Sistemi Informativi:

Progetto del sistema

sommario

- Alcuni schemi di modellazione:
  - proxy
  - fabbrica astratta
  - costruttore
  - comando
  - osservatore
  - strategia
- I frameworks (infrastrutture)

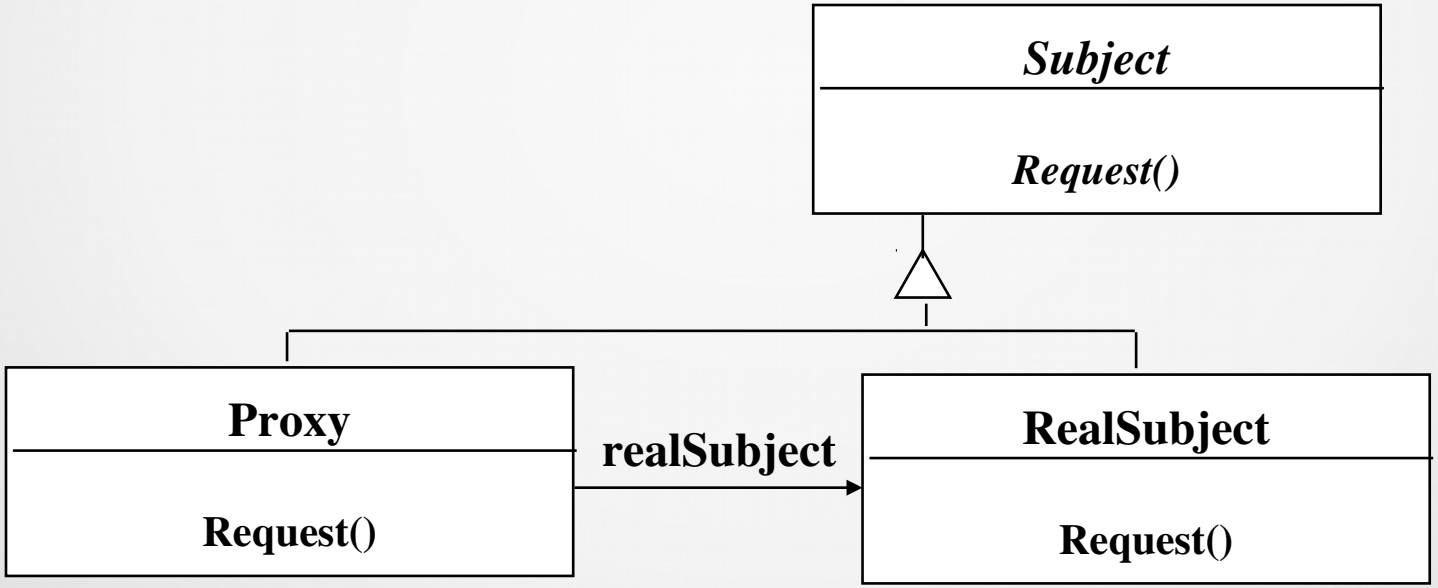
Interfacce e classi in java

- un'interfaccia definisce costanti e metodi astratti.
  - puo'estendere * interfacce
  - non definisce attributi o comportamento
- una classe definisce attributi e metodi
  - puo' estendere 0..1 classi
  - puo' implementare * interfacce
- le classi astratte si realizzano con classi o interfacce java (criterio?)
- le concrete con classi java

schema “proxy”

- e’ un oggetto che rimpiazza quello reale
- crea l’oggetto reale solo quando serve realmente
- es connessione a database remoto:
  - creo un proxy che memorizza le precedenti query senza riinvocare il server
- e’ un adattatore per la dinamica anziche’ la statica dell’interfaccia

- proxy ed oggetto vero condividono la stessa interfaccia
- il proxy intercetta le chiamate all'oggetto vero tramite delegazione

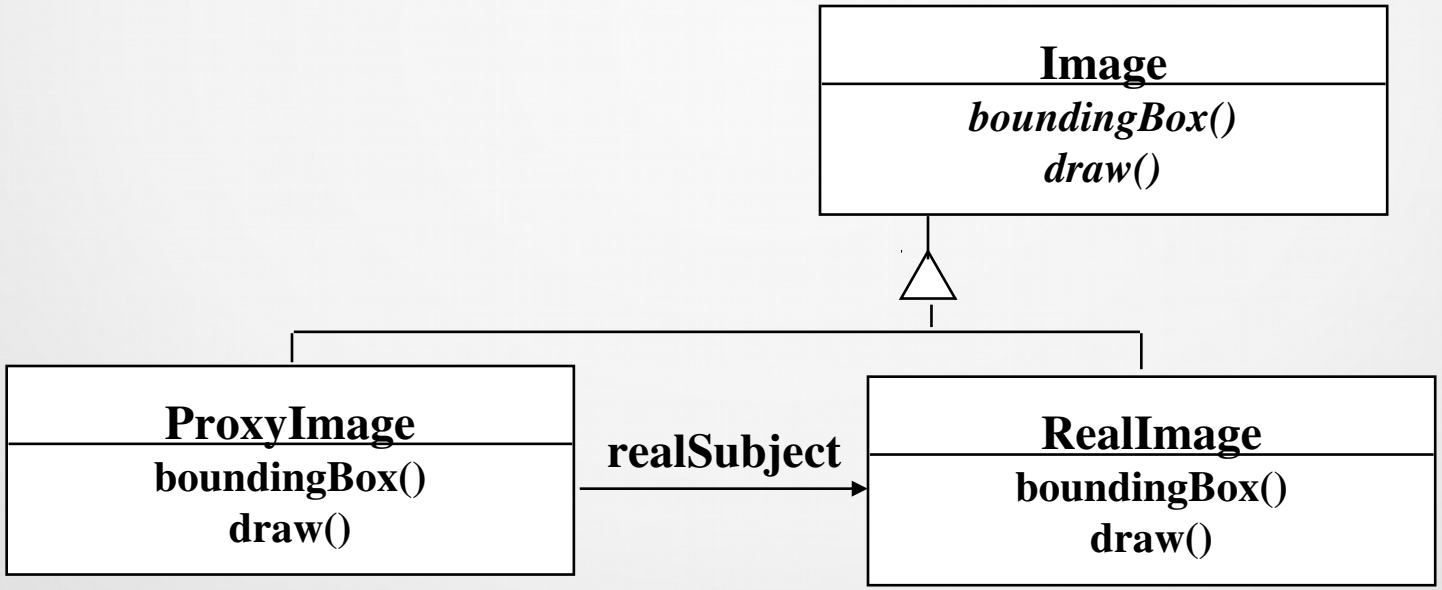


tipi di “proxy”

- Proxy remoto
  - rappresentante locale di oggetto remoto, usabile per conservare informazioni che non cambiano troppo di frequente (cache)
- Proxy virtuale
  - se l’oggetto e’ troppo costoso da scaricare o creare
- Proxy di protezione
  - quando l’oggetto vero deve essere acceduto con diritti differenti

proxy virtuale

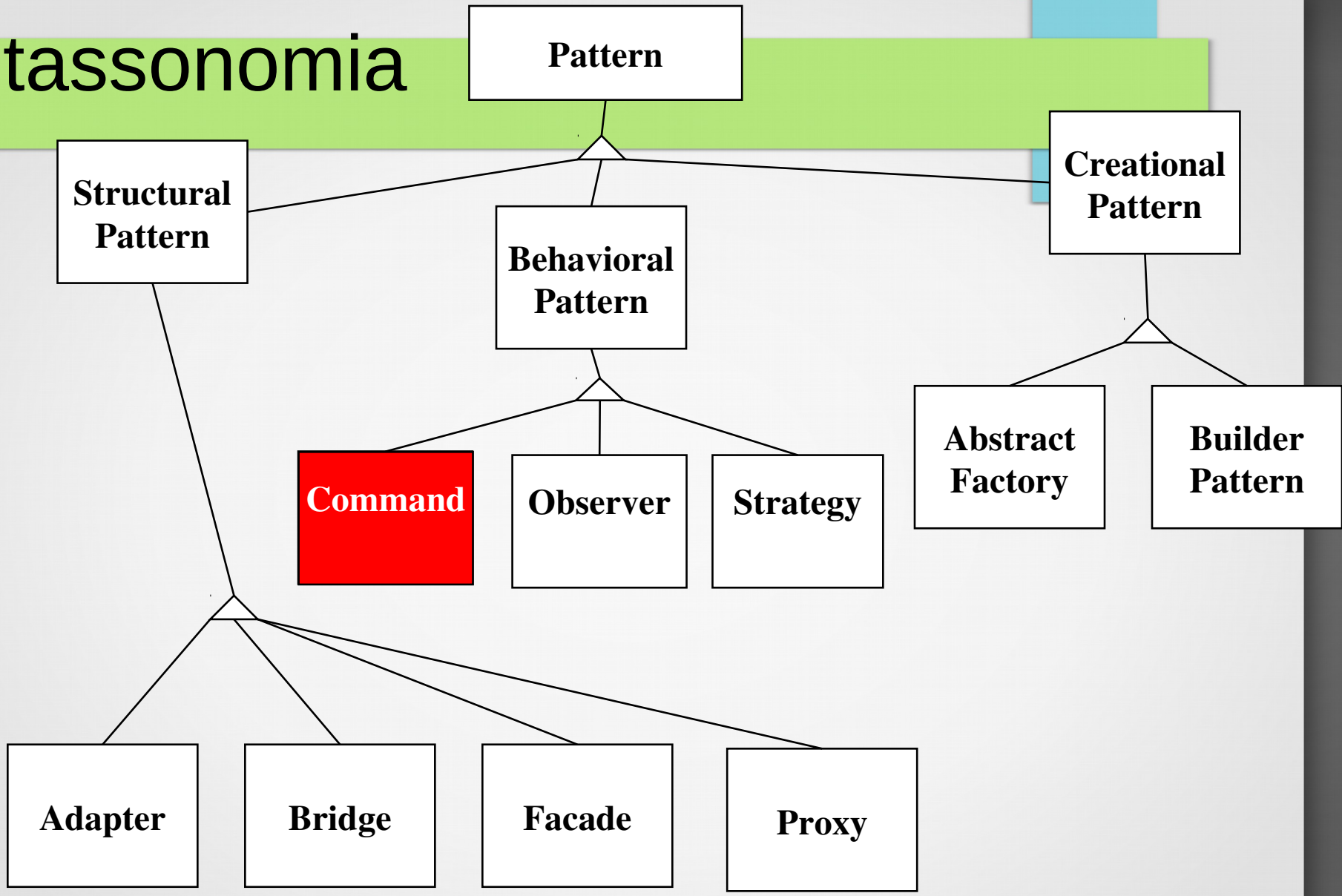
- le immagini sono memorizzate separatamente dal testo
- di una immagine reale, un proxy mostra una area grigia
- il client non sa se sta trattando l'immagine reale o no



classificazione di schemi

- strutturali: adattatori, ponti, facciate, proxy:
  - riducono l’accoppiamento tra oggetti
- comportamentali:
  - riguardano il comportamento e l’assegnazione dei compiti
  - caratterizzano flussi di controllo complessi
- creazionali:
  - astraggono il processo di creazione di istanze
  - rendono il sistema indipendente dal modo di rappresentazione degli oggetti

tassonomia

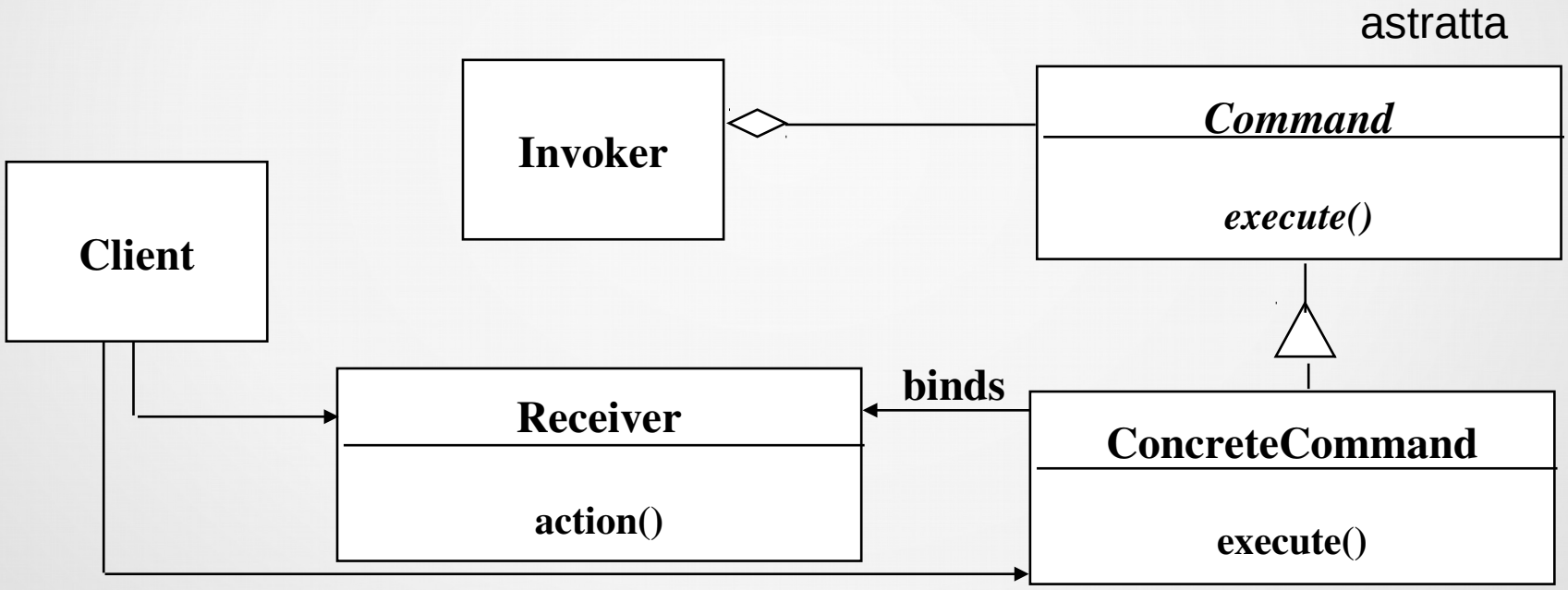

Pattern
Structural Pattern
Behavioral Pattern
Creational Pattern
Command
Observer
Strategy
Abstract Factory
Builder Pattern
Adapter
Bridge
Facade
Proxy

schema “comando”

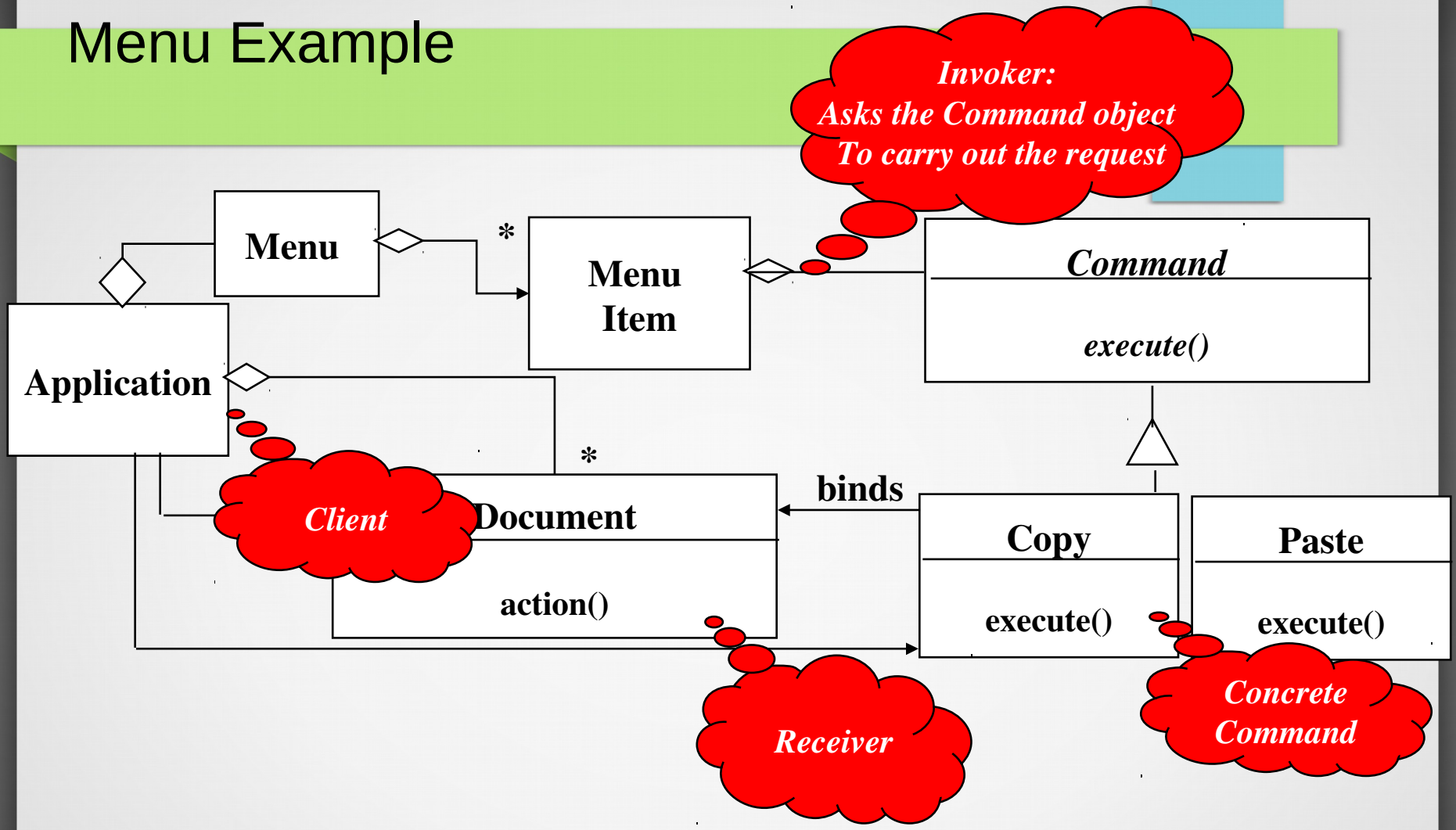
- es. si vuole costruire una GUI con menu etc. riutilizzabile in diverse applicazioni
- non si vuole inserire il significato dei comandi direttamente nel programma
- l’applicazione sa solo che un certo comando e’ stato selezionato
- proviamo diagramma sequenza delle due slide dopo

schema “comando”

- il cliente crea un comando concreto, lo lega al ricevente, lo passa all’esecutore
- (es GUI) – di nuovo indirezione



Menu Example



uso dello schema “comando”

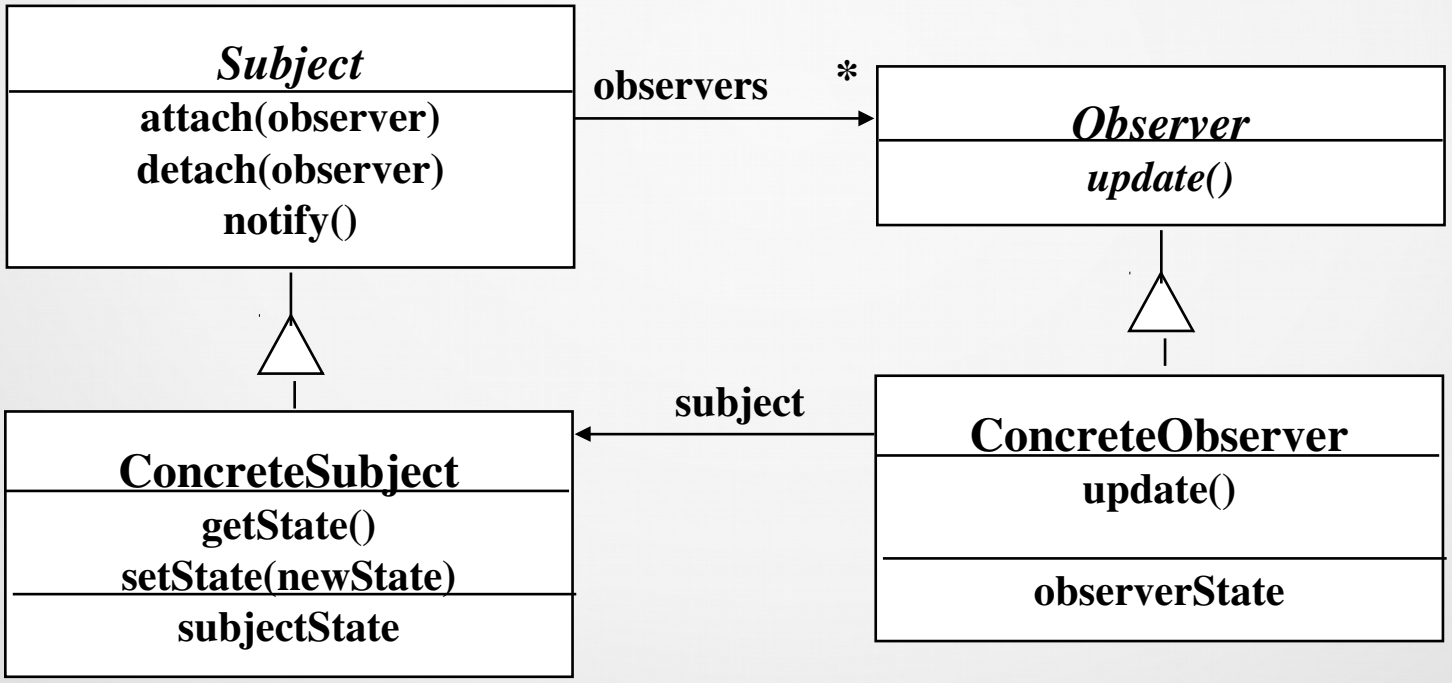
- incapsulare una richiesta in un oggetto
  - parametrizzazione di un cliente con richieste differenti
  - code di richieste, traccia di richieste
  - undo

schema “osservatore”

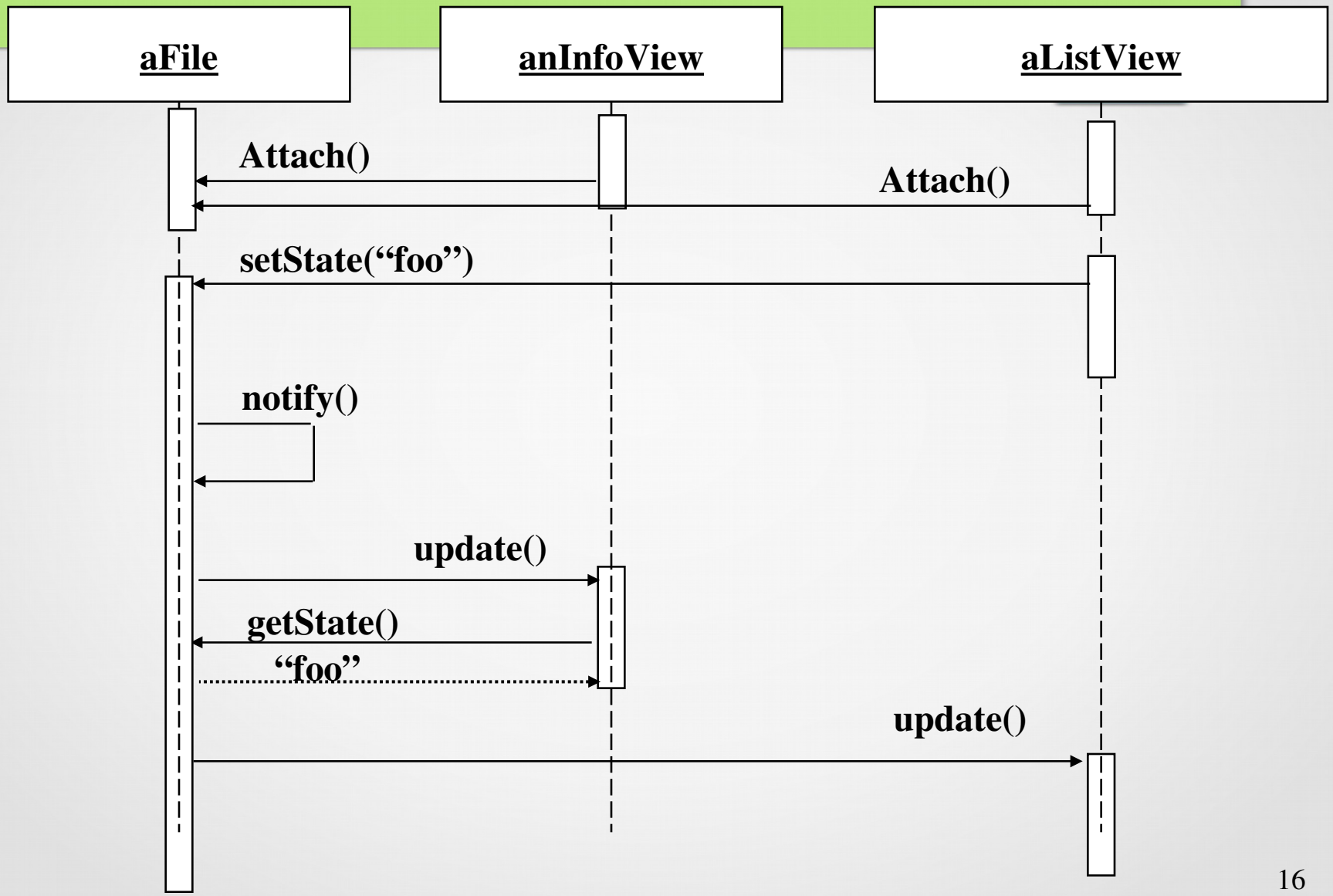
- stabilisce una relazione tra un oggetto ed un insieme di oggetti in modo che le modifiche al primo sono notificate ai secondi
- anche detto “publish and subscribe”
- usato per ottimizzare e mantenere la consistenza in un gruppo di modifiche
- usato anche per rendere piu’ flessibile sistemi di scambio messaggi

schema “osservatore”

- il soggetto rappresenta lo stato, gli osservatori delle differenti viste su di esso
- il soggetto memorizza l’insieme di osservatori



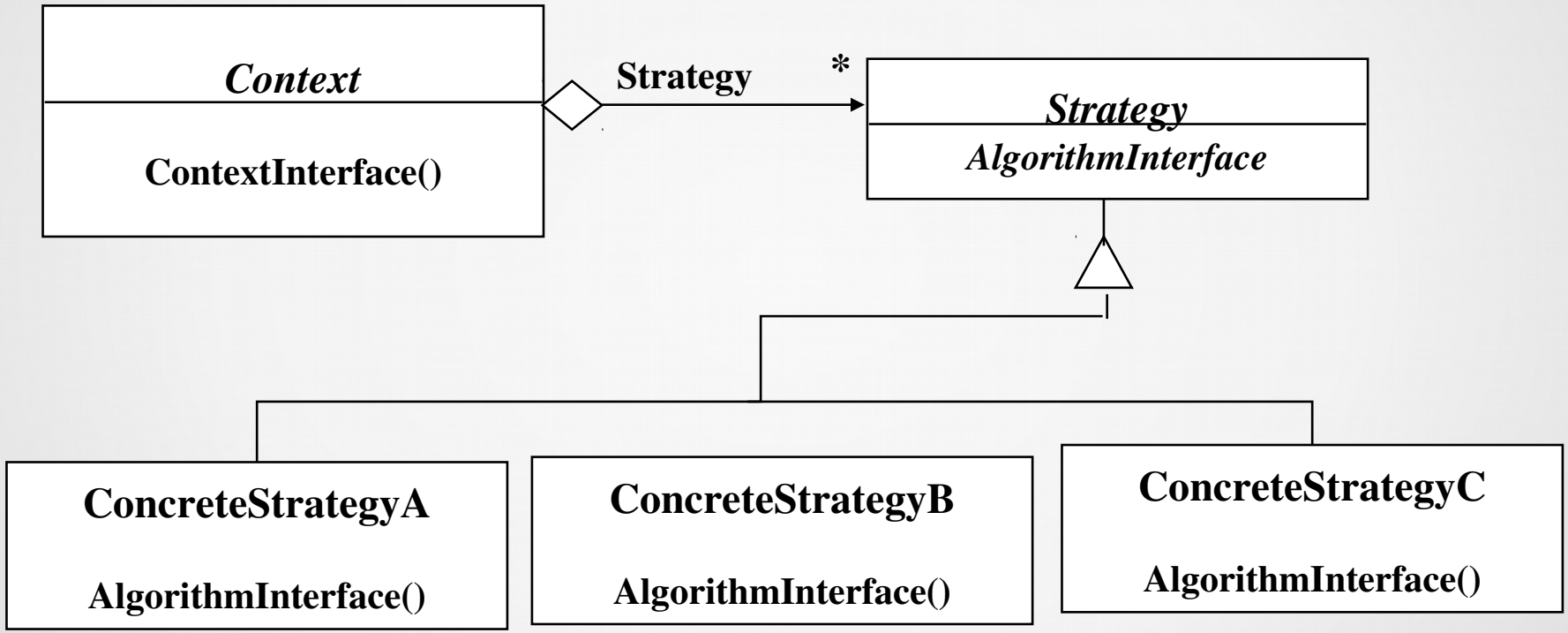
Sequence diagram for scenario: Change filename to “foo”



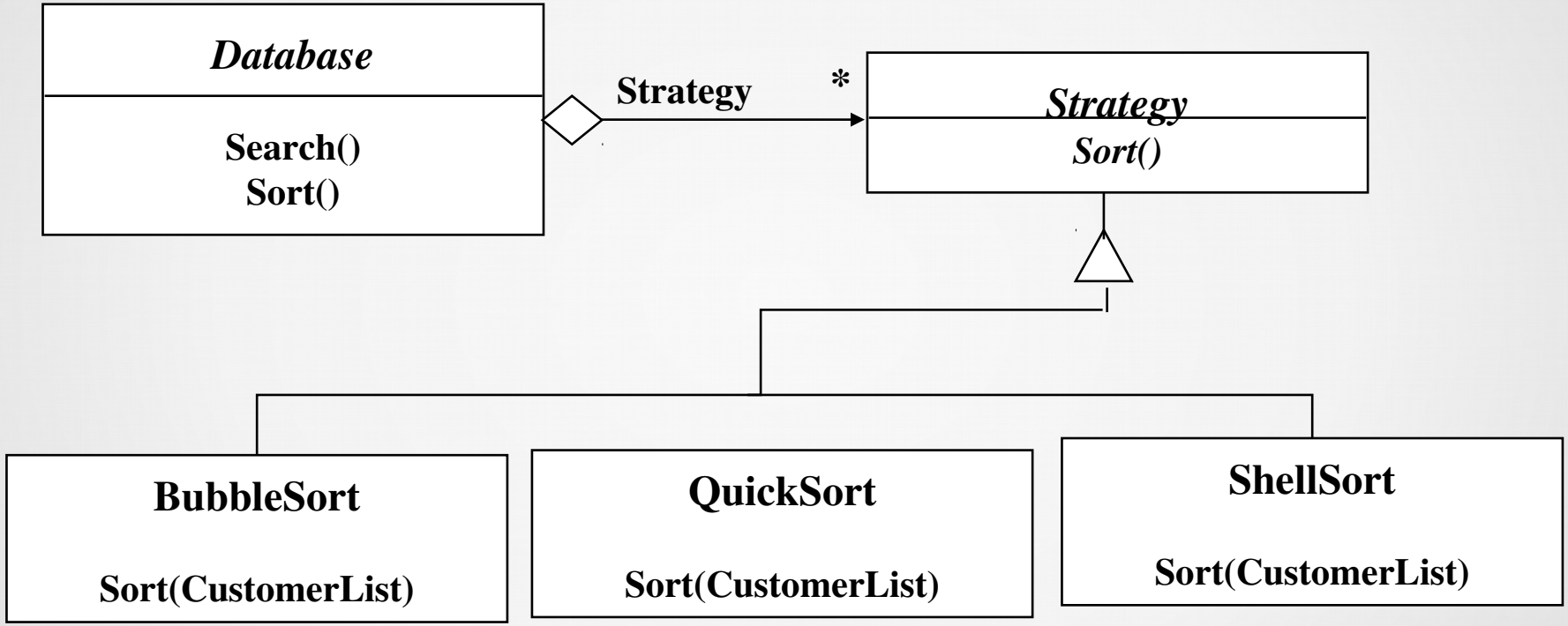
schema “strategia”

- se molti algoritmi esistono per risolvere un problema, es ordinare una lista di clienti
- i differenti algoritmi saranno appropriati in momenti diversi, es prototipazione vs prodotto finale.
- vogliamo poter aggiungere algoritmi senza modificare l’applicazione

schema “strategia”



esempio applicazione DB

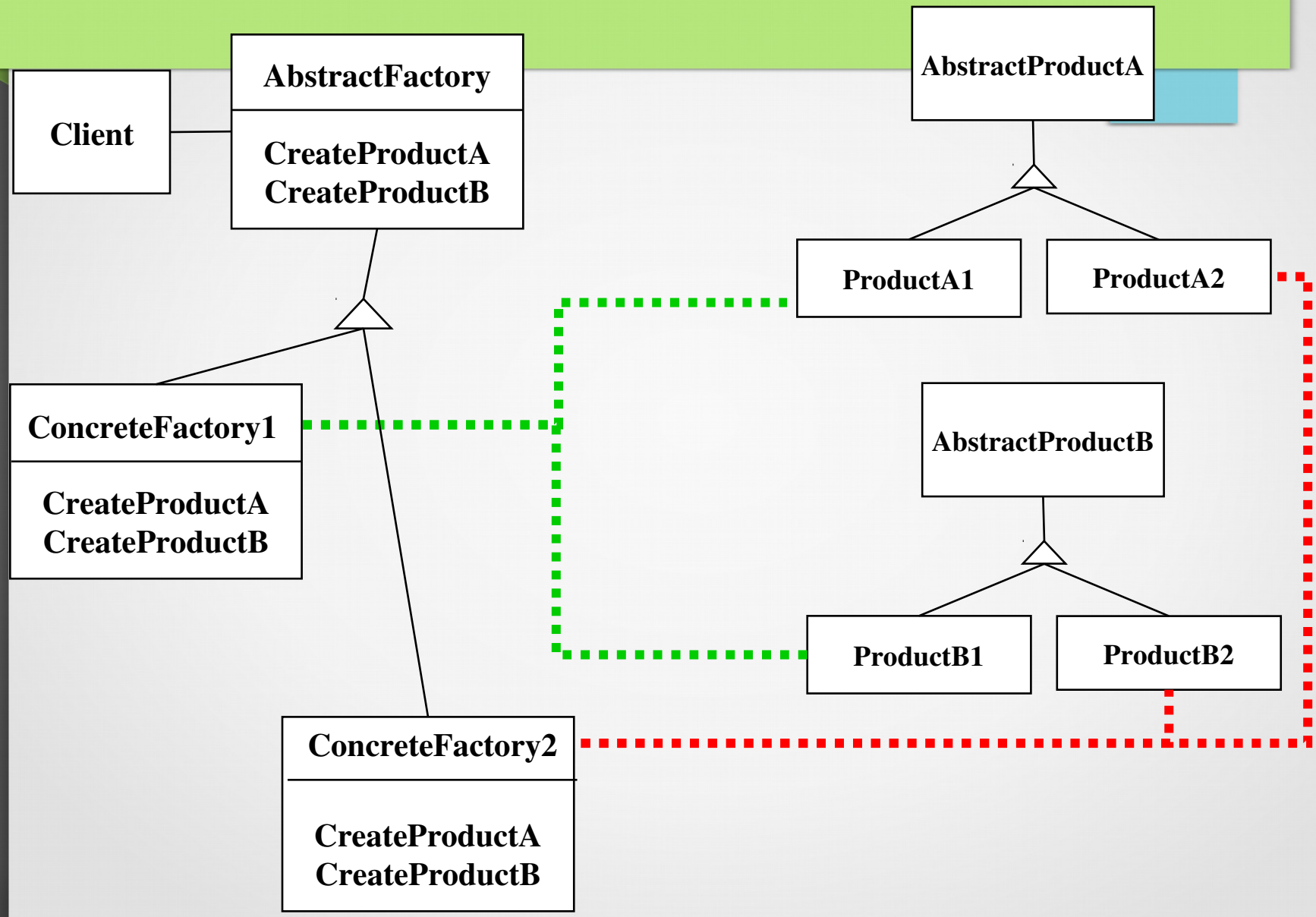


applicazione dello schema “strategia”

- molte classi differiscono solo nel loro comportamento: si puo’ creare una singola classe con comportamenti differenti
- algoritmi differenti con differenti prestazioni spazio/tempo

schema “abstract factory”

- supponiamo di creare una libreria che deve supportare contemporaneamente interfacce tipo windows, motif etc
- una interfaccia che supporti DB relazionale e legacy


Client
AbstractFactory
CreateProductA
CreateProductB
AbstractProductA
ProductA1
ProductA2
ConcreteFactory1
CreateProductA
CreateProductB
AbstractProductB
ProductB1
ProductB2
ConcreteFactory2
CreateProductA
CreateProductB

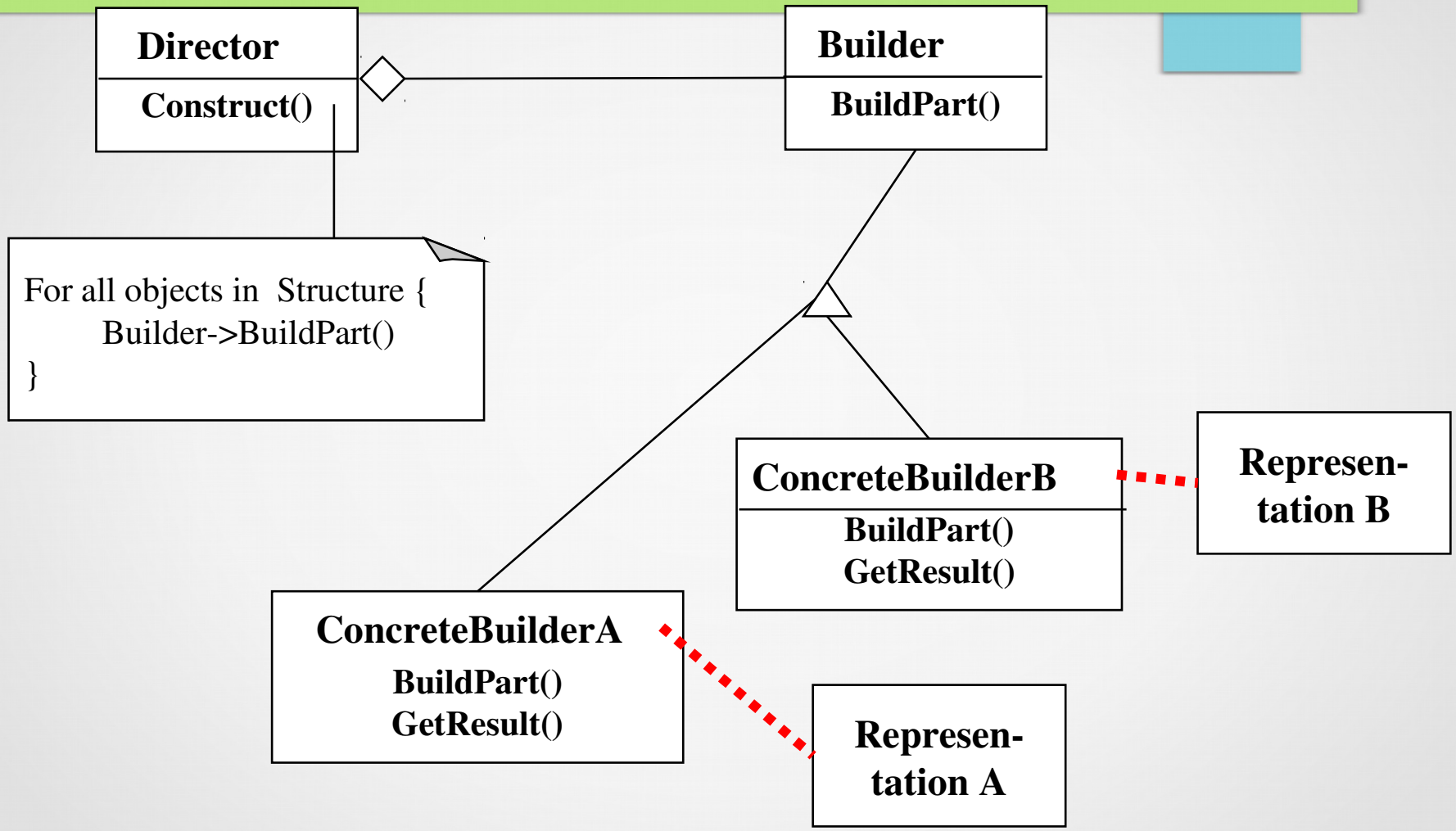
applicare “fabbrica astratta”

- indipendenza da rappresentazione o creazione
- indipendenza dal venditore
- possibili sviluppi futuri con famiglie diverse di prodotti

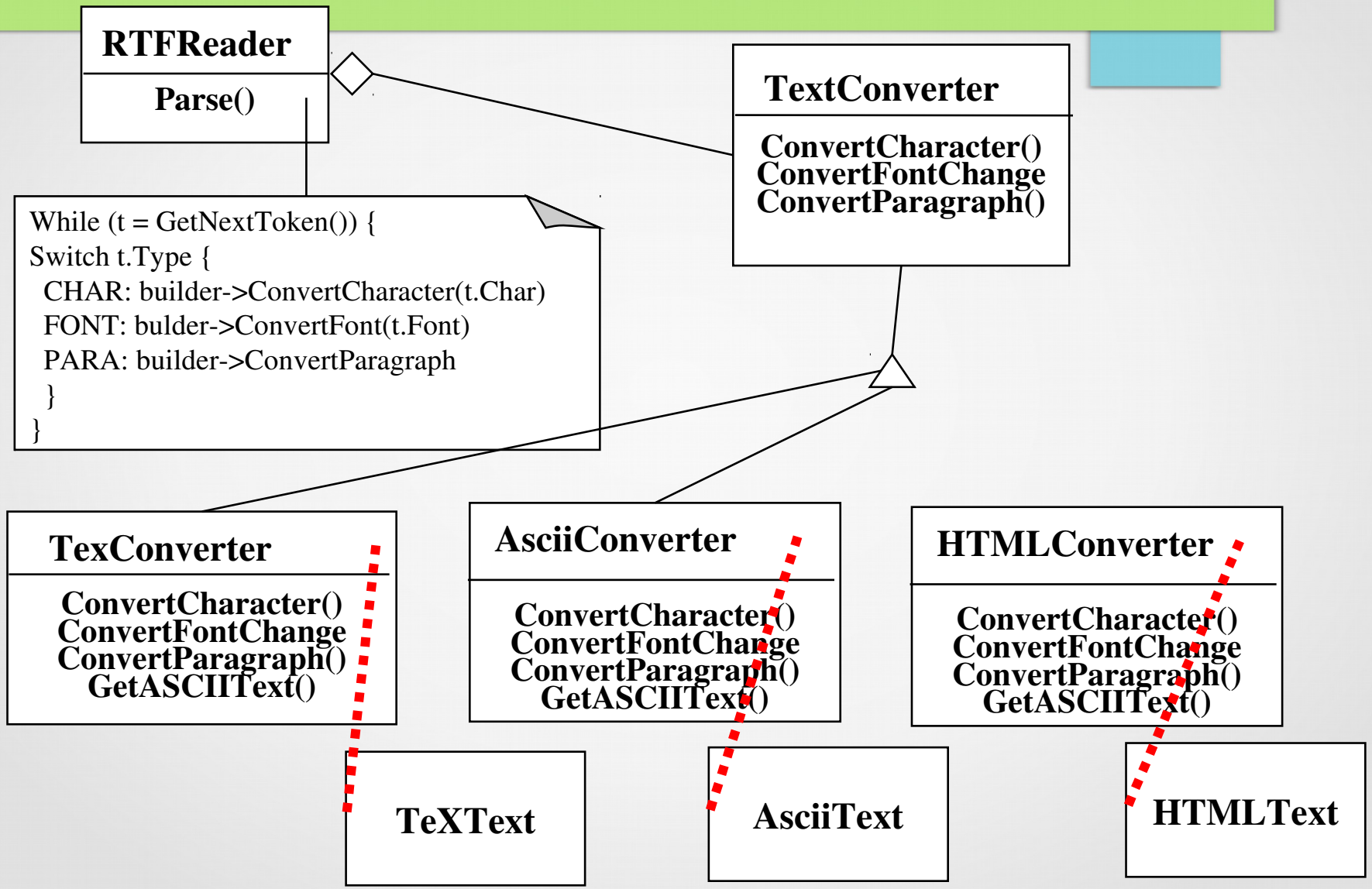
schema “builder”

- se dobbiamo lavorare con un documento in formati diversi es testo word, pdf, ascii etc.
- costruiamo un oggetto che puo’ effettuare tutte le conversioni nei vari formati
- processo senza fine

schema “builder”



Esempio


RTFReader
Parse()
TextConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
While (t = GetNextToken()) {
Switch t.Type {
CHAR: builder->ConvertCharacter(t.Char)
FONT: bulder->ConvertFont(t.Font)
PARA: builder->ConvertParagraph
}
}
TexConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
AsciiConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
HTMLConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
TeXText
AsciiText
HTMLText

“abstract factory” e “builder”

- abstract factory
  - usata su famiglie di prodotti, semplici o complessi
  - non nasconde il processo di creazione
- builder
  - il prodotto e’ costruito in modo complesso
  - ogni tanto la costruzione cambia
  - la creazione e’ un singolo passo per il cliente

spunti ipotetici per i pattern da requisiti nonfunzionali

- “indipendente dal fabbricante”, “dal device”, “deve supportare un gruppo di prodotti”:
  - abstract factory
- “deve interfacciare un prodotto esistente”
  - adapter
- “deve interfacciare con diversi sistemi, alcuni futuri” occorre un prototipo dimostrabile
  - bridge

esempio di spunti per i pattern da requisiti nonfunzionali

- “struttura complessa”, “variabile ampiezza e profondita’”
  - composite
- “deve interfacciare un insieme di oggetti esistenti”
  - facade
- “trasparente alla locazione”
  - bridge
- “estensibile”, “scalabile”
  - observer
- “deve fornire una politica indipendente dal meccanismo”
  - strategy

occorre conoscere tutti questi pattern?

- quando si deve esaminare il codice o la documentazione relativa ad un progetto
- In definitiva tutti i pattern hanno lo scopo di aumentare la flessibilita’ e indipendenza di un sistema inserendo indirettezza nei riferimenti tra oggetti

Framework (scheletro di applicazione, infrastruttura)

- Un framework e' una architettura SW riutilizzabile che fornisce una struttura generica per una classe di applicazioni
- Una estensione di pattern specifica ad un problema
  - contiene un insieme di schemi con elementi di supporto
  - scheletri di soluzione in cui inserire specifici componenti per il suo completamento
  - rappresentati come package collo stereotipo di <<framework>>

Framework 2

- Scheletro non modificabile
- Comportamento di default
- Costituito da un insieme di classi istanziabili o specializzabili per implementare un’applicazione
- dispone di algoritmi che implementano strutture di controllo in cui inserire le invocazioni alle proprie classi (Inversione di controllo)

Framework 3

- occorre conoscere bene l’architettura per poterla utilizzare
- soluzione standardizzata
- la strutturazione in componenti e le loro interfacce sono in parte predefinite
- esempi Application Servers per applicazioni con Servlet (architettura EJB)
- Applicazioni web
- Windowing systems

conclusioni

- come avete visto, tutti questi pattern sono semplicemente la razionalizzazione di buone idee per evitare di riinventare la ruota
- spesso questi schemi sono ormai utilizzati in librerie (es quelle grafiche).
- se le usate, siete praticamente obbligati a seguirle
- esempi diagrammi di sequenza