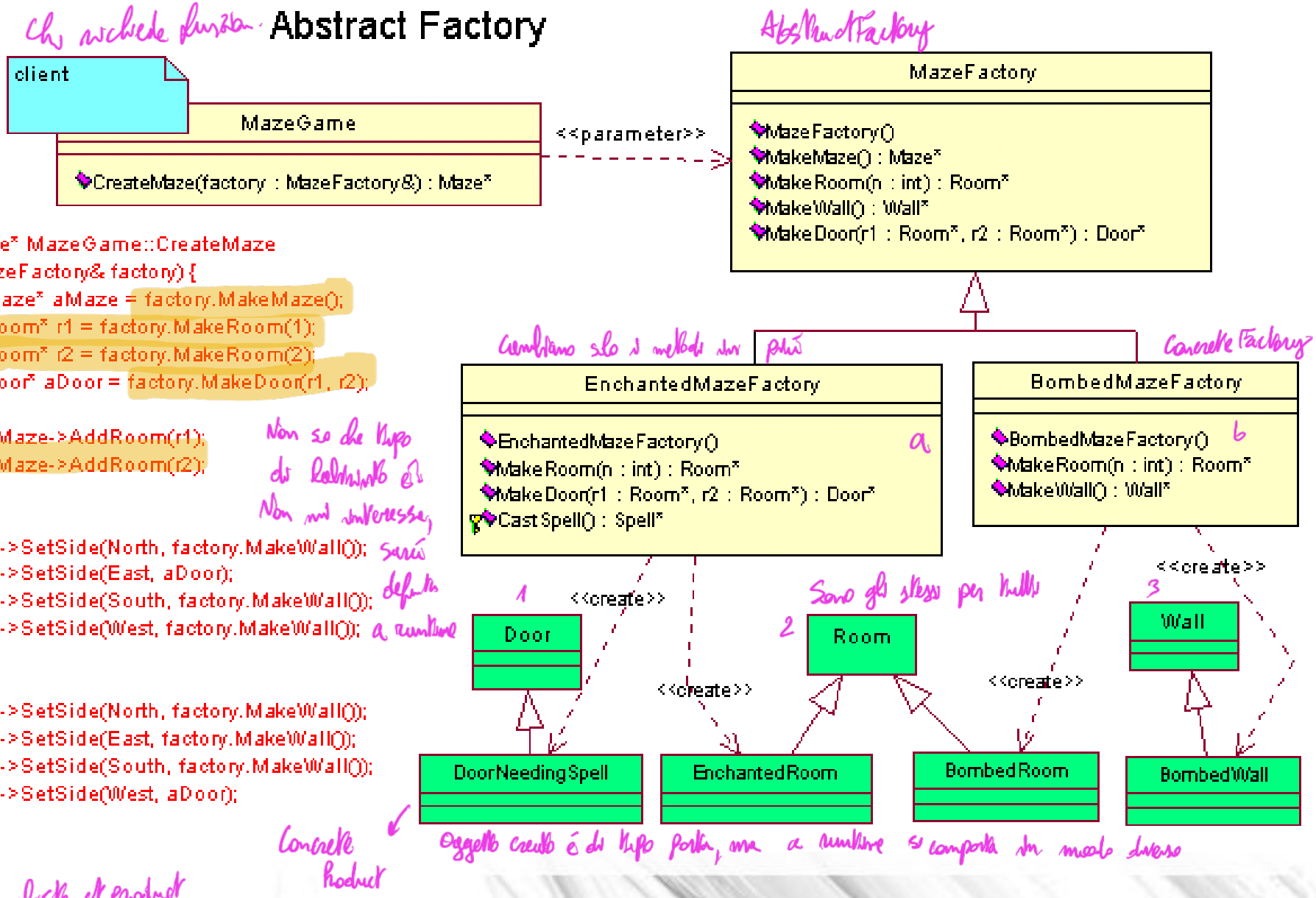
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Abstract Factory

Consequences

- It isolates concrete classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new kinds of products is difficult.

Abstract Factory



Abstract Factory

Related Patterns

- AbstractFactory classes are often implemented with factory methods, but they can also be implemented using Prototype.

Factory Method

Come faccio a far decidere quale istanza dell'oggetto andare a selezionare?
Uso un pattern per l'implementazione di questo oggetto

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Per scegliere dove fare cosa. Post factory class

- Factory Method lets a class defer instantiation to subclasses.

Factory Method

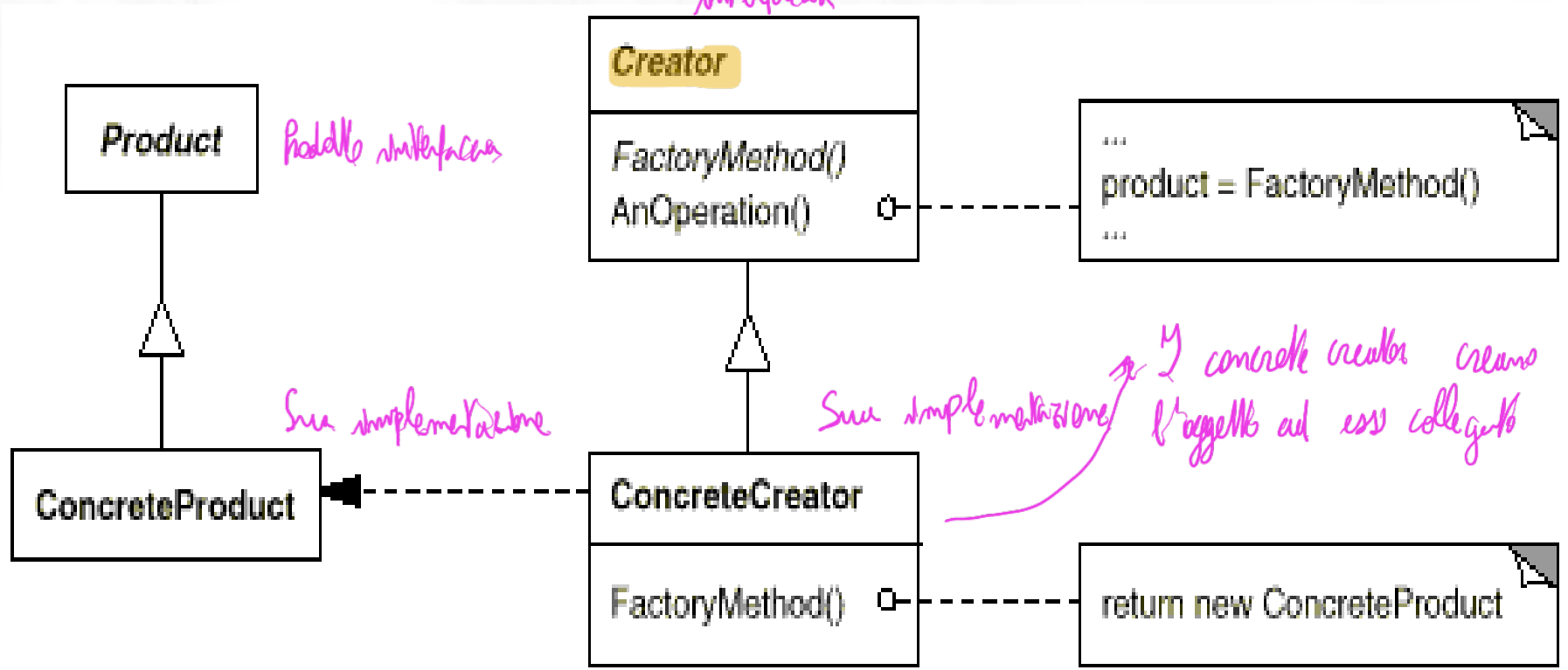
Applicability

- A class can't anticipate the class of objects it must create.

- a class wants its subclasses to specify the objects it creates.

- hiding which helper subclass has been delegated responsibility for implementing a task.

Factory Method Structure



Factory Method

Consequences

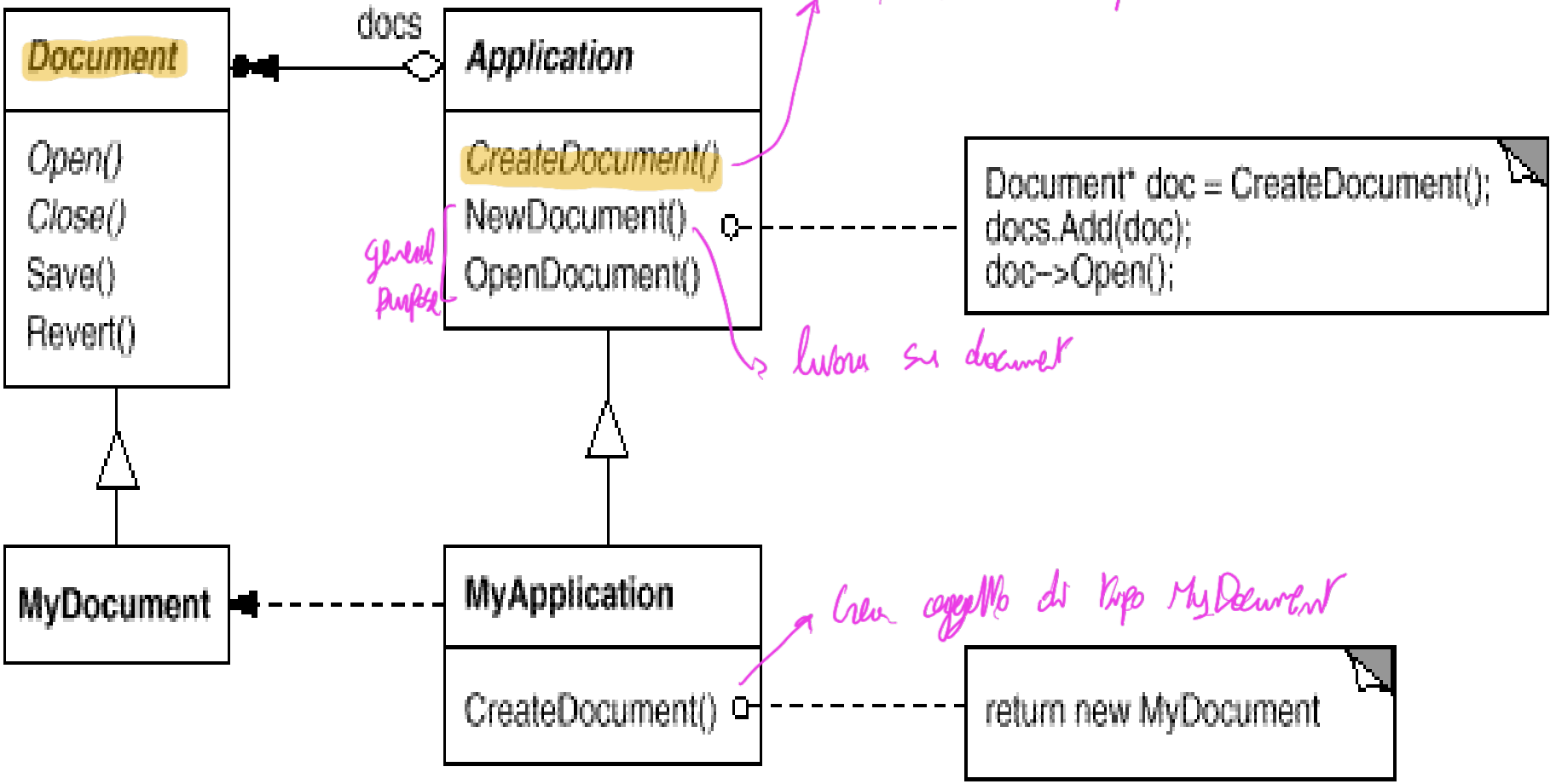
- *Eliminates binding to a specific implementation at instance creation.*
- *Can increase sub-classing.*
- *Provides hooks for subclasses*
- *Connects parallel class hierarchies*

Factory method

Implementation

- *With or without default implementation.*
- *Parameterized factory methods.*
- *Using templates to avoid sub-classing.*
- *Naming conventions.*

Examples


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

Prototype

- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

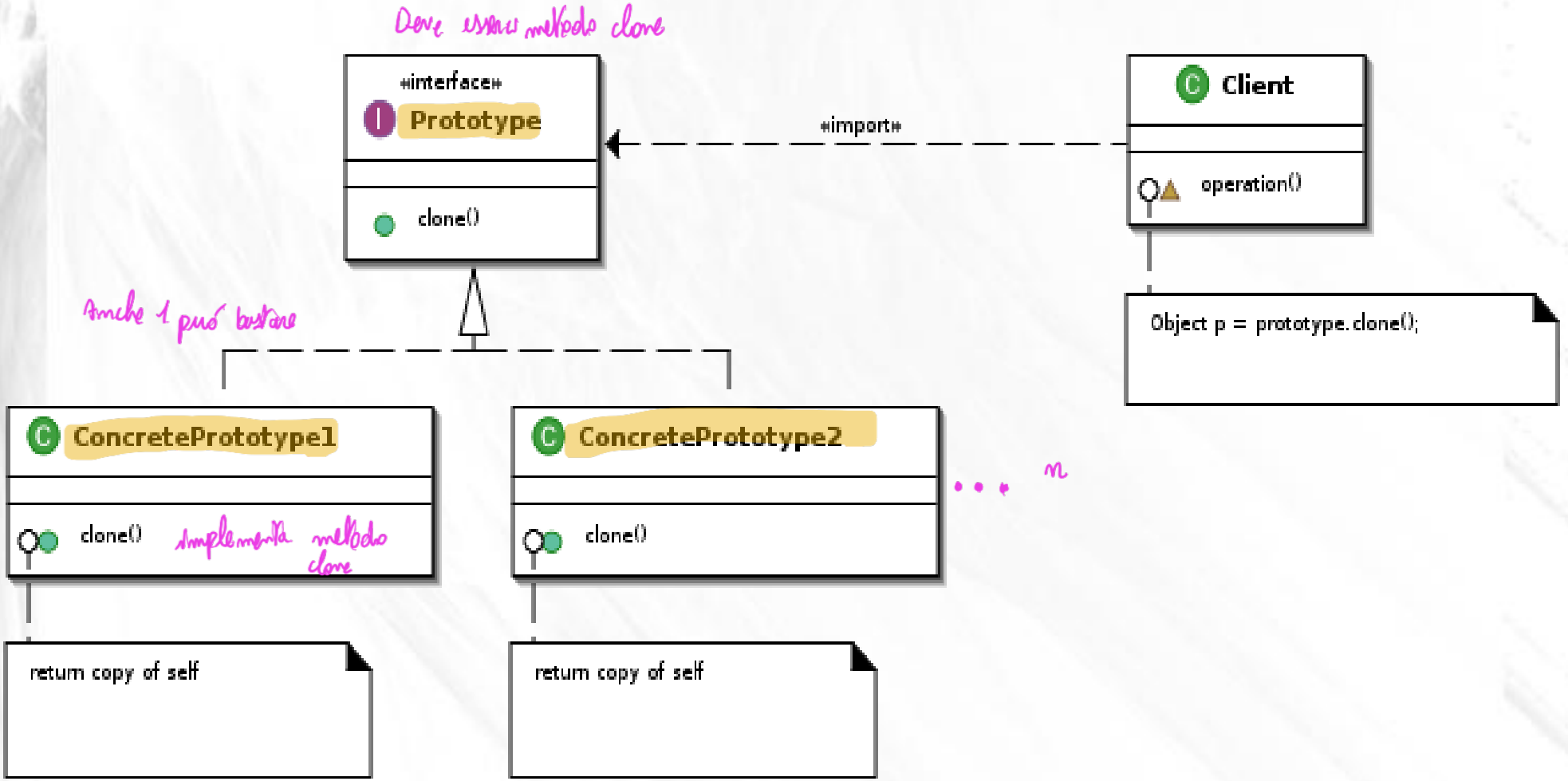
Motivation

- Sometimes, it becomes necessary to copy or clone an “already grown” object rather than instantiating it and setting its values.
- Consider the case of Photoshop. A graphics designer add an image to the canvas. Then, he adds a border to it. Then, he gives it a bevel effect. Finally, he sets its transparency to 50%. Now, he selects this image, presses Ctrl+C, and presses Ctrl+V 5 times.
- What will happen? The same image will be pasted 5 times. So, there will be 6 identical images on the canvas at different locations.
- How will you achieve this programmatically? “New”-ing from scratch at client side? Use a factory?

Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Prototype


«interface»
Prototype
clone()
«import»
Client
operation()
Object p = prototype.clone();
ConcretePrototype1
clone()
return copy of self
ConcretePrototype2
clone()
return copy of self

Participants

- **Prototype**
  - declares an interface for cloning itself.
- **ConcretePrototype**
  - implements an operation for cloning itself.
- **Client**
  - creates a new object by asking a prototype to clone itself.

Collaborations

- A client asks a prototype to clone itself.

Consequences

- *Adding and removing products at run-time.*
- *Specifying new objects by varying values.*
- *Specifying new objects by varying structure.*
- *Reduced subclassing.*
- *Configuring an application with classes dynamically.*
- The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references.

Implementation

- *Using a **prototype manager** (a prototype register).*
- *Implementing the Clone operation:*
  - **copy constructor** implementation
  - **"shallow copy versus deep copy"** problem
- *Initializing clones.*

Builder

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

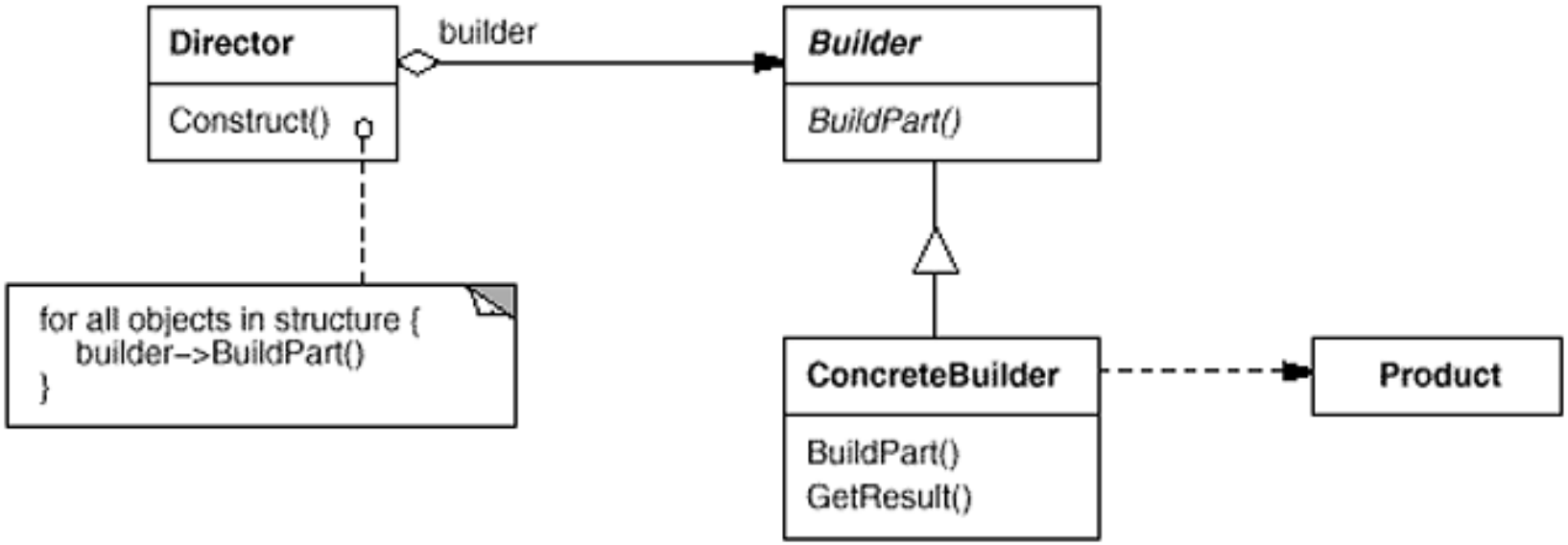
Applicability

Use the Builder pattern when

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Builder

Structure



Participants

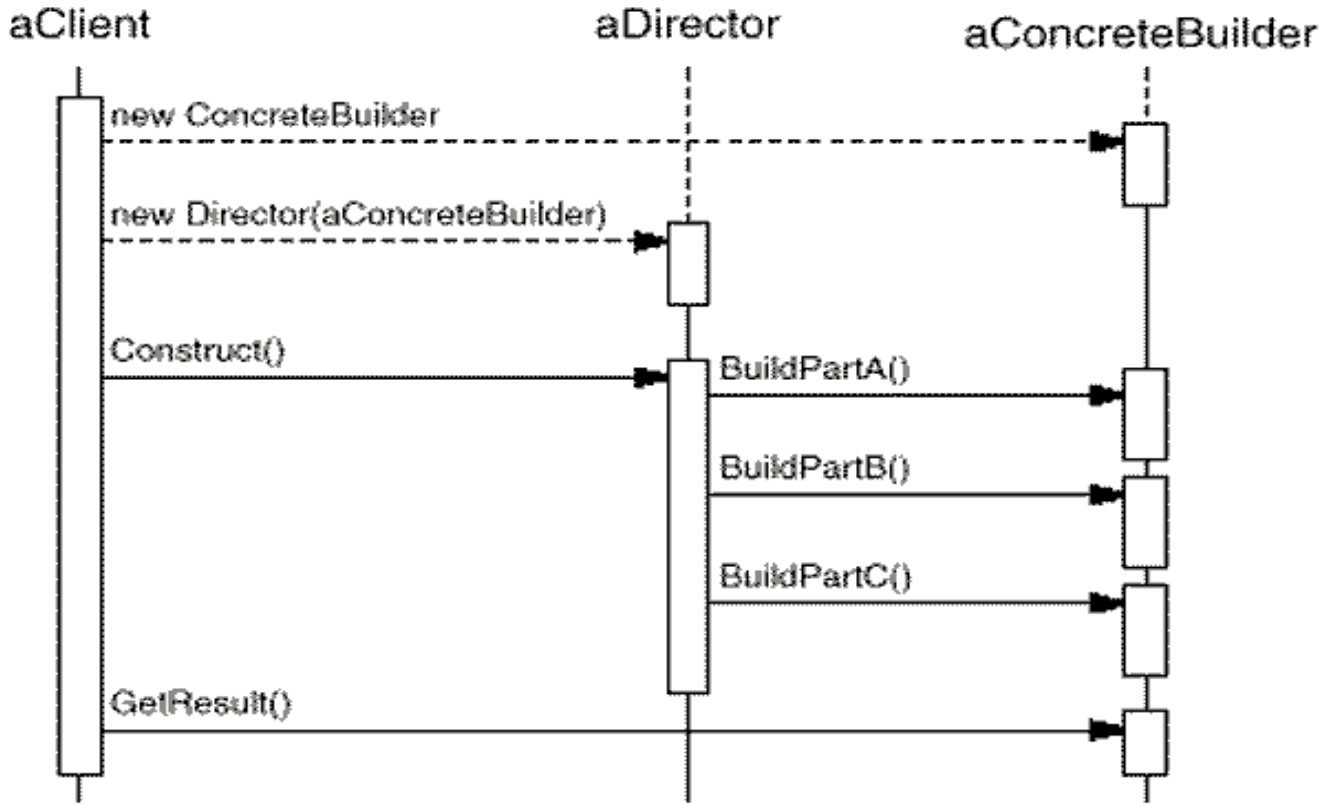
- **Builder**
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product
- **Director**
  - constructs an object using the Builder interface.
- **Product**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

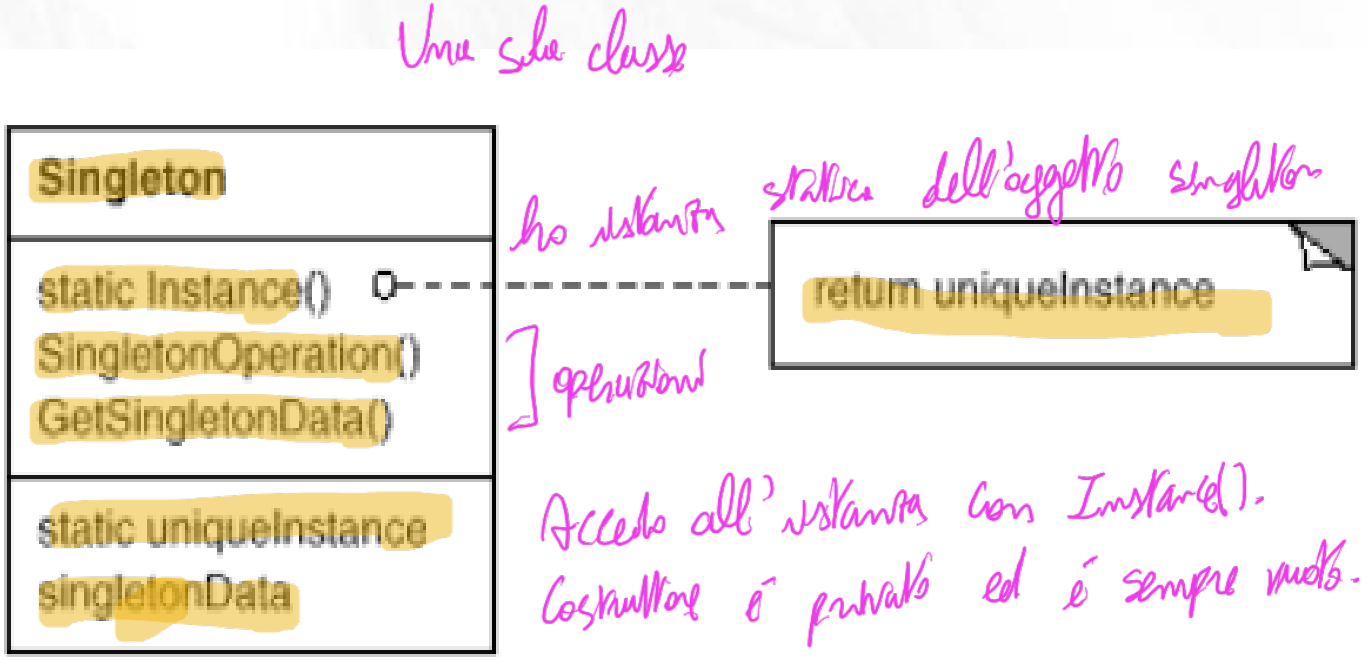
Collaborations

- The following interaction diagram illustrates how Builder and Director cooperate with a client.



Singleton

Structure



Singleton

Participants

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Structural pattern

Come devo strutturare gli oggetti al loro interno

- **Adapter**: converte l'interfaccia di una classe in una interfaccia diversa.
- **Bridge**: separa l'astrazione di una classe dalla sua implementazione, permettendo loro di variare indipendentemente.
- **Composite**: dà la possibilità all'utilizzatore di manipolare oggetti in modo uniforme, organizzandoli in una struttura ad albero.
- **Container**: offre una soluzione alla rottura dell'incapsulamento causata dell'ereditarietà.
- **Decorator**: consente di aggiungere metodi a classi esistenti durante il run-time.

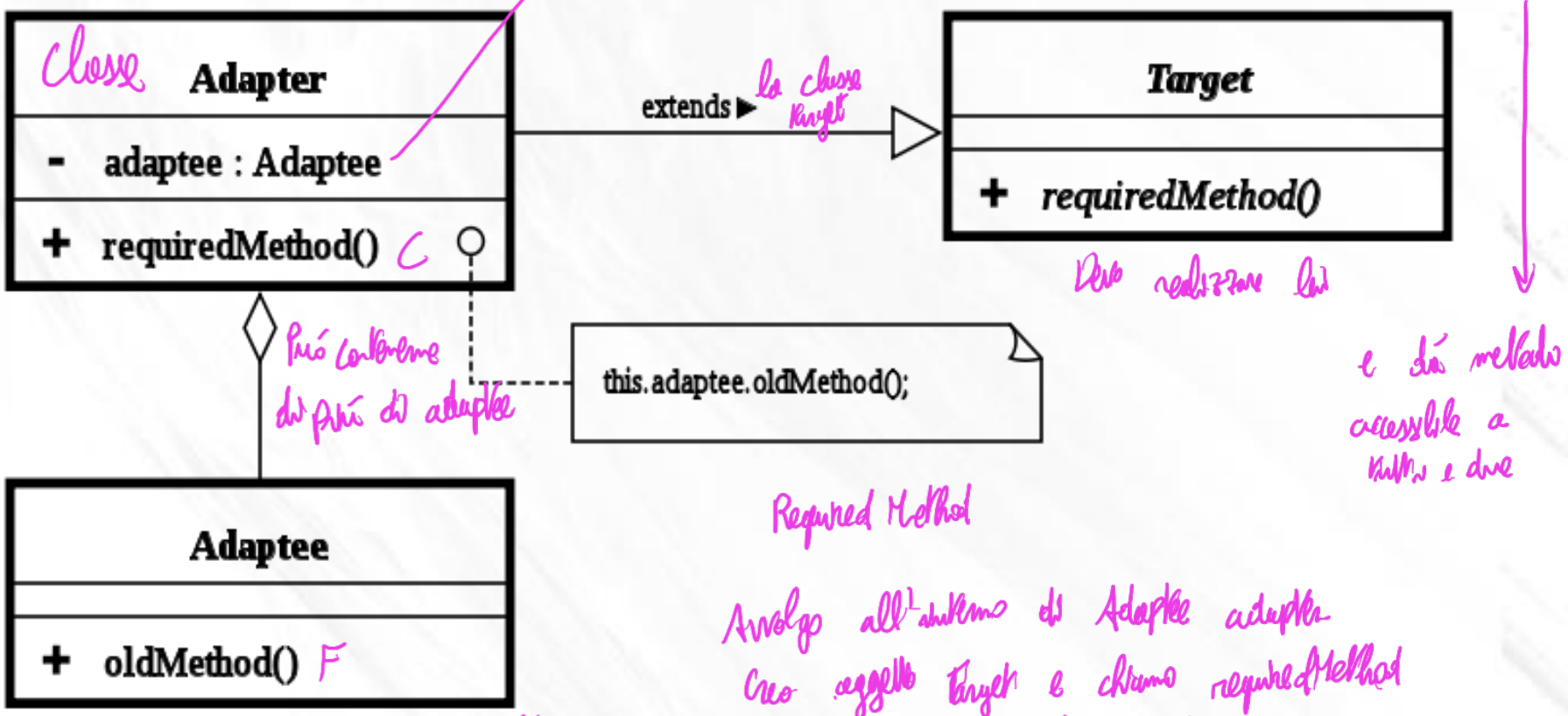
Structural pattern

- **Façade**: definisce interfacce più semplici per l'accesso a sottosistemi che espongono interfacce complesse e diverse tra loro
- **Flyweight**: separa la parte variabile di una classe da quella che può essere riutilizzata.
- **Proxy**: rappresenta oggetti di accesso difficile o che richiedono un tempo importante per accesso o creazione.

Adapter Pattern

- a.k.a. Wrapper
- Has both **class** and **object** forms
- Adapts the interface of one class into that of an otherwise incompatible class
- Use to:
  - make use of a class with incompatible interface
  - create reusable class that will play nice with others
  - use several subclasses without having to subclass each one to adapt their interface

Adapter


Adapter
- adaptee : Adaptee
+ requiredMethod()
extends
Target
+ requiredMethod()
this.adaptee.oldMethod();
Adaptee
+ oldMethod()

Adapter Pattern

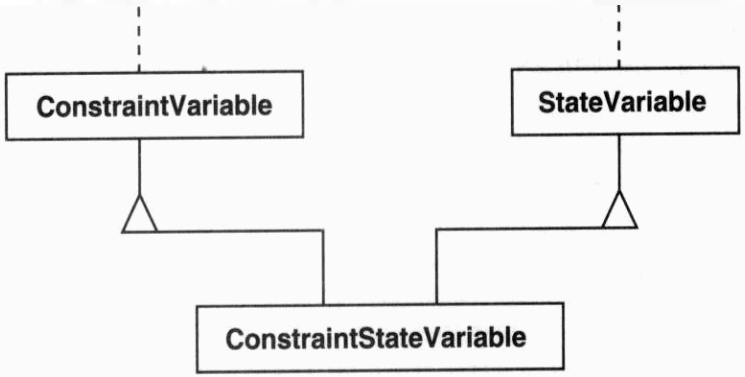
- Participants
  - Target
    - defines the domain-specific interface that Client uses
  - Adaptee
    - defines existing interface that needs adapting
  - Adapter
    - adapts the interface of Adaptee to the Target interface
- Collaborations
  - Clients call operations on Adapter instance, which in turn calls the appropriate Adaptee operations

Adapter Pattern

- **Consequences**
  - Class form
    - commits to a concrete Adaptee class, so won't work if we want to adapt a class as well as subclasses (use object form)
    - Adapter can override Adaptee's behavior
  - Object form
    - allows single Adapter to work with many Adaptee subclasses; can add functionality to all Adaptees at once
    - harder to override Adaptee behavior – have to subclass Adaptee and make Adapter refer to the subclass

Adapter Pattern

- Other Issues
  - How much work does Adapter do?
    - simple: changing names of otherwise identical operations
    - complex: supporting new functionality
  - Pluggable Adapters
    - use interface adaptation to make class more reusable
  - Two-way Adapters
    - a class adapter conforms to both incompatible classes
    - use multiple inheritance



Adapter Pattern

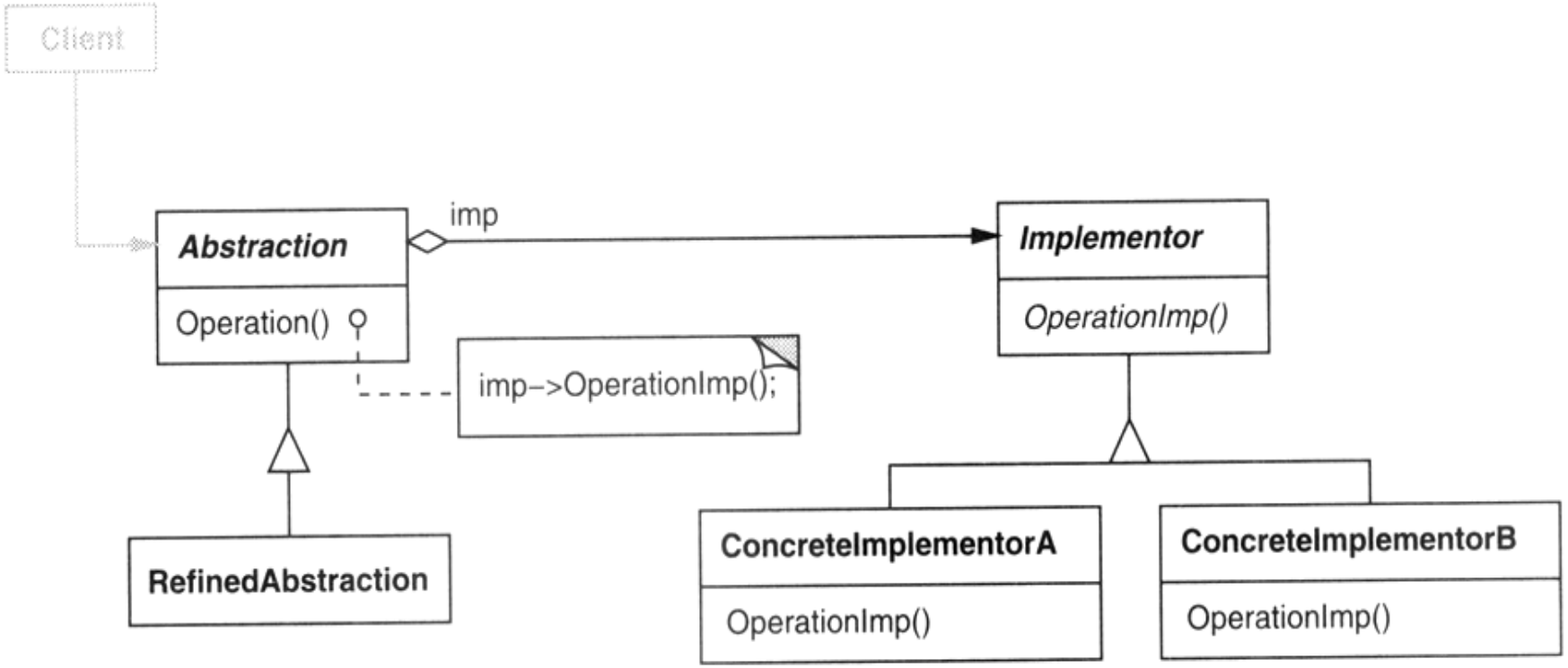
- **Related Patterns**
  - **Bridge –** similar structure, but meant to separate interface from its implementation, rather than change interface of existing object
  - **Decorator –** enhances object without changing its interface. Thus more transparent and supports recursive composition
  - **Proxy –** defines surrogate for another object without changing its interface

Bridge Pattern

- a.k.a. Handle/Body
- Decouples abstraction from its implementation
- Use to:
  - avoid binding abstraction to implementation (i.e. want to choose different implementations at runtime)
  - make both abstraction and implementations independently extensible
  - change implementation or abstraction without impacting clients, or hide them from clients
  - share implementation among clients without letting them know

Bridge Pattern

- **Structure**:



Bridge Pattern

- **Participants**
  - **Abstraction**
    - defines abstraction's interface
    - maintains reference to Implementor instance
  - **RefinedAbstraction**
    - extends interface defined by Abstraction
  - **Implementor**
    - defines interface for implementation classes
    - typically provides more primitive operations than Abstraction
  - **ConcreteImplementor**
    - implements Implementor interface
- **Collaborations**
  - Abstraction forwards client requests to its Implementor object

Bridge Pattern

- **Consequences**
  - decouples interface and implementation
    - allows selecting implementation at runtime
    - avoids compilation dependencies on implementation
  - improves extensibility
    - Abstraction and Implementation can be extended independently
  - hides implementation details from clients

Bridge Pattern

- Related Patterns
  - **Abstract Factory** can create and configure Bridges
  - **Adapter** makes existing unrelated classes work together; whereas Bridge is used in design in effort to avoid needing Adapters later

Composite Pattern

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable

Solution:

The composite patterns describe how to use recursive composition so that the client's don't have to make this distinction. One Key aspect of the Composite patterns is an abstract class that represents both primitives and their containers.

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects.

Subclass the Primitive and Composite classes off of the Component class.

Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

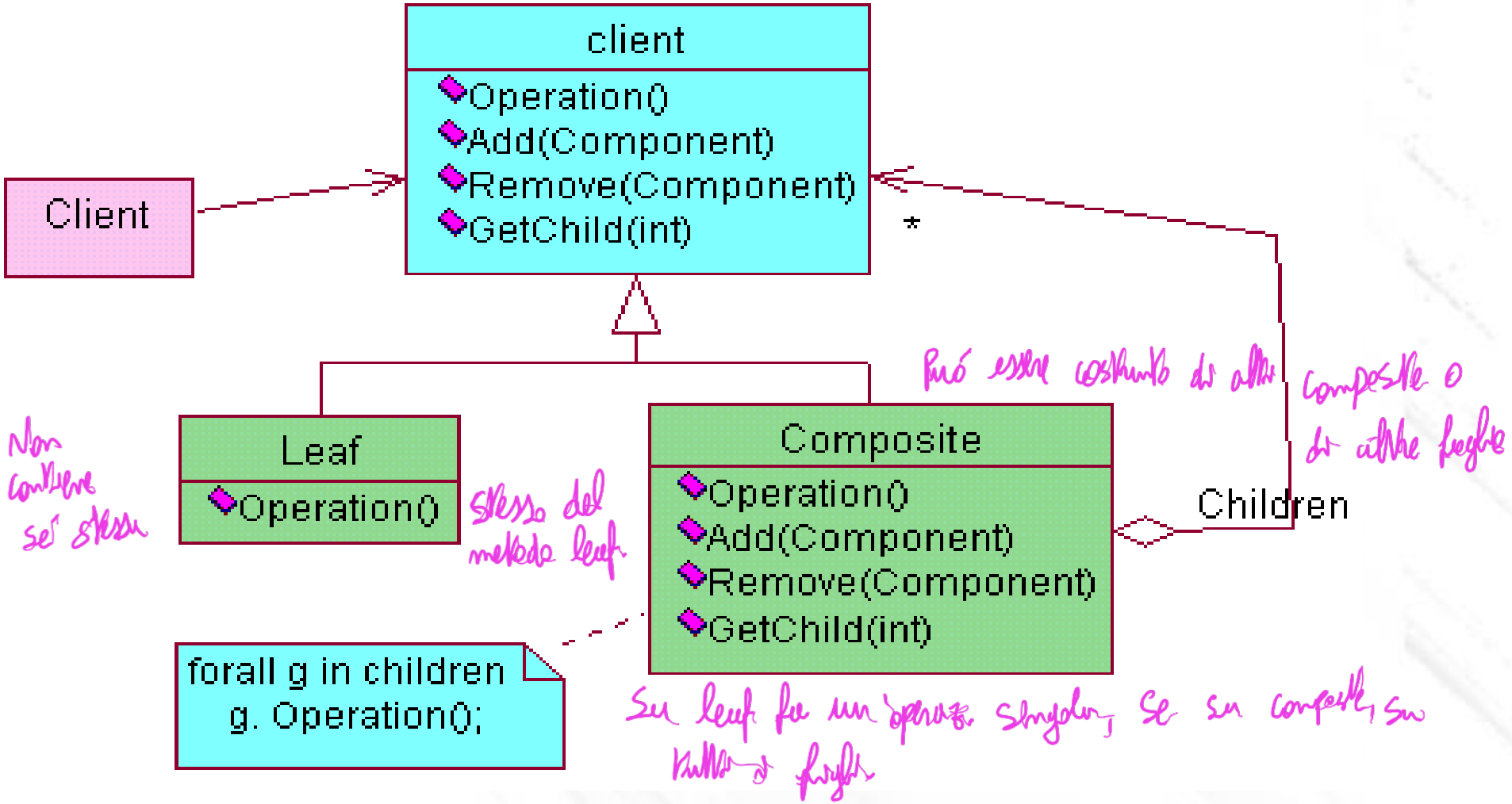
Applicability

Use the Composite pattern when

You want to represent part-whole hierarchies of objects.

You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure


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

Composite = Libro Leaf = Frase.

↓

Composite = Capitolo

↓

Composite = Pagina

↓

Composite = frase.

Leggi: Libro: ∀ capitolo leggi capitolo

Capitolo: ∀ pagina leggi pagina

Pagina: ∀ frase leggi frase

Frase: leggi frase

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

Defines class hierarchies consisting of primitive objects and composite objects recursively: Wherever client code expects a primitive object, it can also take a composite objects

Make the client simple: clients treat composite and leaf uniformly. This simplifies client code, because it avoids having to write tag-and-case-statements-style functions over the classes that define the composition.

Makes it easier to add new kinds of components :Newly defined composition or leaf subclasses work automatically

Decorator

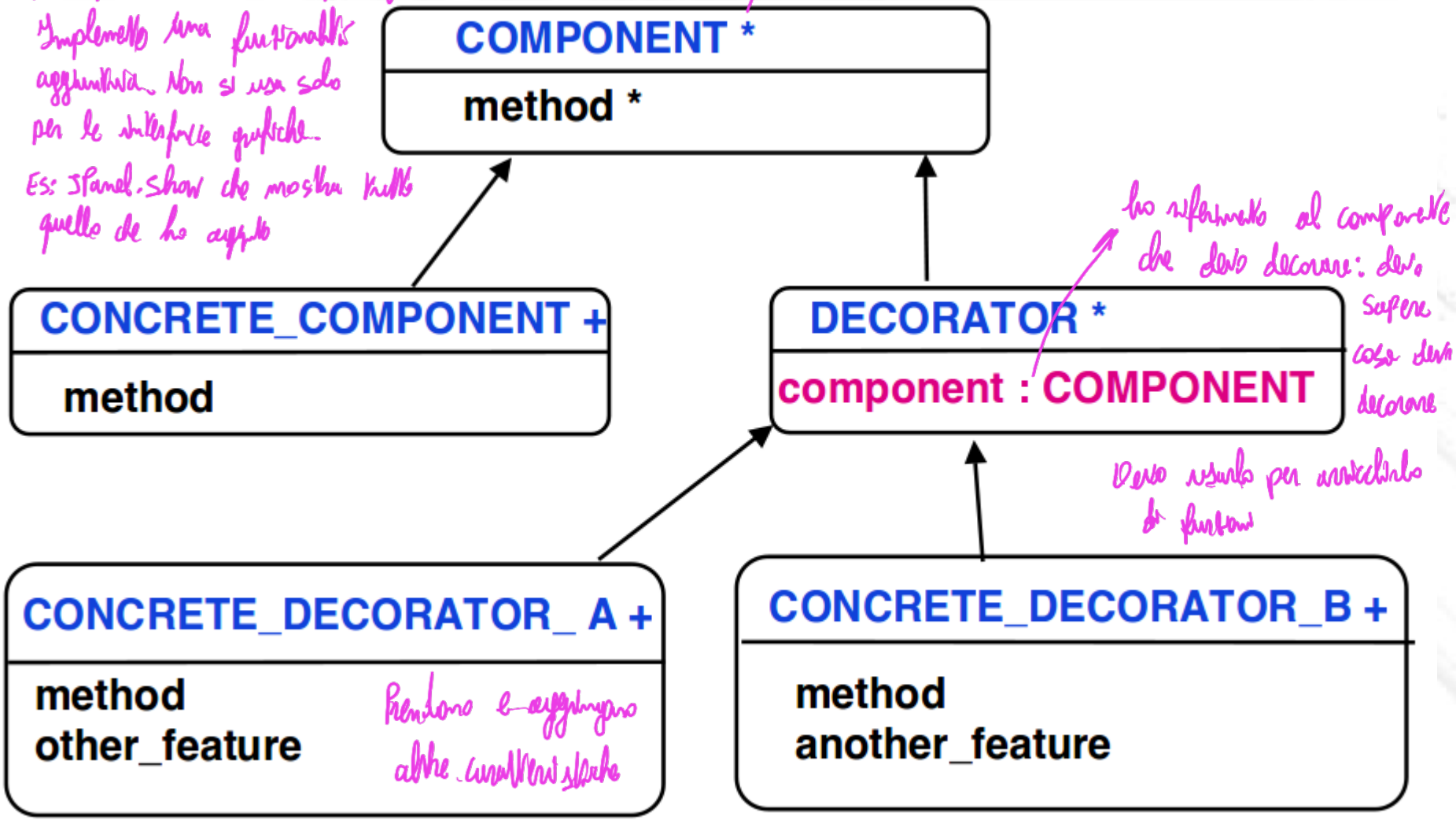
Intent

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

Motivation - Applicability

- Need to add responsibility to individual objects not to entire classes
- Add properties like border, scrolling, etc to any user interface component as needed
- Enclose object within a decorator object for flexibility
- Nest recursively for unlimited customization

Decorator


COMPONENT *
method *
CONCRETE_COMPONENT +
method
DECORATOR *
component : COMPONENT
CONCRETE_DECORATOR_ A +
method
other_feature
CONCRETE_DECORATOR_B +
method
another_feature

Decorator

Component: Defines the interface for objects that can have responsibilities added to them dynamically

Concrete component: Defines an object to which additional responsibilities can be attached

Decorator: Maintains a reference to a component object and defines an interface that conforms to COMPONENT

Concrete decorator: Add responsibilities to the component

Decorator

Consequences

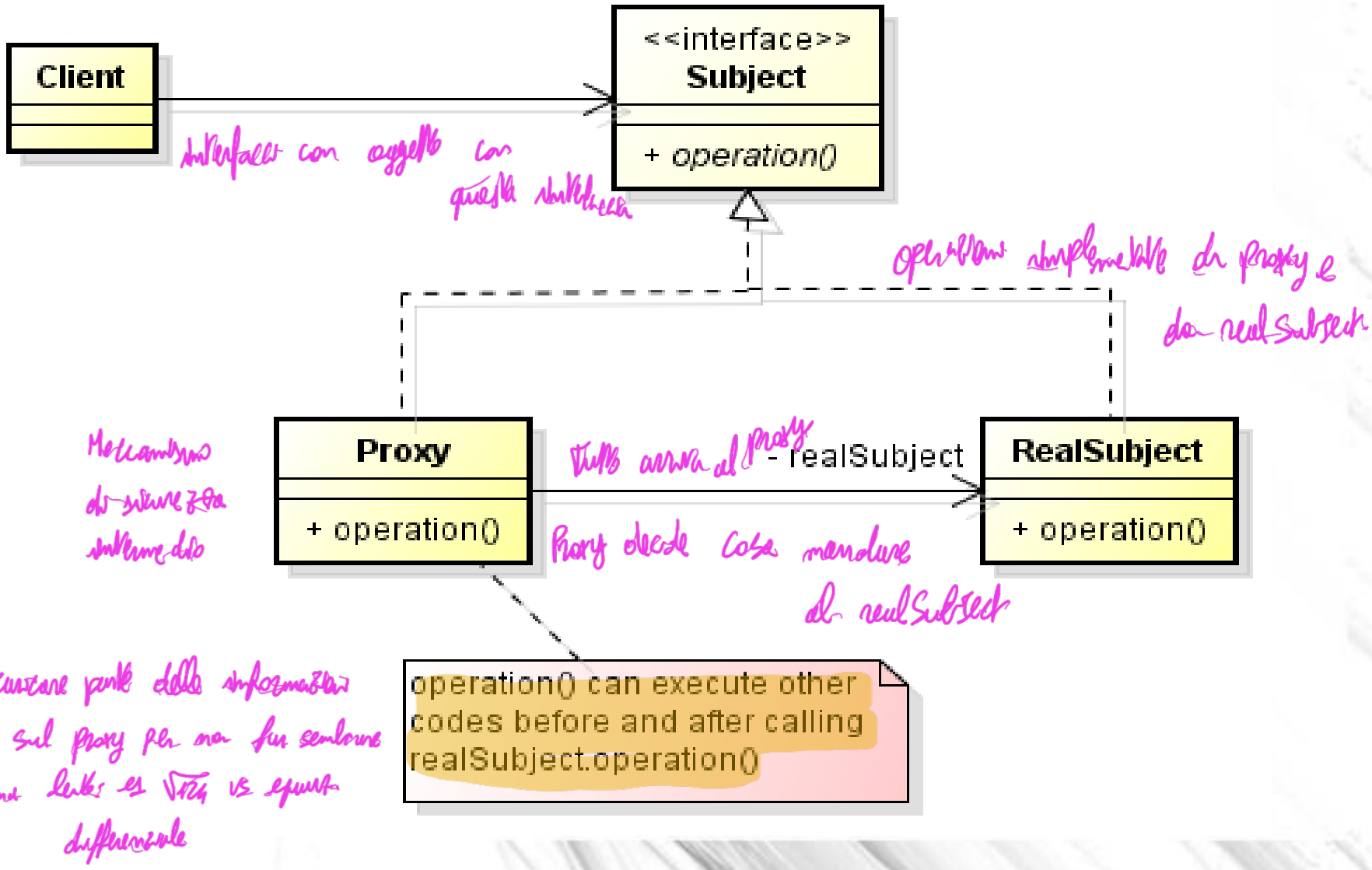
- More flexible than static inheritance
- Can add and remove responsibilities dynamically
- Can handle combinatorial explosion of possibilities
- Avoids feature laden classes high up in the hierarchy
- Pay as you go when adding responsibilities
- Can support unforeseen features
- Decorators are independent of the classes they decorate
- Functionality is composed in simple pieces

Decorator

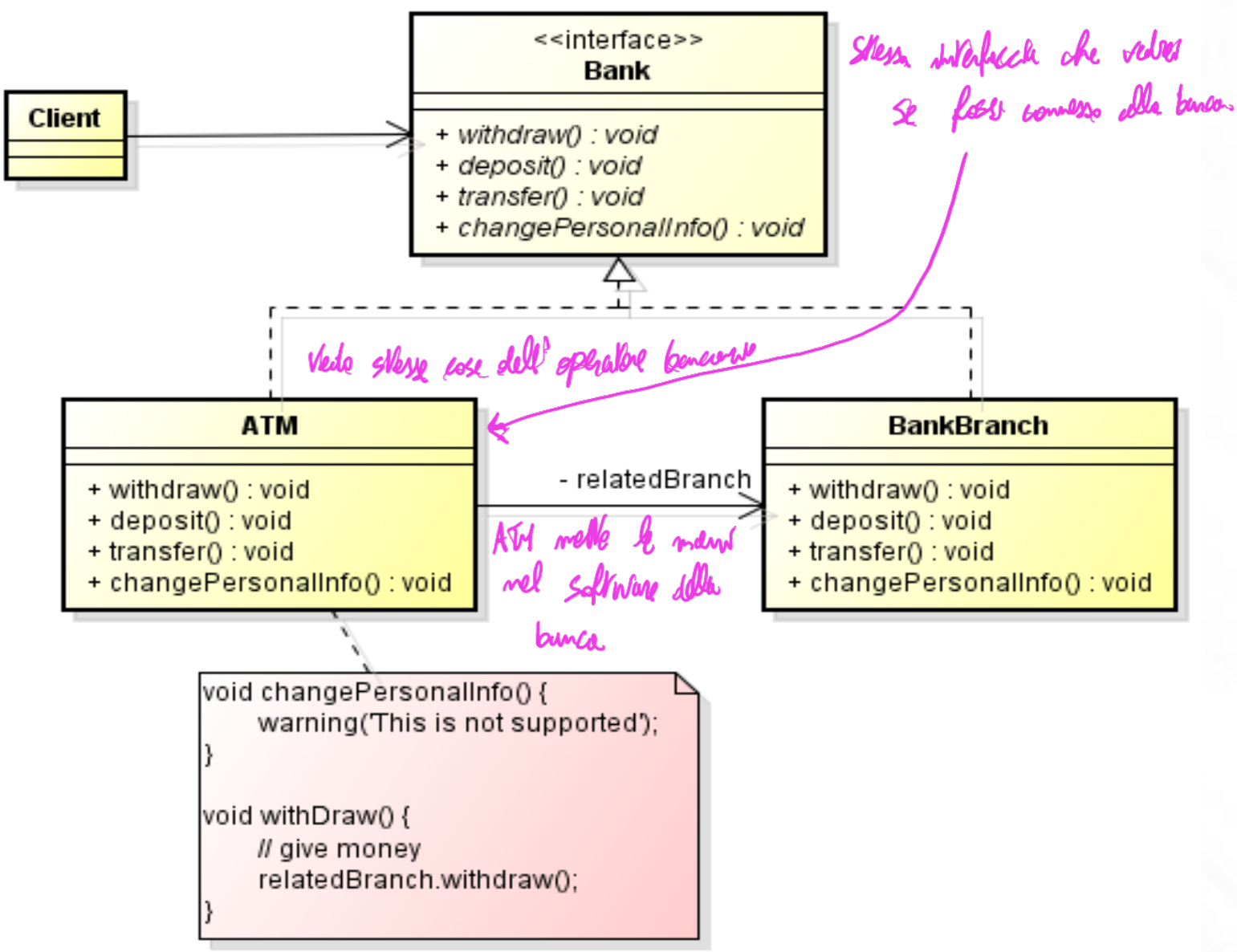
Related Patterns

- **Adapter** changes interface to an object, while Decorator changes an objects responsibilities
- Decorator is a degenerate **Composite** (only one component)
- **Strategy** lets you change the internals of an object, while Decorator changes the exterior

Proxy


Client
<<interface>>
Subject
+ operation()
Proxy
+ operation()
- realSubject
RealSubject
+ operation()
operation() can execute other codes before and after calling realSubject.operation()

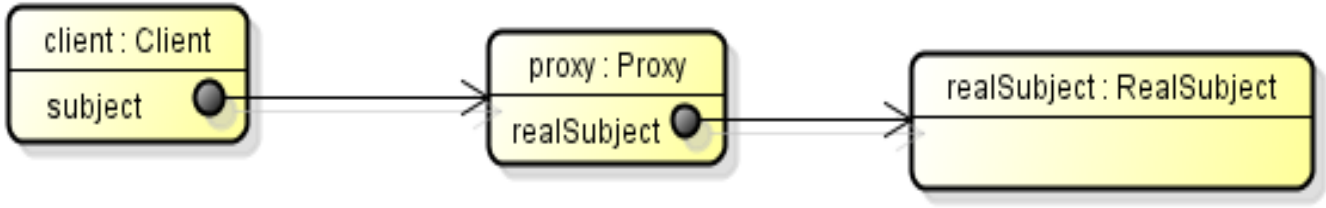
Proxy: Implementation

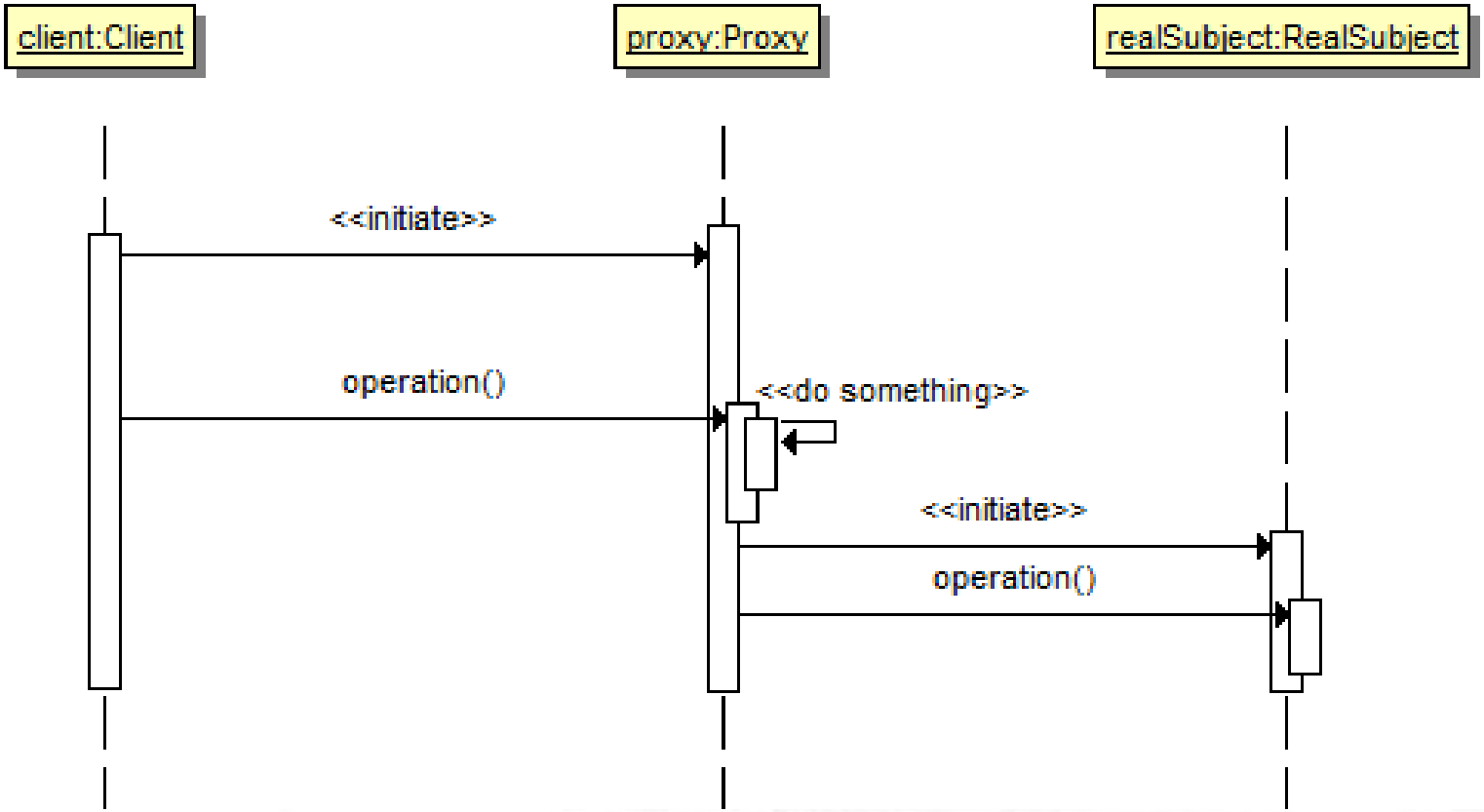


Participants

- Subject (Bank)
  - The common interface for the proxy and the real subject
- Real Subject (Bank Branch)
  - The concrete subject that implements the interface.
- Proxy (ATM)
  - Provides the same interface as Real Subject (or a subset)
  - Maintains a reference to the Real Subject
  - Since it doesnot have all the data as Real Subject (a lightweight), it can do many things faster than Real Subject.
  - Then invokes Real Subject (if needed).

Object & Sequence Diagrams





Consequences

- A level of indirection while accessing real subject.
  - We can think this as another tier between real subject and us.
- We want to support heavyweight objects, but we want to create only when they are requested.
  - Memory will not get full.
  - Programs will run faster.
- It is like giving a small example of the same subject.
  - Most of the times, proxy will have same interfaces as subject.
  - Programmers can declare variables without caring whether a proxy or the heavyweight will be put.

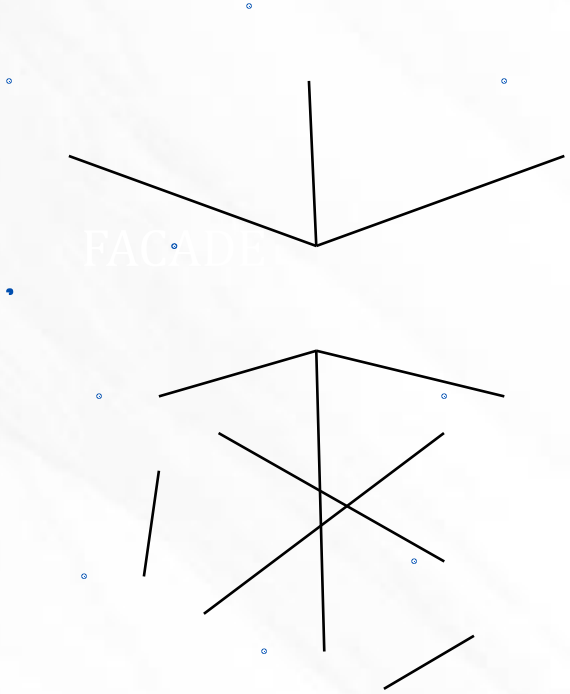
Disadvantages

- **Identity Comparison**
  - We can't do identity comparison since we don't know exact real subject, just a surrogate.
- **Inambiguity**
  - Client may not be aware that the real subject it is accessing now is not same as the previous one.
  - Because client doesn't know what else proxy is doing, other than calling real subject.

Façade

Motivation

Client classes



Subsystem classes

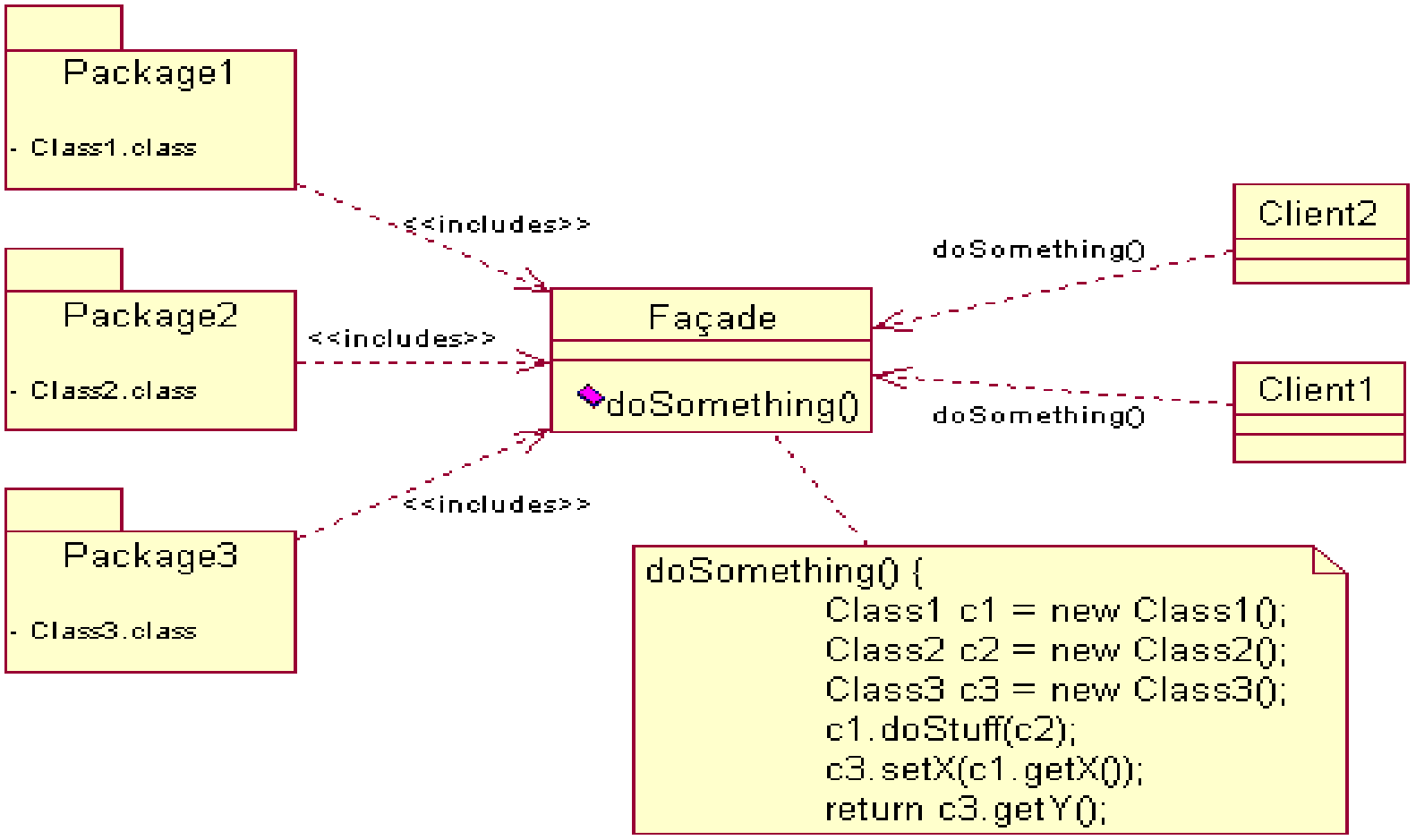
Applicability

- Provide a simple interface to a subsystem.
- There are many dependencies between clients and the implementation classes of an abstraction.
- Layer your subsystems.

Participants

- **Facade**
  - Delegate requests to appropriate subsystem objects.
- **Subsystem classes**
  - Implement subsystem functionality.
  - Handle work assigned by the facade.

Façade


Package1
- Class1.class
Package2
- Class2.class
Package3
- Class3.class
<<includes>>
<<includes>>
<<includes>>
Façade
doSomething()
Client2
doSomething()
Client1
doSomething()
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();

Consequences

- Shields clients from subsystem components.
- Promotes weak coupling between the subsystems and its clients.
- It’s does not prevent applications from using subsystem classes if they need to.

Behavioral pattern

- **Chain of Responsibility**: riduce l'accoppiamento fra l'oggetto che effettua una richiesta e quello che la soddisfa, dando a più oggetti la possibilità di soddisfarla
- **Command**: isola il codice che effettua un'azione da quello che la richiede
- **Interpreter**: specifica classi per l'interpretazione di frasi in un linguaggio
- **Iterator**: disaccoppia gli algoritmi per la navigazione in contenitori di oggetti dalla natura del contenitore
- **Mediator**: definisce un oggetto che descrive come due set di oggetti interagiscono, interponendosi fra loro
- **Memento** : permette di riportare un oggetto ad uno stato precedente

Behavioral pattern

- **Observer**: definisce una dipendenza uno a molti fra oggetti diversi, gestendo cambiamenti di stato e relative notifiche
- **State**: permette ad un oggetto di cambiare il suo comportamento in base al proprio stato interno
- **Strategy**: permette di modificare dinamicamente gli algoritmi utilizzati da un'applicazione
- **Template method**: permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi
- **Visitor**: separa un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa.

Chain of responsibility

Intent

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

Concetto: invio richiesta a qualcuno che è in anello di una catena. Esso può scegliere di gestire la richiesta o mandarla a qualcuno di più responsabile - Es. Khow o gestione eccez.

Applicability

Use Chain of Responsibility when:

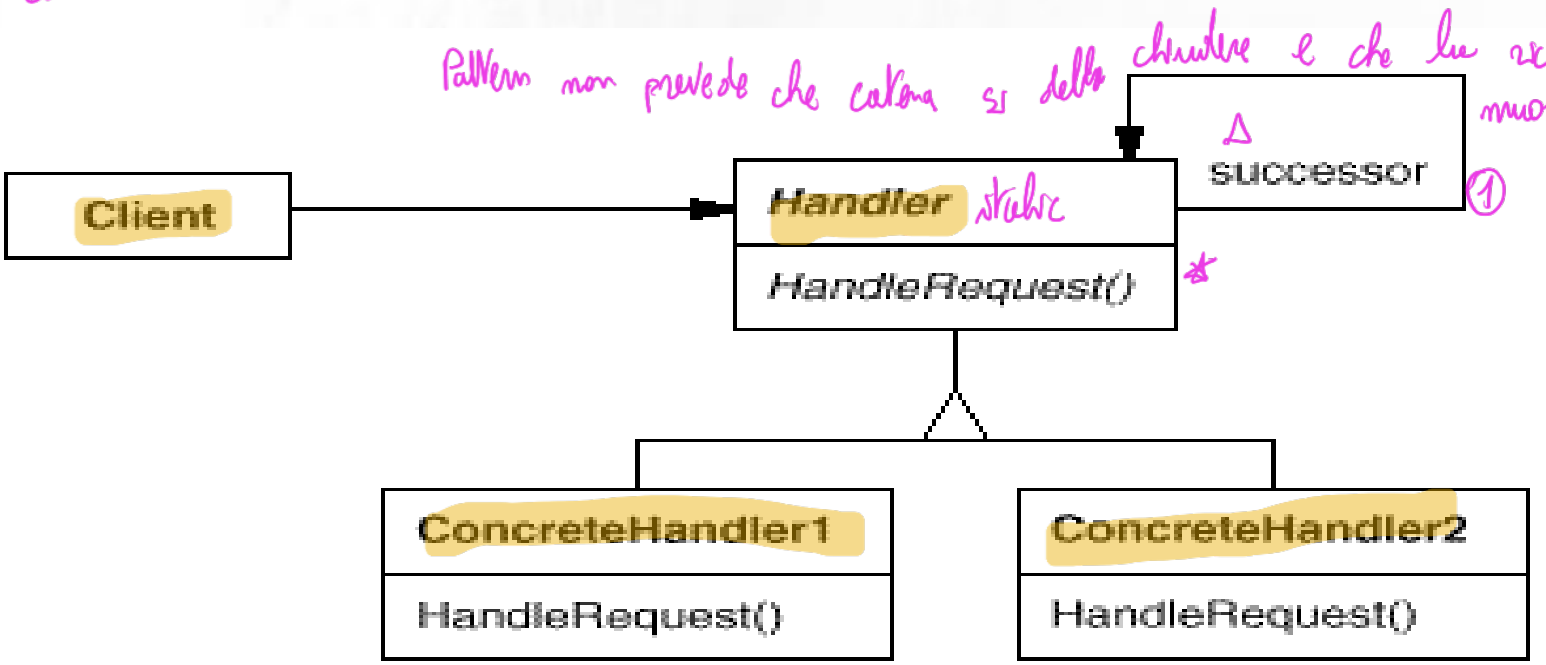
• More than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.

Non so chi possa rispondere

• You want to issue a request to one of several objects without specifying the receiver explicitly.

• The set of objects that can handle a request should be specified dynamically.

Structure



A typical object structure might look like this:

Es: Gestione Documenti Studenti. Passo all'interfaccia Immatricolazione. Segreteria Dipartimento può gestire questa richiesta? Ok, lo posso fare.
Se non avesse potuto, avrebbe passato la mano all'handler successivo.

△ Ogni handler è collegato alla sua successore.

NOTA: qui non c'è una richiesta di risposta di feedback finale. Non sono obbligato a comunicare col client

Participants

Handler

- defines an interface for handling requests.
- (optional) implements the successor link.

ConcreteHandler

- handles requests it is responsible for.
- can access its successor.
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

Client

- initiates the request to a ConcreteHandler object on the chain.

Consequences

1. ***Reduced coupling:*** An object only has to know that a request will be handled "appropriately."

2. ***Added flexibility in assigning responsibilities to objects.*** You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time.

3. ***Receipt isn't guaranteed.*** *Since a request has no explicit receiver, there's no guarantee it'll be* handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

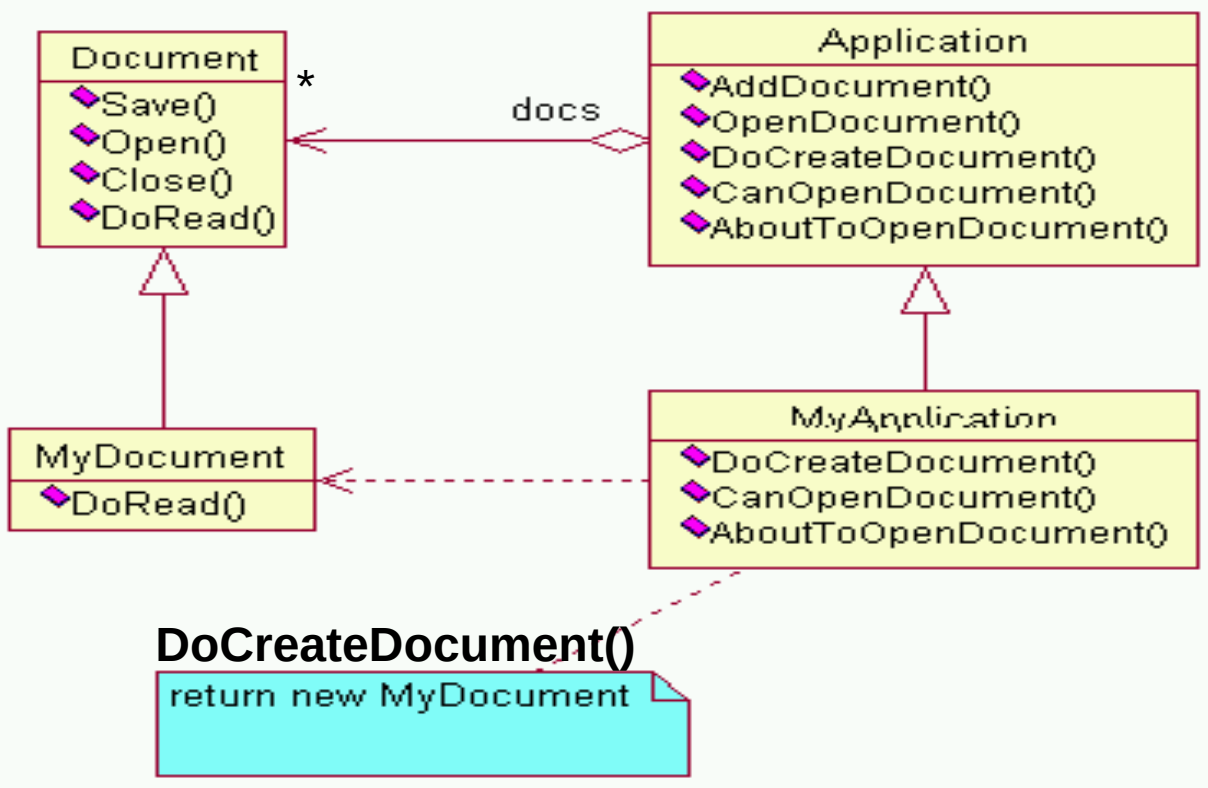
Template Method

Intent

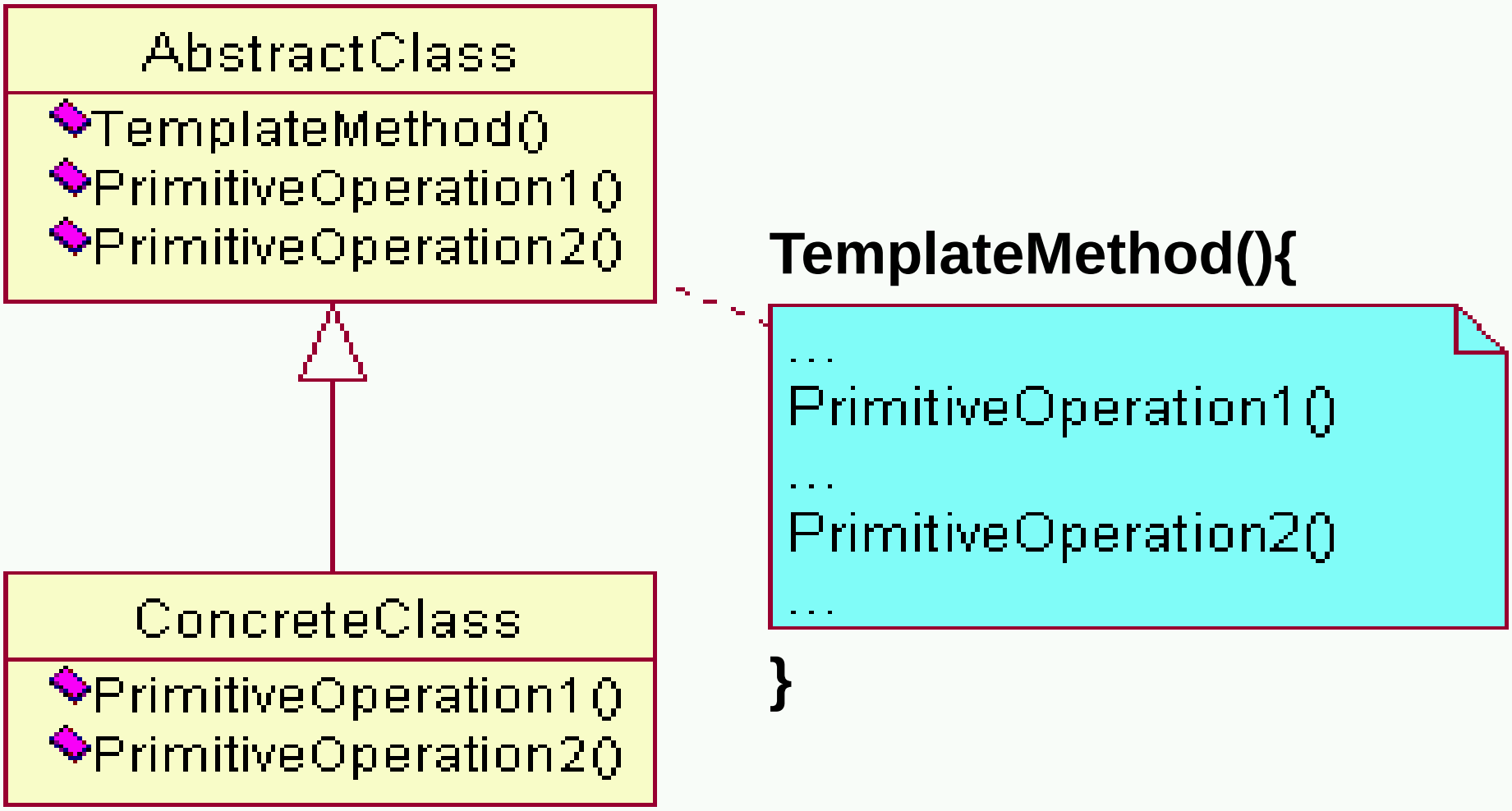
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

Problem:

Given a template method if we need to customize or redefine a particular set in the algorithm one may have to override the template function this causes duplication and an inefficient usage of template methods.



Structure


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()
TemplateMethod(){
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
}

Applicability

To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.

When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

To control subclasses extensions. You can define a template method that calls “hook” operations at specific points, thereby permitting extensions only at those points.

Interpreter

Intent

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

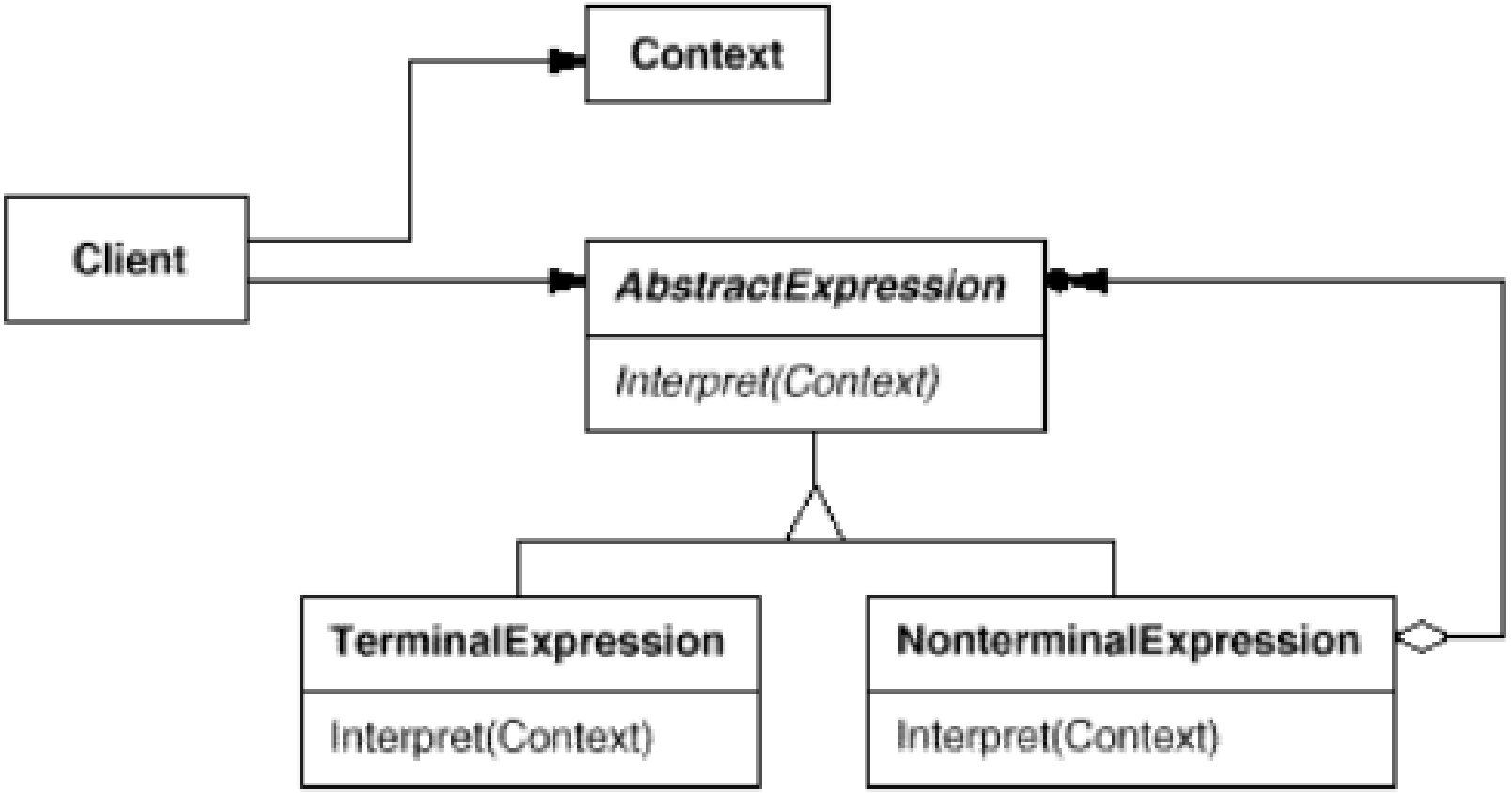
Applicability

Use the Interpreter pattern when:

- There is a language to interpret, and you can represent statements in the language as abstract syntax trees.

- Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

Structure



Participants

AbstractExpression

- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression

- implements an Interpret operation associated with terminal symbols in the grammar.
- an instance is required for every terminal symbol in a sentence.

NonterminalExpression

- one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
- maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
- implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the

variables representing R1 through Rn.

Context

- contains information that's global to the interpreter.

Consequences

1. It's easy to change and extend the grammar

2. Implementing the grammar is easy, too

3. Complex grammars are hard to maintain

4. Adding new ways to interpret expressions

Memento

- **Intent**

Without violating encapsulation, capture and externalize an object'sinternal state so that the object can be restored to this state later.

- **Also Known As**

Token

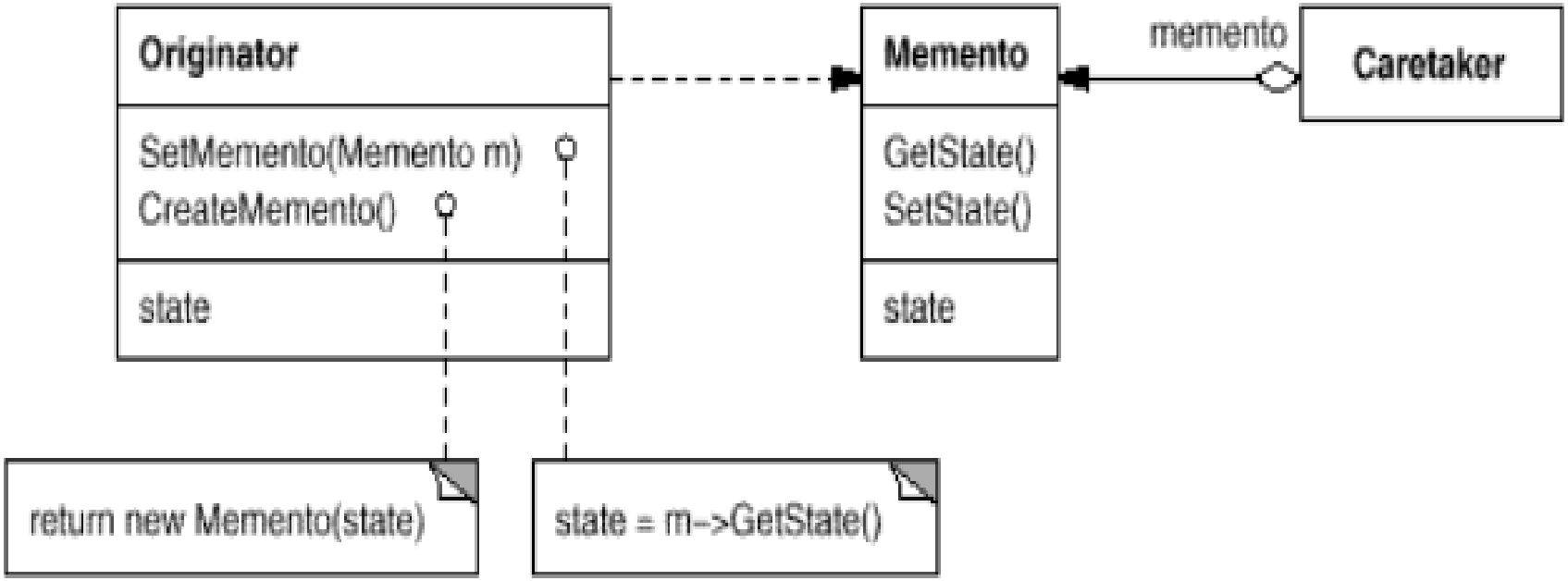
Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later

- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure


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

Participants

Memento

- Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
- Protects against access by objects other than the originator.

Originator

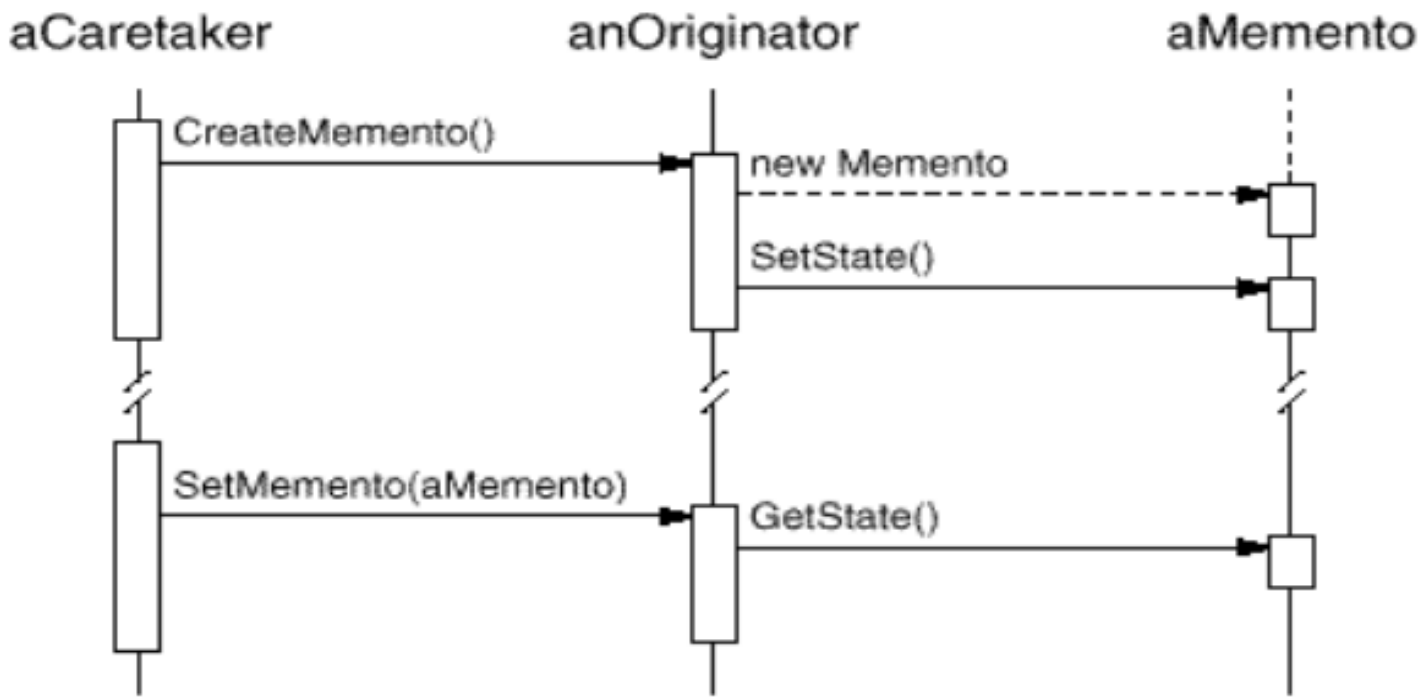
- creates a memento containing a snapshot of its current internal state.
- uses the memento to restore its internal state.

Caretaker

- is responsible for the memento's safekeeping.
- never operates on or examines the contents of a memento.

Collaborations

A caretaker requests a memento from an originator, holds it for atime, and passes it back to the originator, as the followinginteraction diagram illustrates:



Consequences

1. ***Preserving encapsulation boundaries***
2. **It simplifies Originator**
3. **Using mementos might be expensive**
4. **Defining narrow and wide interfaces**
5. **Hidden costs in caring for mementos**

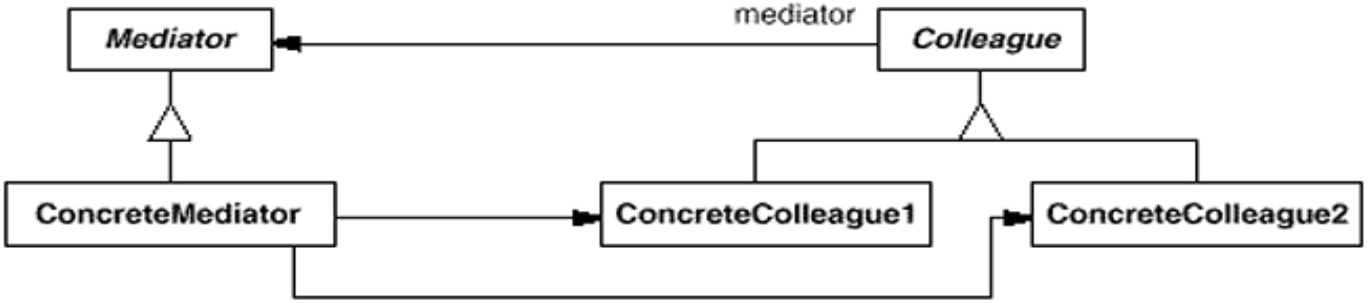
Mediator

- **Intent**
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

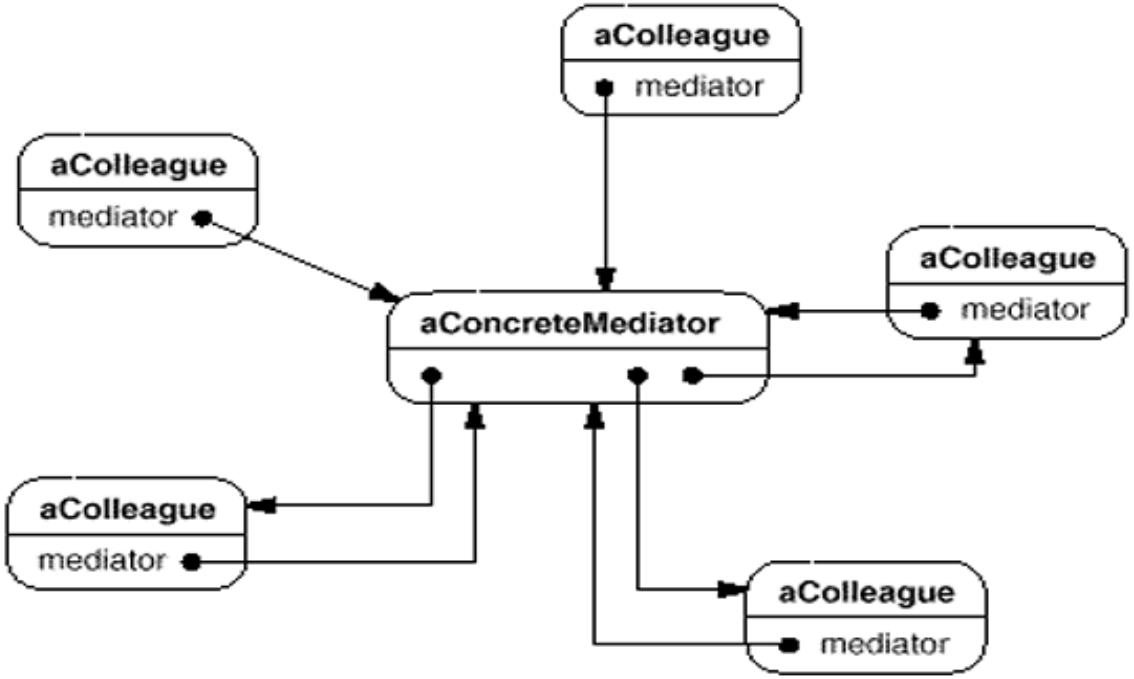
Applicability

- Use the Mediator pattern when
  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - reusing an object is difficult because it refers to and communicates with many other objects.
  - a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure



A typical object structure might look like this:



Participants

- **Mediator**
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator**
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.
- **Colleague classes**
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Collaborations

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences

- ***It limits subclassing****.* A mediator localizes behavior that otherwise would be distributed among several objects.
- ***It decouples colleagues****.* A mediator promotes loose coupling between colleagues.
- ***It simplifies object protocols****.* A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues.
- ***It abstracts how objects cooperate****.* Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior.
- ***It centralizes control.*** The Mediator pattern trades complexity of interaction for complexity in the mediator.

Observer

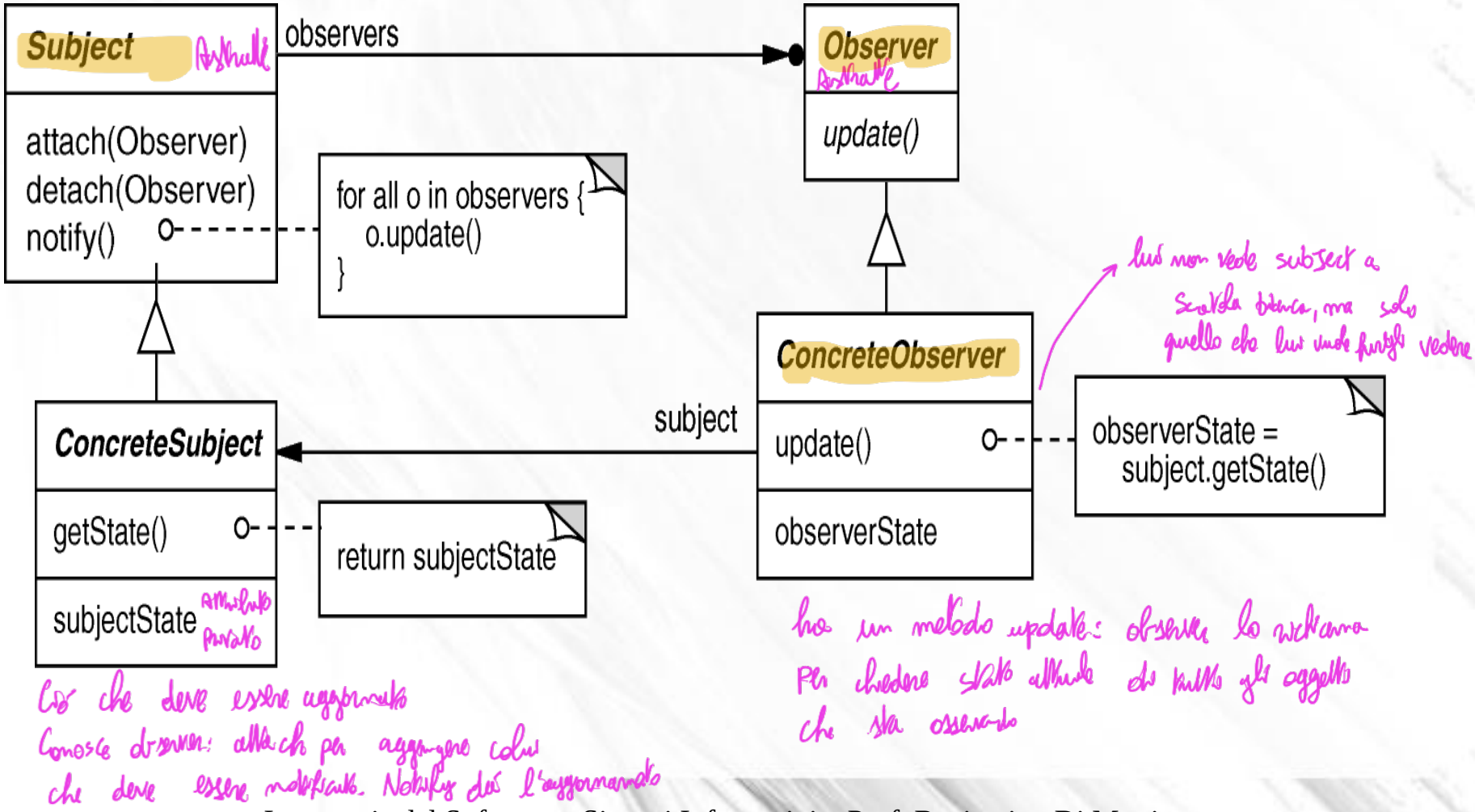
Intent

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

Applicability

- An abstraction has two aspects, one dependent on the other.
- When changing one object requires changing others, and you don’t know how many objects need changed.
- When an object needs to notify others without knowledge about who they are.

Observer pattern: struttura



Observer

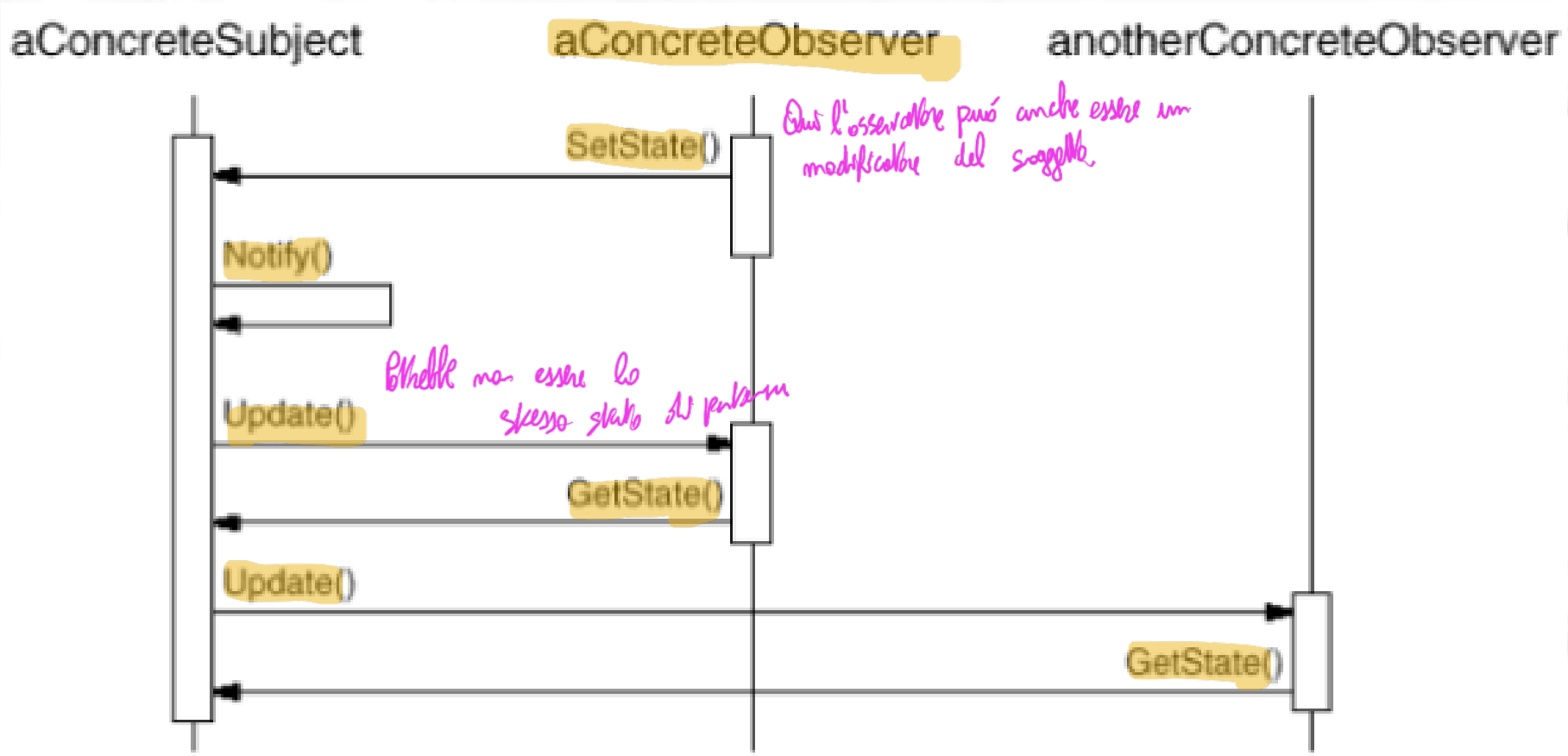
Participants

- **Subject**
  - Knows its observers, but not their “real” identity.
  - Provides an interface for attaching/detaching observers.
- **Observer**
  - Defines an updating interface for objects that should be identified of changes.
- **ConcreteSubject**
  - Stores state of interest to ConcreteObserver objects.
  - Sends update notice to observers upon state change.
- **ConcreteObserver**
  - Maintains reference to ConcreteSubject (sometimes).
  - Maintains state that must be consistent with ConcreteSubject.
  - Implements the Observer interface.

Collaborations

- ConcreteSubject notifies observers when changes occur.
- ConcreteObserver may query subject regarding state change.

Observer pattern: comportamento

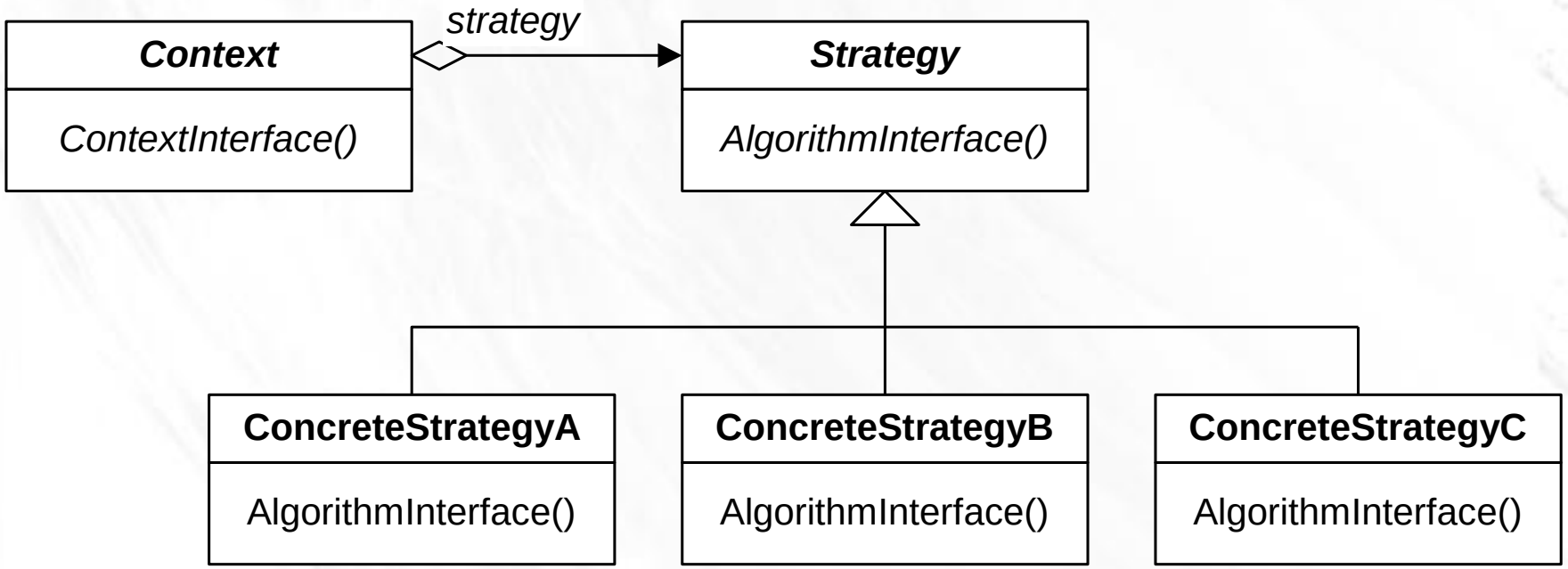


Strategy Pattern

- **Intent:** defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Motivation:** when there are many algorithms for solving a problem, hard-wiring all algorithms in client's code may have several problems:
  - Clients get fat and harder to maintain
  - Different algorithms may be appropriate at different time
  - It is difficult to add new algorithms

Strategy Pattern



Participants

- **Strategy:** declares an interface common to all supported algorithm. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy:** implements the algorithm using the Strategy interface
- **Context:** maintains a reference to a Strategy object and defines an interface that let Strategy access its data

Command

Intent

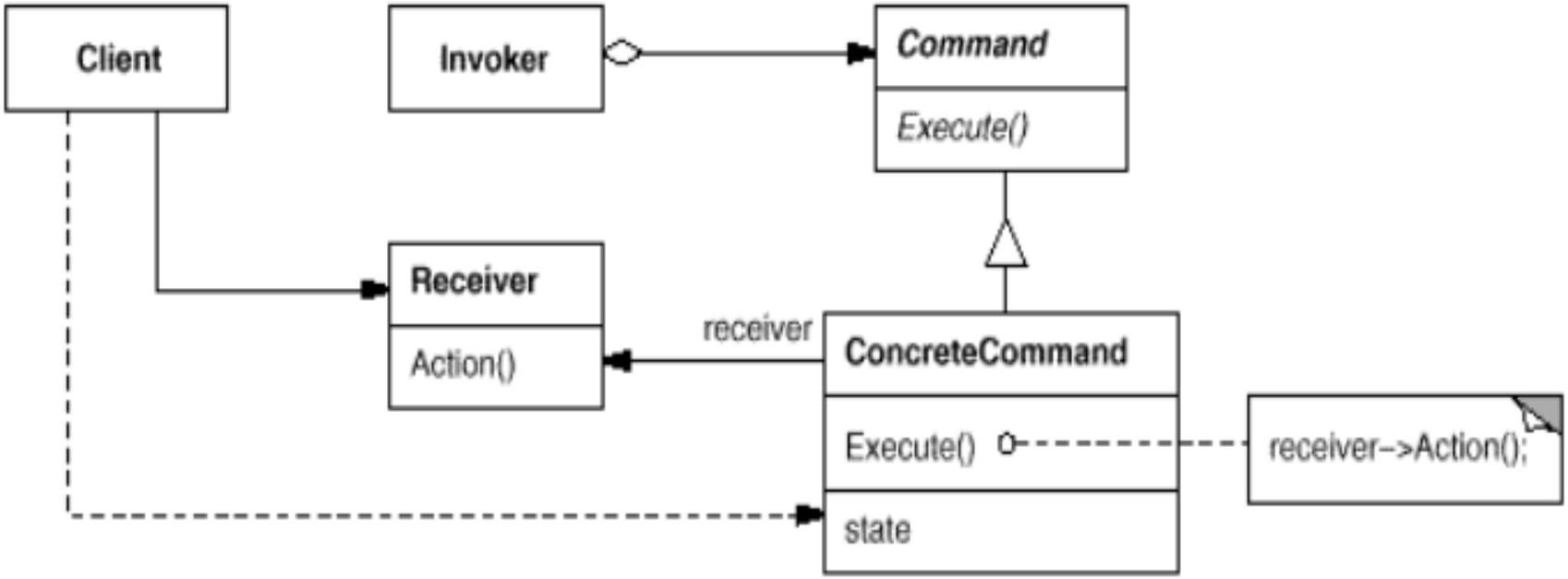
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Applicability

- Parameterize objects by an action
- In place of “callbacks”
- Specify, queue, and execute requests at different times
- Supports undo when Command maintains state information necessary for reversing command.
- Added support for logging Command behavior.
- Support high-level operations built on primitive operations (transactions).

Behavioral Patterns - Command

Structure



Command

Participants

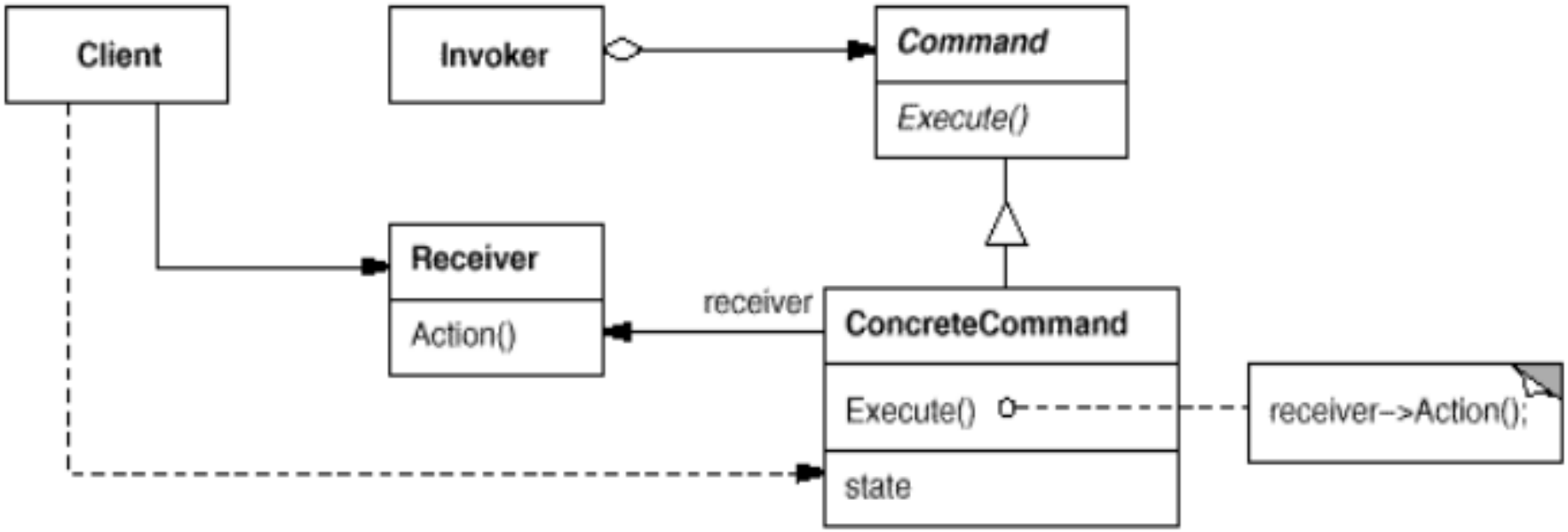
- **Command**: Declares an interface for executing an operation.
- **ConcreteCommand**
  - Defines a binding between a Receiver object and an action.
  - Implements execute() by invoking a corresponding operation on Receiver.

- **Client** (Application): Creates a Command object and sets its Receiver.
- **Invoker**: Asks the Command to carry out a request.
- **Receiver**: Knows how to perform the operation associated with a request. Can be any class.

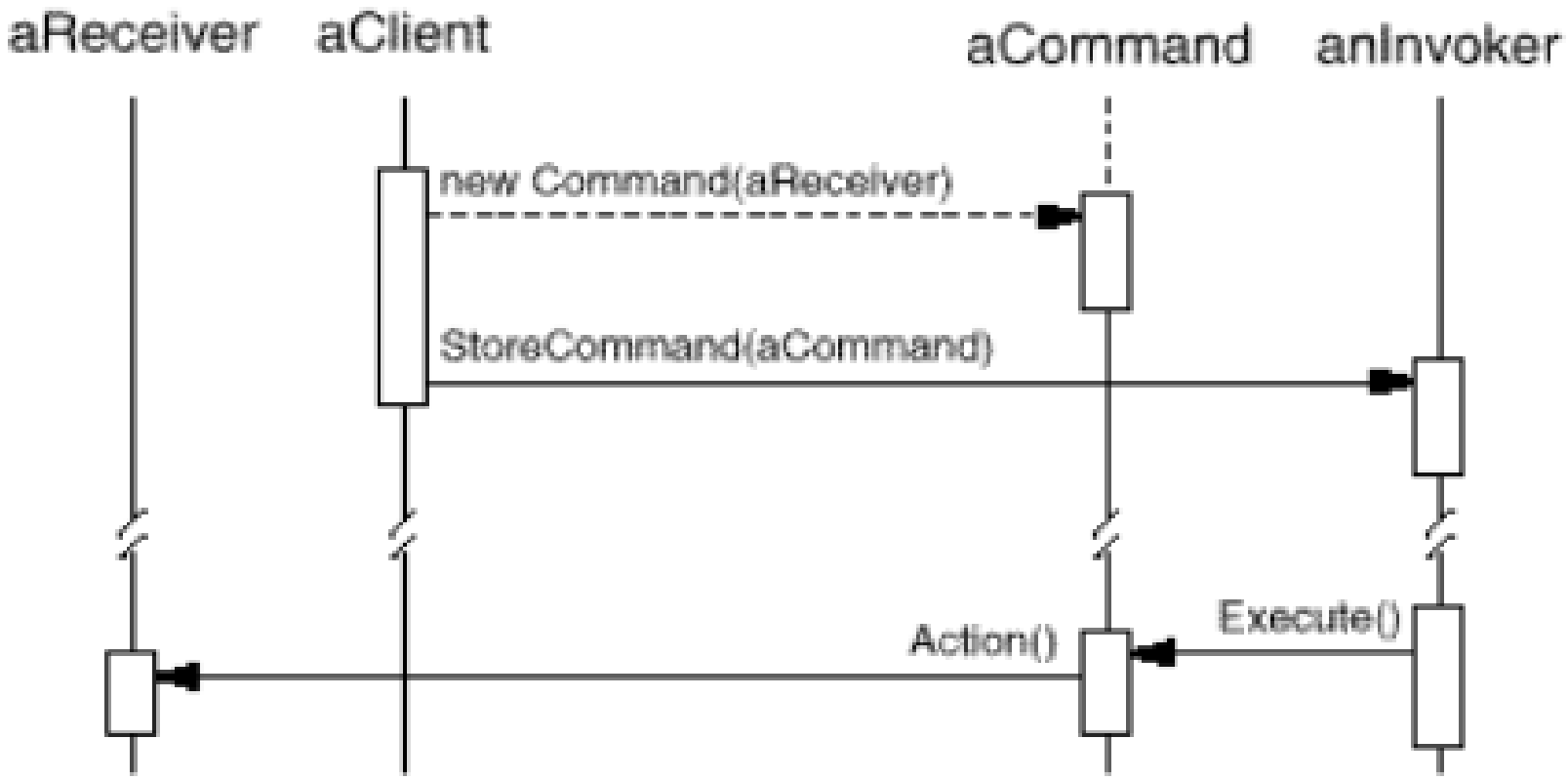
Collaborations

- Creates a ConcreteCommand object and sets its Receiver.
- An Invoker stores the ConcreteCommand.
- Invoker calls *execute()* on command.
- ConcreteCommand invokes operation on its receiver.

Command: Struttura



Command: Comportamento



State

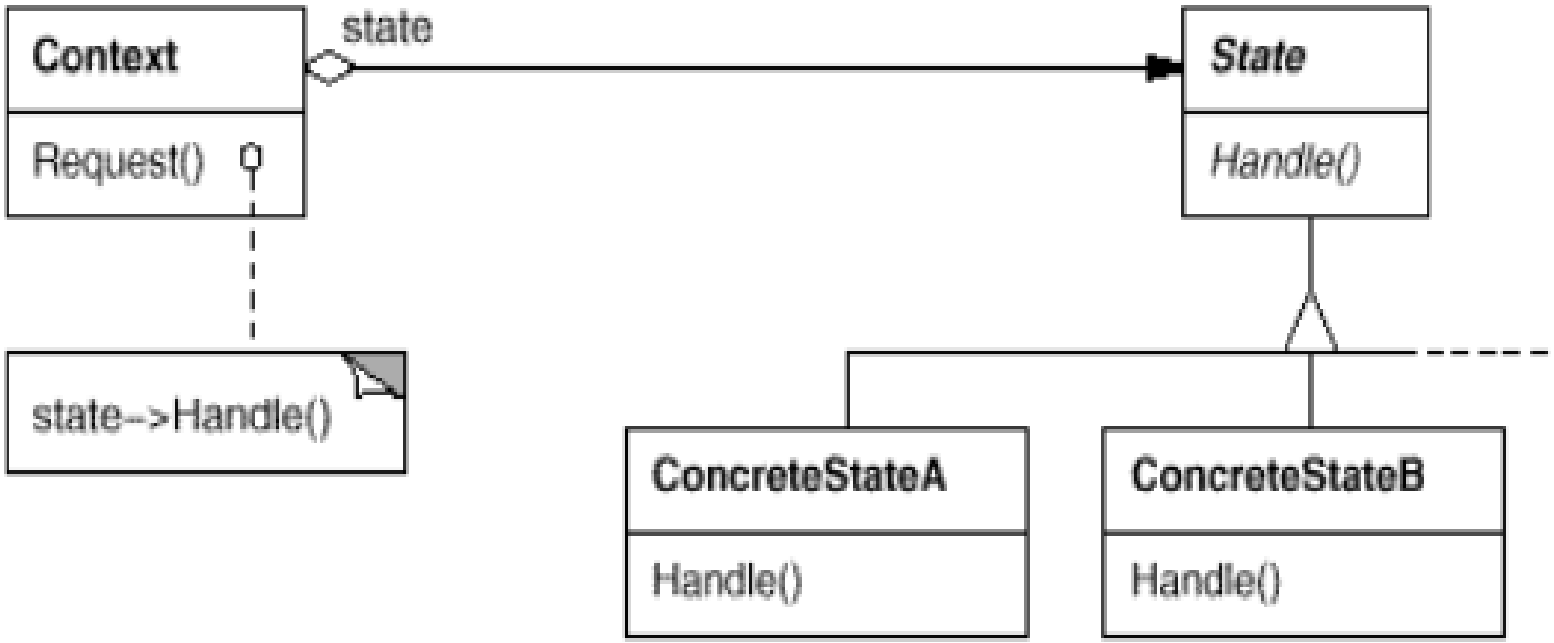
Intent

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

Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on its state.
- Operations have large, multipart conditional statements that depend on the object's state.
  - Usually represented by constants.
  - Sometimes, the same conditional structure is repeated.

State


Context
state
State
Request()
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

State

Participants

- **Context**
  - Defines interface of interest to clients.
  - Maintains an association with a subclass of State, that defines the current state.
- **State**
  - Defines an interface for encapsulating the behavior with respect to state.
- **ConcreteState**
  - Each subclass implements a behavior associated with a particular state of the Context.

State

Collaborations

- Context delegates state-specific behavior to the current concrete State object.
- The state object may need access to Context information; so the context is usually passed as a parameter.
- Clients do not deal with State object directly.
- Either Context or a concrete State subclass can decide which state succeeds another.

Visitor

Oggetto riceve visitor che fa operazioni (su sé stesso) e poi lo passo all'elemento successivo

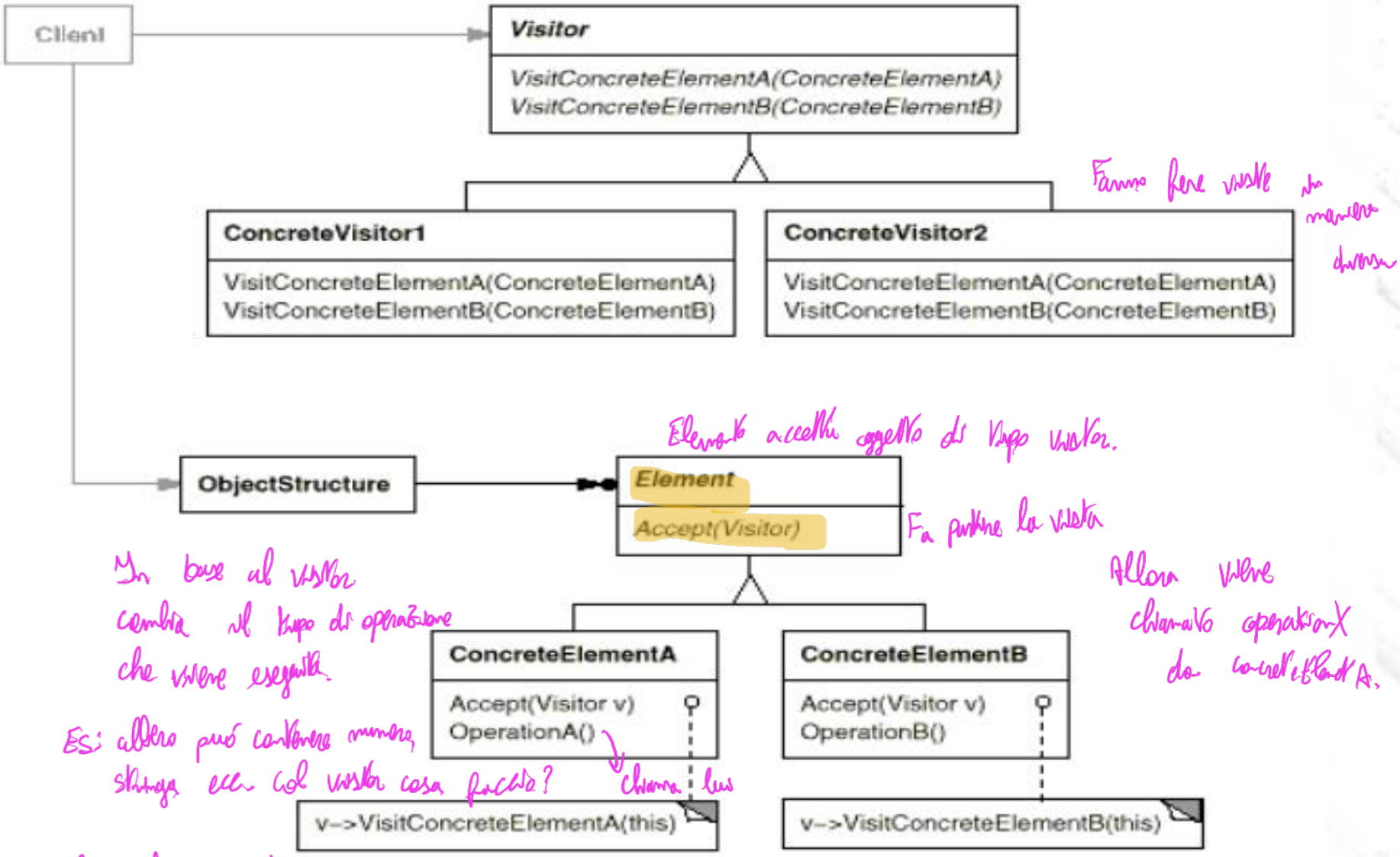
Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability

- An object structure contains many disparate classes, and operations need to be performed based on concrete classes.
- Many distinct operations need to be performed on an object structure.
- An object structure rarely changes, but new operations need to be defined over the structure.

Visitor: Struttura



Visitor

Participants

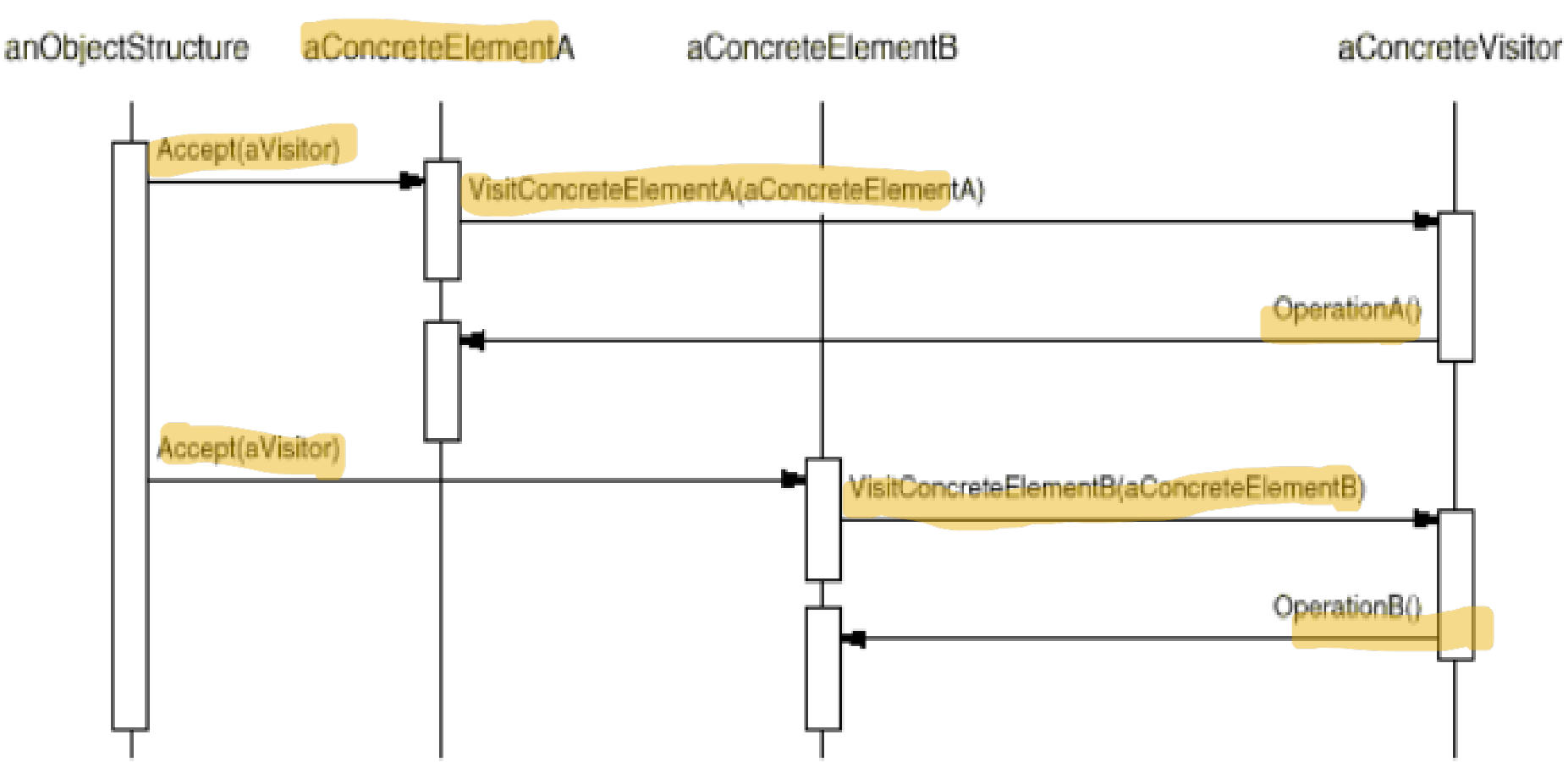
- **Visitor** — declares a visit operation for each class within the object structure aggregation.
- **ConcreteVisitor** — implements each operation declared by Visitor. Provides algorithm context.
- **Element** — defines an accept operation taking a Visitor as an argument.
- **ConcreteElementX** — implements an accept operation taking a Visitor as an argument.
- **ObjectStructure**
  - Enumerates its elements; potentially disparate classes.
  - May provide a high level interface for visitor to visit its elements.
  - Potentially a composite or just a general collection

Visitor

Collaborations

- A client creates an instance of a concrete Visitor subclass.
- Client requests the ObjectStructure to allow the visitor to visit each.
- When visited, Element invokes the appropriate operation on Visitor; overloading to know the element type.

Visitor: Comportamento



Concurrency pattern

- **Active Object**: disaccoppia l'esecuzione dei metodi dalla loro esecuzione per oggetti che risiedono in thread diversi.
- **Balking**: un oggetto può eseguire un'azione su di un altro solo se questo si trova in un certo stato.
- **Double checked locking**: riduce l'overhead dovuto all'acquisizione di un lock, testando prima un criterio ("lock hint") senza acquisire realmente il lock.
- **Guarded suspension**: gestione di operazioni che richiedono un lock e una precondizione.
- **Monitor Object**: definizione di un oggetto a cui i thread possono accedere solo in modo mutualmente esclusivo

Concurrency pattern

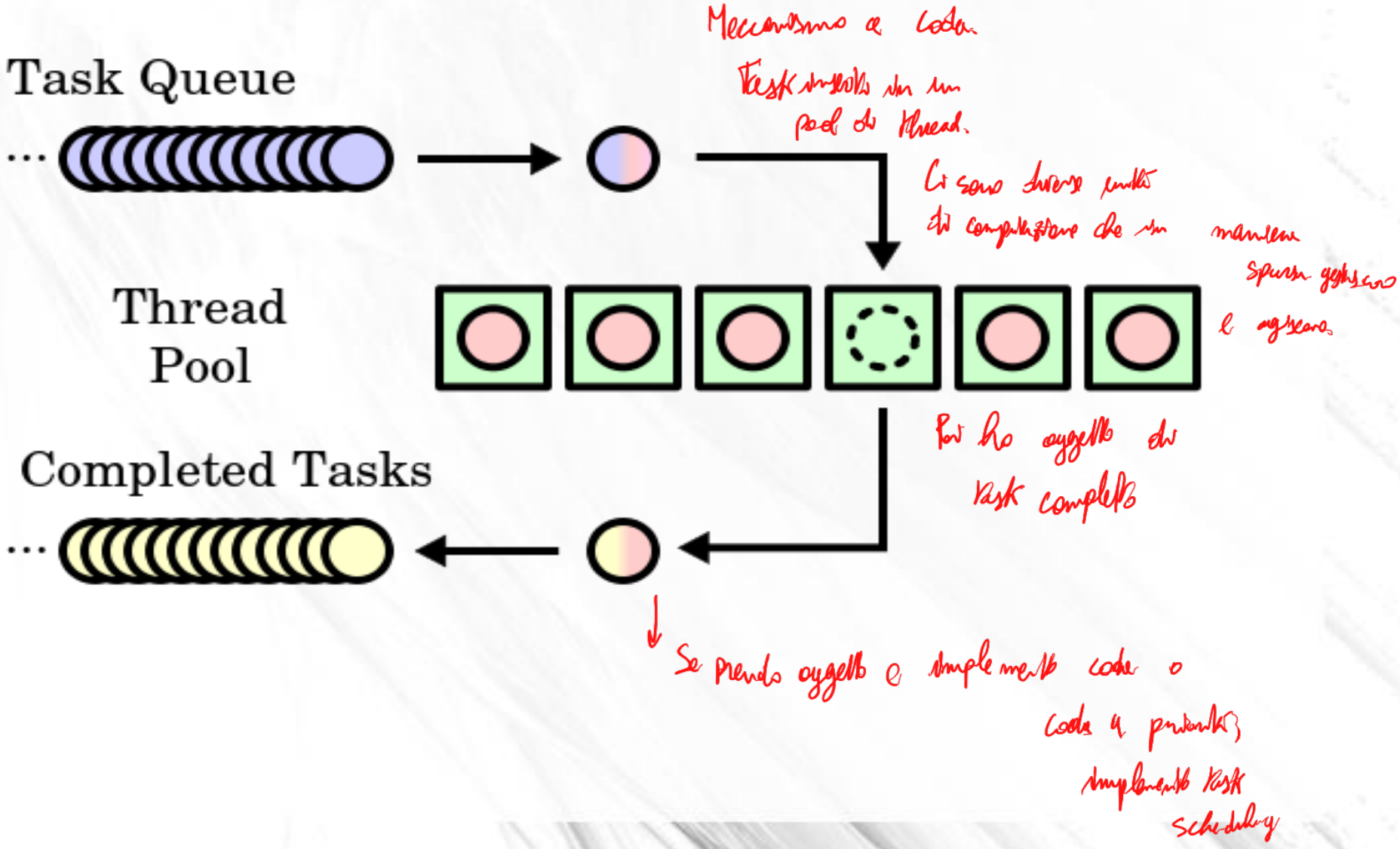
- **Reactor**: definisce un gestore di eventi che raccoglie richieste concorrenti, le indirizza agli handler corretti e inoltra le risposte ai richiedenti.

- **Read write lock**: fornisce un meccanismo che permette accessi concorrenti in lettura da parte di thread multipli, ma impedendo l'accesso in scrittura a più di un thred alla volta.

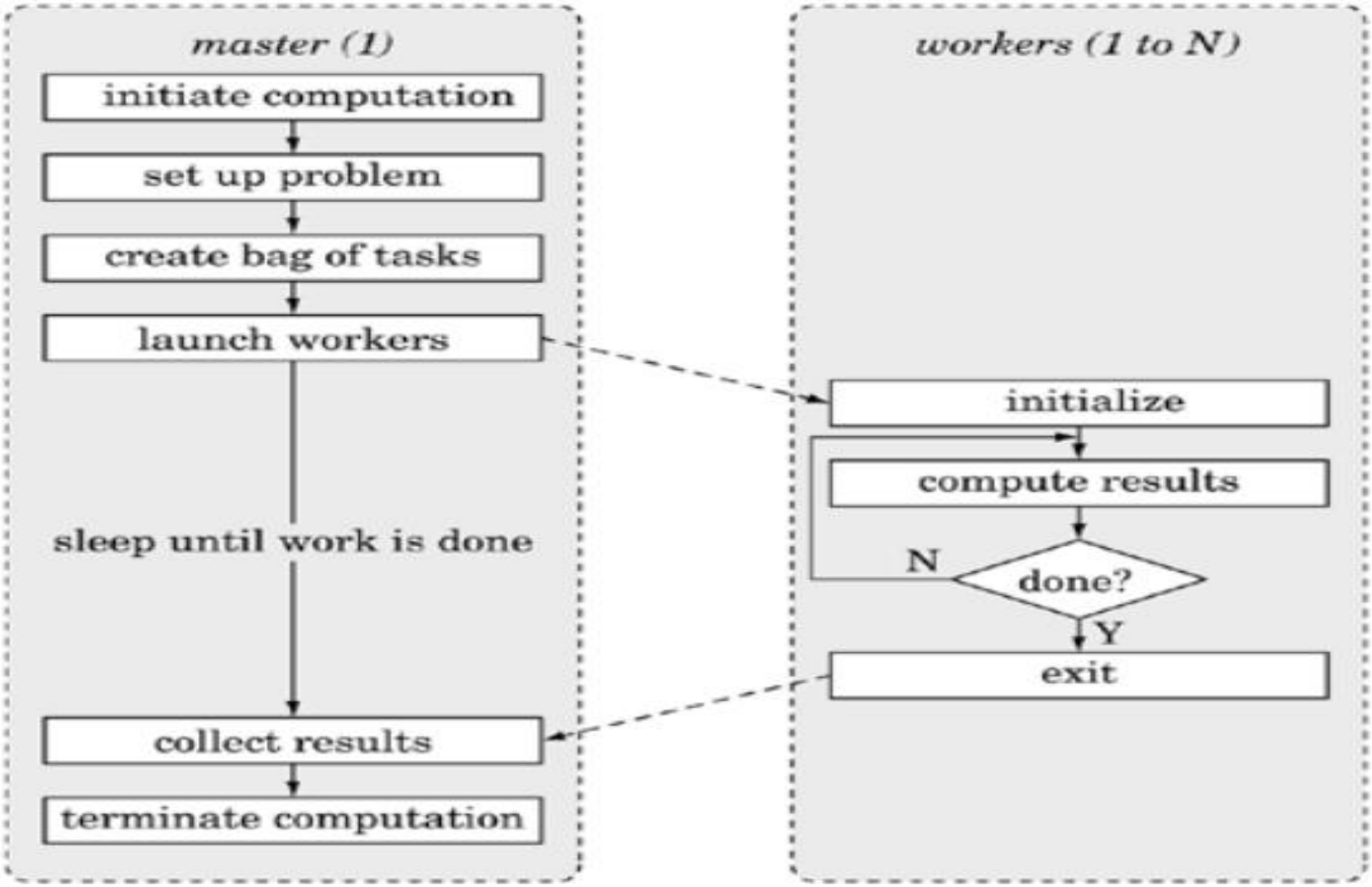
- **Scheduler:** usato per gestire l'esecuzione di più thread, definisce un oggetto in grado di implementare differenti politiche di scheduling, rimanendo indipendente da esse.

- **Thread pool**: Definisce un certo numero di thread per la gestione di task, generalmente organizzati in una coda.

Thread pool



Master/Worker



Divide et Impera