LEZIONE 30
DESIGN PATTERNS 2:
altri GoF

Ingegneria del Software e Progettazione Web

Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

catalogo GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

pattern strutturali

- descrivono le modalità di composizione di classi e oggetti per formare strutture complesse
- basati su classi
  - utilizzano l'ereditarietà per comporre interfacce o implementazioni
  - esempio: ereditarietà multipla combina due o più classi per ottenere una classe con tutte le proprietà delle super-classi
- basati su oggetti
  - modellano le modalità di composizione di oggetti per realizzare nuove funzionalità
  - hanno maggiore flessibilità poiché è possibile cambiare la composizione durante l'esecuzione

adapter – 1

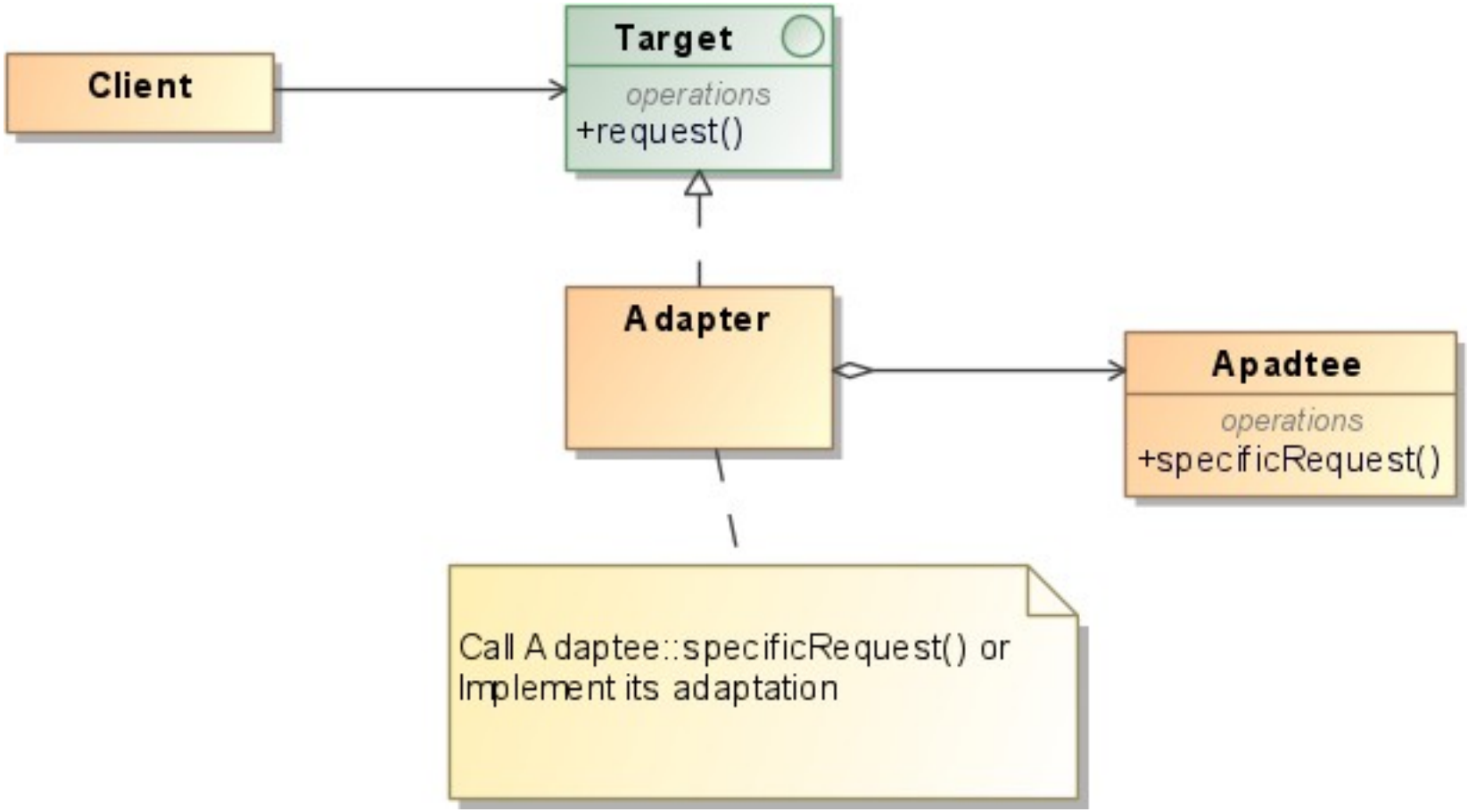
- Scopo
  - gestire interfacce incompatibili
  - fornire un’interfaccia stabile a classi funzionalmente simili o con interfacce diverse
- Sinonimi
  - Wrapper
- Motivazione
  - spesso le classi di un sistema vengono strutturate/progettate cercando di offrire un profilo riusabile. Tuttavia, può capitare che queste classi non possono essere effettivamente riusate; per esempio perchè la loro interfaccia offerta non rispecchia esattamente i requisiti (o la segnatura) richiesti di uno specifico dominio

adapter – 2

- Applicabilità
  - si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve
  - si vuole realizzare una classe riusabile che coopera con altre classi anche se scorrelate o impreviste e con una interfaccia eventualmente incompatibile

adapter – 3

- Struttura



adapter – 4

- Partecipanti
  - Target
    - definisce l'interfaccia di dominio con il client
  - Client
    - coopera con oggetti conformi all'interfaccia Target
  - Adaptee
    - definisce l'interfaccia esistente da adattare
  - Adapter
    - realizza il meccanismo di adattamento

adapter – 5

- Collaborazioni
  - i Clients invocano operazioni su una istanza di Adapter. A sua volta l'istanza di Adapter gestisce opportunamente l'invocazione verso le istanze di Adaptee
- Consequenze
  - adatta Adaptee a Target attraverso la classe concreta Adapter; la classe Adapter non è indicata se si volesse gestire l'adattamento di una classe ma anche tutte le sue sotto-classi
  - quanto “lavoro” deve fare la classe Adapter? dipende da quanto sono simili le interfacce Target ed Adaptee.
  - l'uso del pattern non è sempre trasparente ai Client

adapter – 6

- Implementazione
  - Target (`Jewel`)
    - interface o classe astratta
  - Client (`Jeweller`)
    - potrebbe risentire dei cambiamenti su Target
  - Adaptee (`Stone`)
    - classe da adattare
  - Adapter (`Adapter`)
    - specializzazione o raffinamento di Target e riferimento ad istanza di Adaptee

adapter – 7

- Codice di esempio :

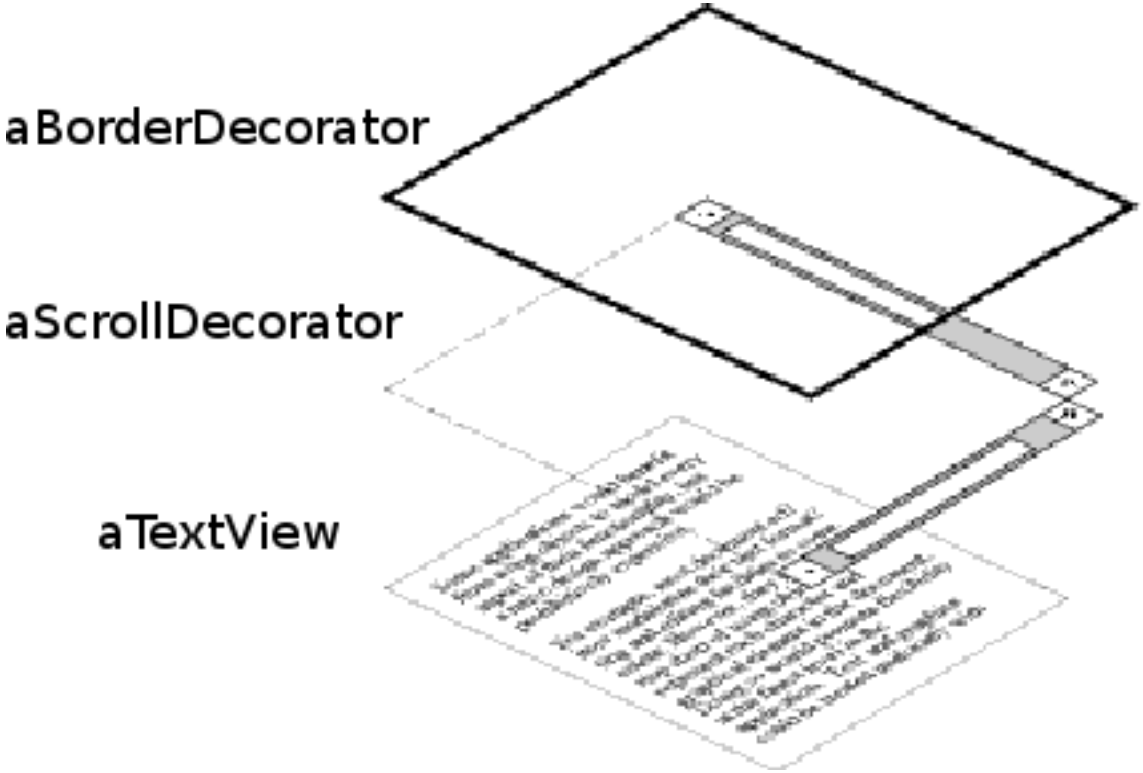
VEDERE MATERIALE ALLEGATO ALLA LEZIONE

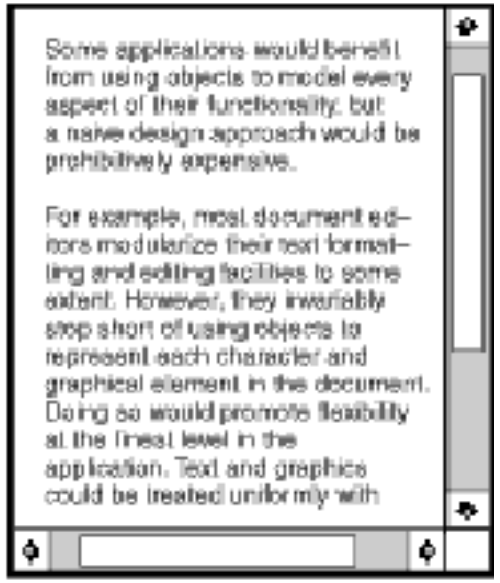
decorator - 1

- Scopo
  - aggiungere dinamicamente responsabilità ad un oggetto. I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità
- Sinonimo
  - Wrapper
- Motivazione
  - GUI toolkit dovrebbe consentire di aggiungere proprietà come bordi, scorrimento, ai singoli elementi grafici
  - il modo per aggiungere responsabilità è tramite ereditarietà
    - estendere una classe e aggiungere il *bordo*

decorator – 2

- Motivazione (2)





decorator – 3

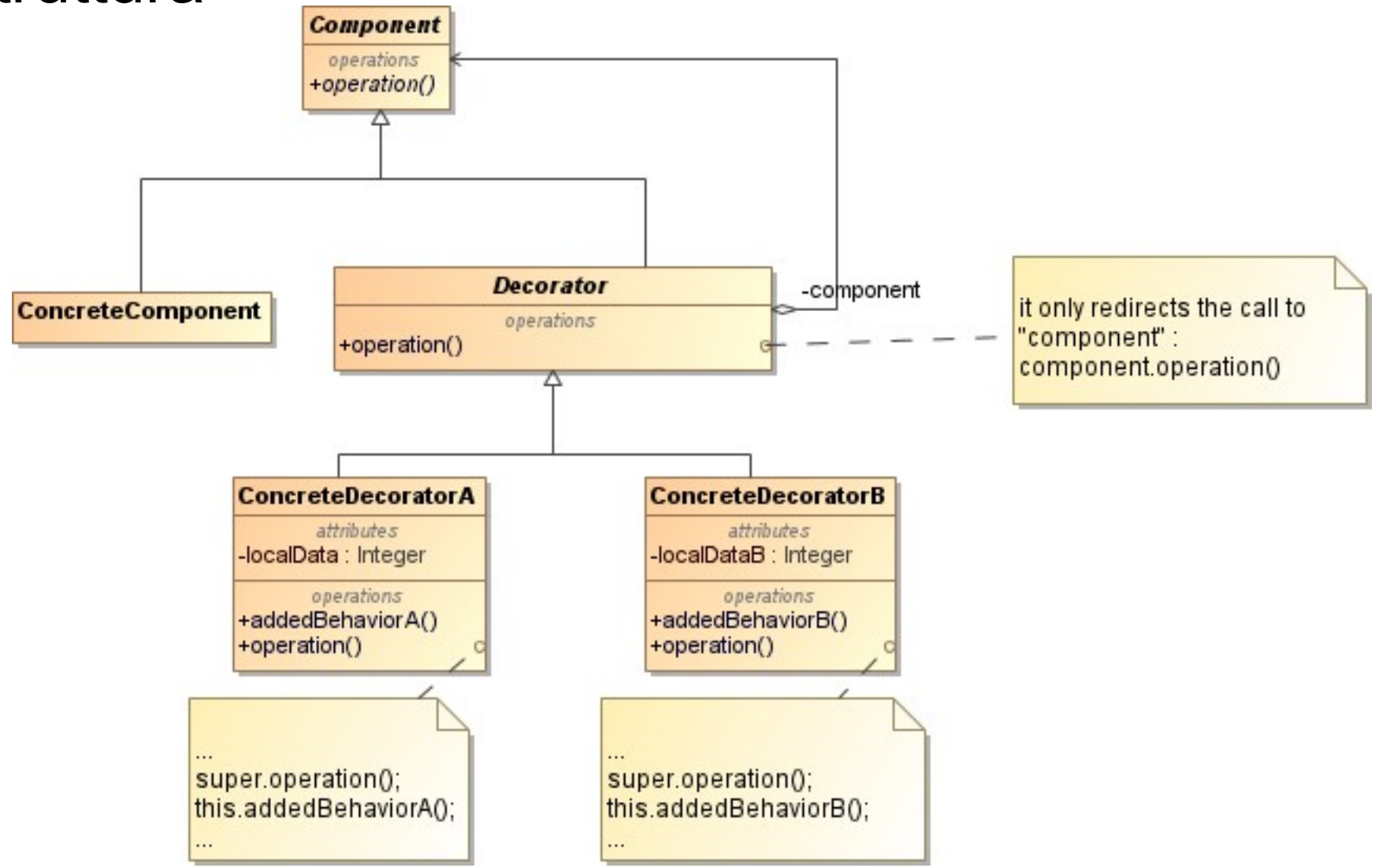
- Motivazione (3)
  - definire un approccio flessibile che consenta di racchiudere il componente da “decorare” in un altro che abbia la sola responsabilità di aggiungere il bordo/scrollbar/etc
    - oggetto contenitore detto decorator
    - decorator ha un’interfaccia conforme all’oggetto decorato in modo tale da essere trasparente ai vari client
    - decorator trasferisce le richieste al componente decorato effettuando azioni aggiuntive (esempio decorando il bordo) prima o dopo il trasferimento della richiesta
    - essendo trasparente ai client è possibile annidare i decorator consentendo l’aggiunta di un numero illimitato di responsabilità agli oggetti decorati

decorator – 4

- Applicabilità
  - si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente ed in modo trasparente, senza coinvolgere altri oggetti
  - si vuole poter togliere responsabilità agli oggetti
  - l'estensione diretta attraverso le definizione di sottoclassi non è praticabile
    - esplosione di sottoclassi per supportare ogni possibile combinazione

decorator – 5

- Struttura



decorator – 6

- Partecipanti
  - Component (`VisualComponent`)
    - definisce l'interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente
  - ConcreteComponent (`TextView`)
    - definisce un oggetto al quale possono essere aggiunte responsabilità ulteriori
  - Decorator
    - mantiene un riferimento ad un oggetto `Component` e definisce un'interfaccia conforme all'interfaccia di `Component`
  - ConcreteDecorator (`BorderDecorator`, `ScrollDecorator`)
    - aggiunge responsabilità al componente

decorator – 7

- Collaborazioni
  - un Decorator trasferisce le richieste al suo oggetto Component. Può svolgere opzionalmente operazioni ulteriori prima e dopo il trasferimento della richiesta
- Conseguenze
  - maggiore flessibilità rispetto all'utilizzo dell'ereditarietà (multipla) statica
    - responsabilità possono essere aggiunte e rimosse in esecuzione semplicemente collegando e scollegando i decoratori agli oggetti decorati
    - in caso di ereditarietà è necessario creare una nuova classe per ogni responsabilità (es. `BorderedScrollableTextView`, `BorderedTextView`)
  - consente di evitare di definire classi troppo complesse nella gerarchia

decorator – 8

- Codice di esempio :

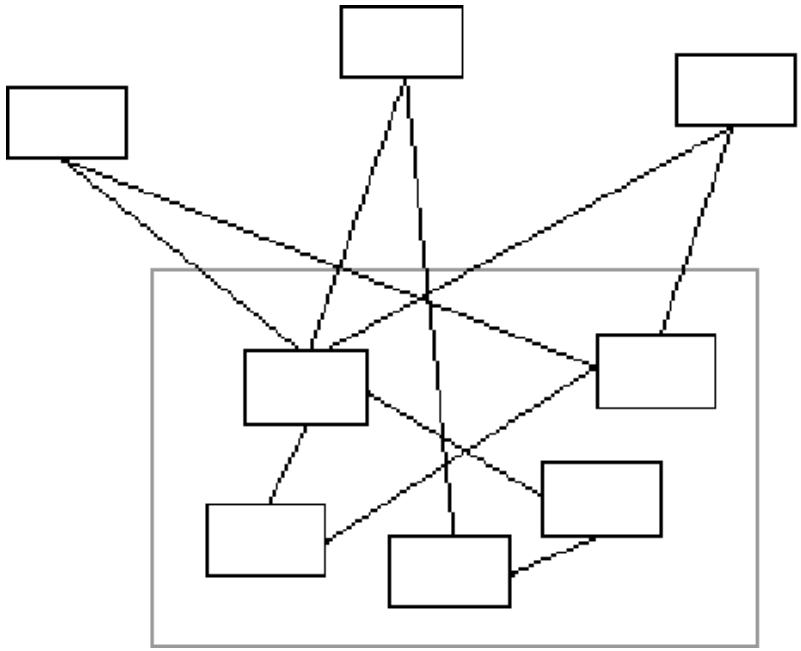
VEDERE MATERIALE ALLEGATO ALLA LEZIONE

façade – 1

- Scopo
  - fornire una interfaccia unificata ad un insieme di interfacce di un sottosistema
- Sinonimo
  - [no]
- Motivazione
  - è richiesta un'interfaccia comune ed unificata per un insieme disparato di implementazioni (i.e. classi o interfacce), come se si stesse identificando un sottosistema
  - minimizzare le comunicazioni e le dipendenze tra sottosistemi
  - mascherare l'implementazione di un sottosistema
    - l'implementazione può cambiare nel tempo ma non le funzionalità offerte

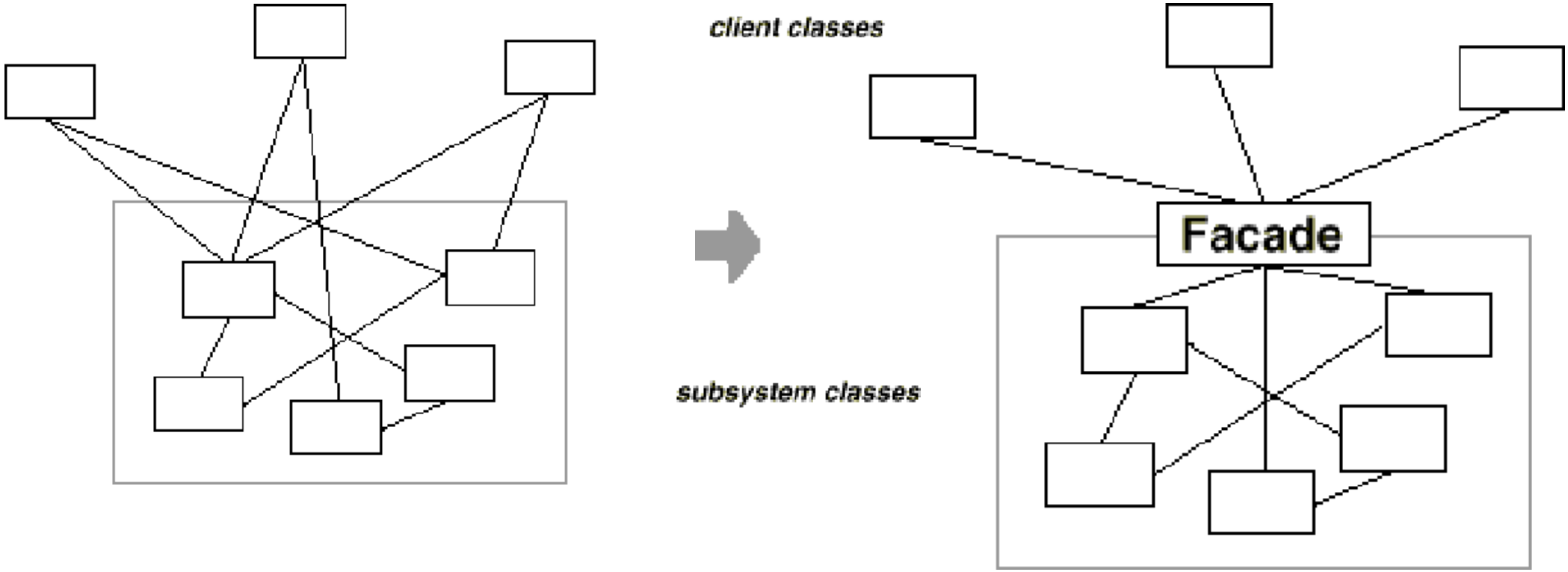
façade – 2

- Motivazione (2)



façade – 2

- Motivazione (2)

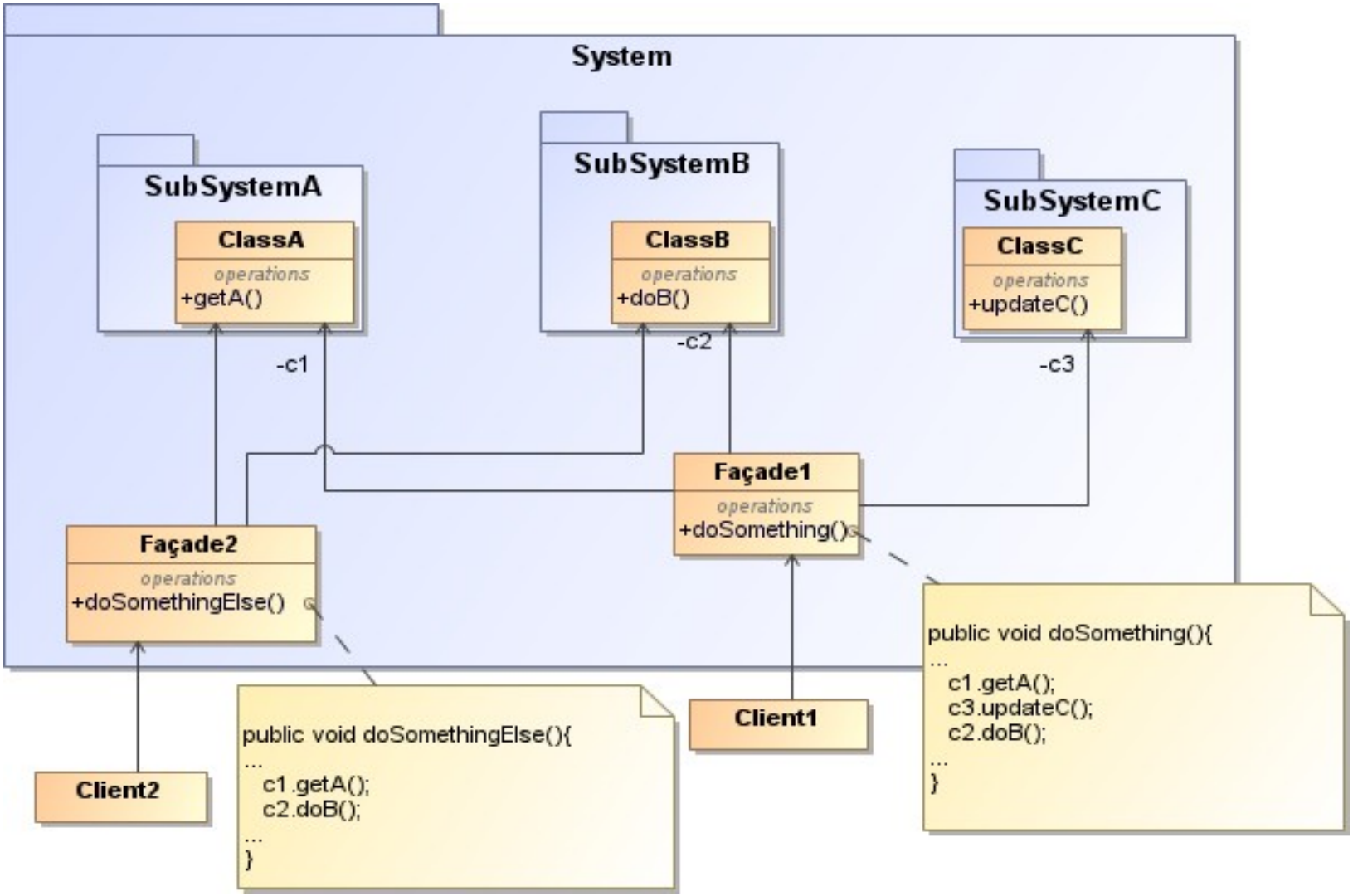


façade – 3

- Applicabilità
  - progettazione architetturale di un sistema
  - fornire una interfaccia ad un sistema complesso o che evolve nel tempo
  - aumentare il grado di riuso di un sottosistema
  - stratificare un sistema identificando i punti di accesso ad ogni sottosistema
  - ci sono molte dipendenze tra l'implementazione del sottosistema ed i suoi client
  - un client per realizzare una singola operazione logica deve accedere a più classi del sottosistema molto differenti tra loro

façade – 4

- Struttura



façade – 5

- Partecipanti
  - System (`System`)
    - definisce un sistema del quale si vuole nascondere i dettagli implementativi all'esterno
  - SubSystem (`System::SystemA, System::SystemB, System::SystemC`)
    - definisce eventuali sottomoduli del sistema
  - Façade (`System::Façade1, System::Façade2`)
    - definisce l'interfaccia comune per l'accesso alle funzionalità del sistema
    - può implementare logiche composizionali per esportare funzionalità di sistema
  - Client (`Client1, Client2`)
    - utilizzatore delle funzionalità del sistema

façade – 6

- Collaborazioni
  - i Client comunicano con il sistema attraverso l'interfaccia comune esportata (i.e. Façade)
  - i Client non hanno in alcun modo accesso agli oggetti dei sottositemi
- Conseguenze
  - riduce il numero di oggetti che un client deve gestire
  - rende il sistema più riusabile
  - riduce il grado di accoppiamento tra un sistema ed i suoi client, mitigando la ripercussione delle modifiche sul sistema anche sui client
  - la strutturazione in livelli consente di progettare sistemi indipendenti, con un basso tasso di dipendenze circolari

façade – 7

- Codice di esempio :

VEDERE MATERIALE ALLEGATO ALLA LEZIONE

pattern comportamentali

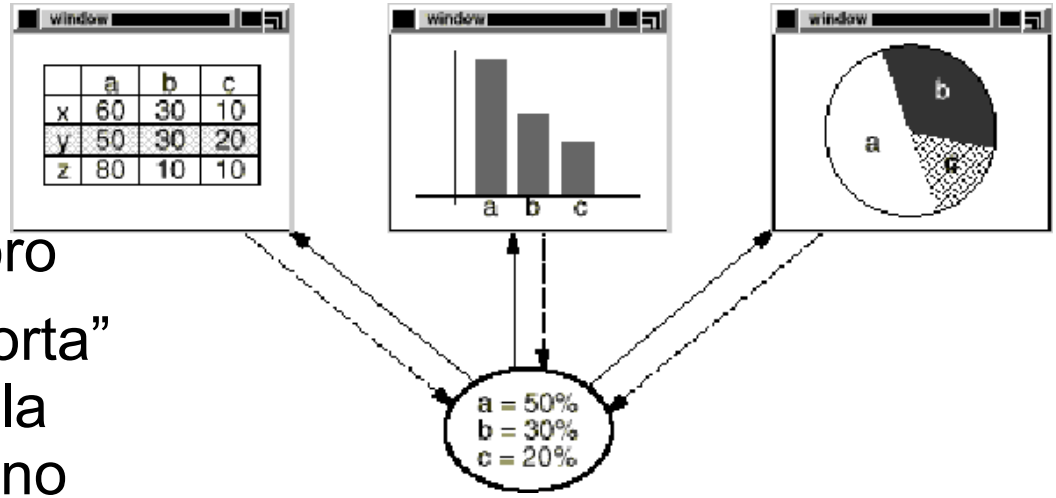
- definiscono i modi di comunicazione tra le classi e gli oggetti coinvolti nel pattern

observer – 1

- Scopo
  - definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente
- Sinonimi
  - Dependents, Publish-Subscribe
- Motivazione
  - classi differenti ed indipendenti che operano su una sorgente di dati o su una classe comune, e che quindi devono essere notificate delle modifiche a tali dati, o dei cambiamenti di stato subiti dalla classe
  - una applicazione complessa dovrebbe consentire la separazione tra l'interfaccia grafica presentata all'utente e la sottostante struttura dati dell'applicazione. Le classi che compongono l'applicazione e quelle che si occupano della presentazione dei dati dovrebbero essere "indipendentemente riusabili", ma in qualche modo devono poter cooperare

observer – 2

- Motivazione (2)
  - in un foglio di calcolo, pannelli di controlli diversi che mostrano diagrammi diversi per gli stessi dati
    - le classi “foglio elettronico”, “barre”, “torta” non sono direttamente dipendenti tra loro
    - “foglio elettronico”, “barre”, “torta” dipendono indirettamente dalla classe “dati” che rappresentano
    - il cambio di informazioni in “foglio elettronico” impone che le modifiche vengano apportate anche sulle altre classi (eventualmente viceversa)

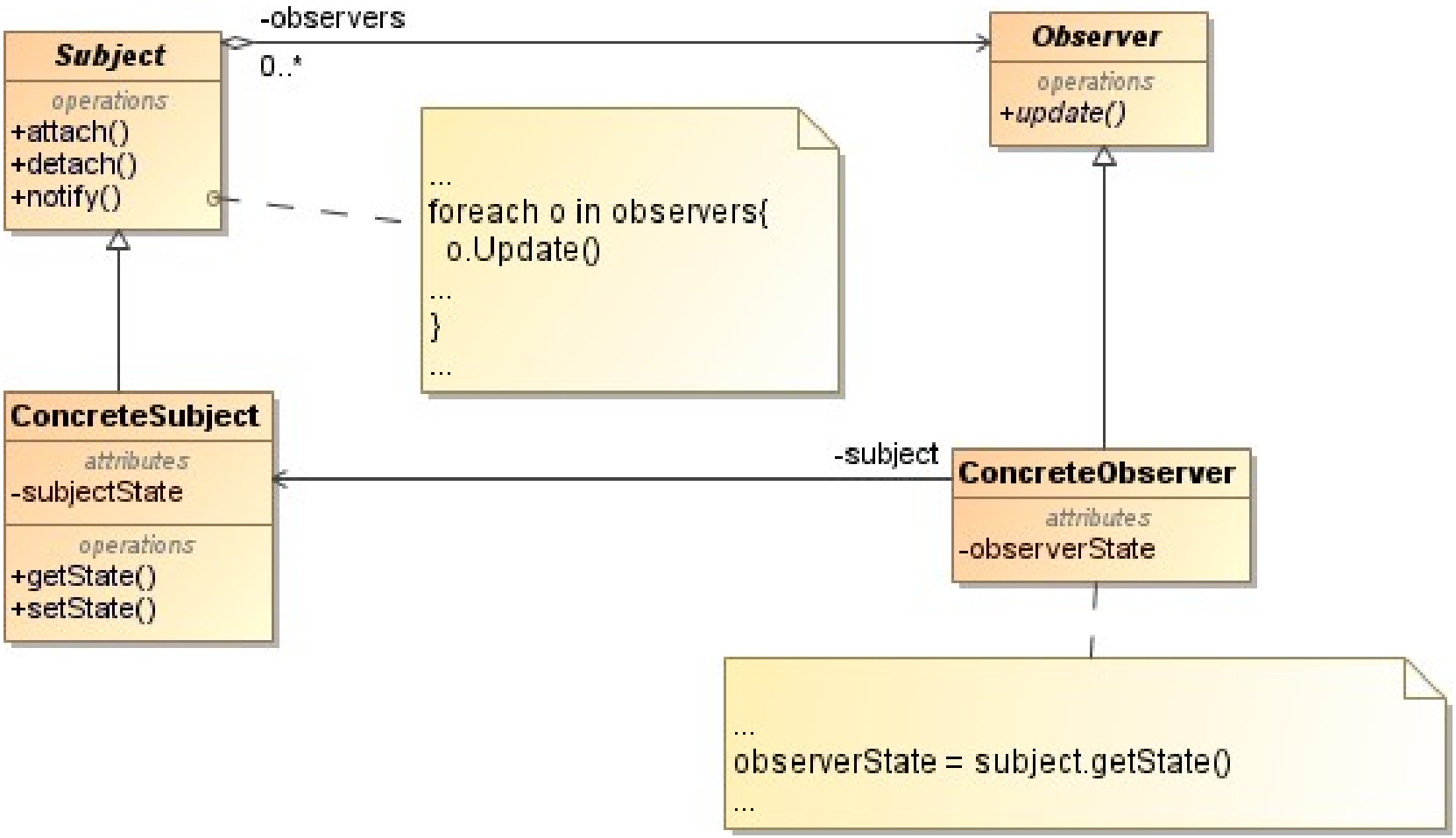


observer – 3

- Applicabilità
  - un'unica classe presenta di fatto due o più aspetti, dipendenti tra loro. Incapsulando questi aspetti in classi separate è possibile riusarli indipendentemente
  - gestire il fatto che una modifica ad una oggetto richiede modifiche anche agli oggetti da questo che dipendono, anche se non si conosce il numero di oggetti dipendenti
  - l'oggetto modificato ha bisogno di “notificare” agli altri oggetti senza conoscerne l’identità precisa. In altre parole si vuole mantenere un alto livello di disaccoppiamento

observer – 4

- Struttura



observer – 5

- Partecipanti
  - Subject (`Subject`)
    - conosce i propri `Observer`. Un numero qualsiasi di oggetti `Observer` può osservare un oggetto
  - Observer (`Observer`)
    - fornisce un'interfaccia di notifica per gli oggetti a cui devono essere notificati i cambiamenti del `Subject`
  - ConcreteSubject (`ChatClient`)
    - contiene lo stato a cui gli oggetti `ConcreteObserver` sono interessati
  - ConcreteObserver (`ComeBackObserver, OnOffObserver`)
    - memorizza un riferimento a un oggetto `ConcreteSubject`
    - contiene informazioni che devono essere costantemente sincronizzate con lo stato del `Subject`
    - implementa l'interfaccia di notifica di `Observer` per mantenere il proprio stato consistente con quello del `Subject`

observer – 6

- Collaborazioni
  - `ConcreteSubject` notifica ai propri `Observer` quando il suo stato cambia in modo tale da poter rendere consistente lo stato degli `Observer` con il proprio
  - a seguito di una notifica sul cambio di stato nel `ConcreteSubject` il `ConcreteObserver` può richiedere ulteriori informazioni riguardo al `Subject`. `ConcreteObserver` userà le informazioni ottenute dal `Subject` per sincronizzare il proprio stato

observer – 8

- Conseguenze
  - è possibile variare soggetti e osservatori in modo indipendente. Inoltre, è possibile riusare oggetti senza riusare i loro osservatori e viceversa.
  - accoppiamento astratto far `Subject` e `Observer`
    - i `Subject` conoscono una lista di `Observer` conformi ad una specifica interfaccia.
    - i `Subject` non conoscono nessuna classe `ConcreteObserver`.
    - accoppiamento tra `Subject` o `Observer` è astratto e minimale
  - supporto per comunicazioni “broadcast”
    - la notifica è inoltrata NON a uno specifico destinatario MA a tutti gli `Observer` interessati che si non registrati.
    - il `Subject` non si occupa di quanti `Observer` sono presenti; la sua unica responsabilità è inoltrare a ognuno di loro la notifica
    - è possibile togliere o aggiungere osservatori in qualsiasi momento
    - `ogni ConcreteObserver` deciderà se e come reagire
  - aggiornamenti inattesi
    - gli `Observer` sono tra loro indipendenti e possono ignorare gli effetti di una richiesta di modifica sul `Subject`
    - una modifica può scatenare una catena di aggiornamenti/sincronizzazioni altri `Observer`
    - il protocollo di notifica “base” non fornisce dettagli su cosa sia cambiato effettivamente nel `Subject`
    - necessità di protocolli aggiuntivi che aiutino gli `Observer` a capire/gestire il cambiamento nel `Subject`

observer – 8

- Codice di esempio :

VEDERE MATERIALE ALLEGATO ALLA LEZIONE

bibliografia di riferimento

- “Design Patterns – Elementi per il riuso di software a oggetti”, Gamma, Helm, Johnson, Vlissides (GoF). Addison-Wesley (1995).
  - Design Patterns: Elements of Reusable Object-Oriented Software
- “Applicare UML e i Pattern – Analisi e progettazione orientata agli oggetti”, Larman. 3za Edizione. Pearson (2005).
  - “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”