Laboratorio di sistemi interattivi

Lezione 5: Introduzione ai Design Pattern

Elementi Essenziali di un Design Pattern

- Nome del Pattern
- Problema che esso risolve
- Soluzione fornita
- Consequenze (positive e negative) del suo uso

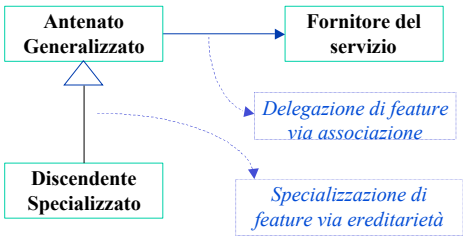
Presentazione nel libro della Gang of Four

- Pattern Name
- Jurisdiction
- Intent
- Motivation
- Applicability
- Participants
- Collaborations
- Structure Diagram
- Consequences
- Implementation
- Examples
- See Also

Fattori di qualità

- Pattern catturano soluzioni, non principi o strategie astratti.
- Soluzioni devono essere comprovate, non teorie o speculazioni.
- Soluzioni non ovvie. Possono risultare indirettamente.
- Descrivono relazioni, non solo moduli, ma strutture e meccanismi del sistema più profondi.

Distribuzione delle Responsabilità



Delegation vs. Specialization

- Delegation
  - Approccio orientato alle componenti
  - Distribuisce caratteristiche fra gli oggetti
  - Associazioni variabili a runtime
- Specialization
  - Approccio architetturale
  - Distribuisce caratteristiche fra classi correlate
  - Gerarchie fissate

Design Pattern Space

- **Creational pattern**
  - Descrivono strategie per inizializzare e configurare sia classi (l'ereditarietà permette di variare la classe istanziata) che oggetti (l'istantiation è delegata ad un altro oggetto)
  - Aiutano a rendere un sistema indipendente da come gli oggetti sono creati, composti e rappresentati
  - Sono importanti quando un sistema dipende maggiormente da come gli oggetti sono composti che dall'ereditarietà
  - Pochi comportamenti fondamentali vengono composti per ottenerne di più complessi
  - Si possono creare oggetti il cui comportamento particolare richiede più che instanziare una classe.

Design Pattern Space

- **Structural pattern**
  - Forniscono strategie per comporre classi ed oggetti per creare strutture più grandi
  - A livello di classi utilizzano l'ereditarietà per comporre interfacce
  - A livello di oggetti descrivono modi per comporli per ottenere nuove funzionalità

Design Pattern Space

- **Behavioral pattern**
  - Descrivono interazioni dinamiche (pattern di comunicazione) tra società di classi ed oggetti
  - Aiutano a gestire la distribuzione delle responsabilità tra oggetti
  - A livello di classi viene usata l'erditarietà per distribuire i comportamenti
  - A livello di oggetti viene usata la composizione

Creational Patterns

- Factory Method (class creational)
  - Definisce una interfaccia per creare un oggetto, ma lascia alla sottoclasse decidere quale classe istanziare durante questa creazione
  - Es. Applicazione – Documento -> il tipo di applicazione incapsula il tipo di documento
- Abstract Factory (object creational)
  - Crea una Factory per costruire famiglie di oggetti correlati o dipendenti
  - Es. Diversi look-and-feel-> diverse presentazioni e comportamenti per i widget
  - Abstract WidgetFactory con interfacce per creare i widget di base (scroll-bar, window, ecc.) e classi astratte (AbstractProduct) per ogni widget – Sottoclassi concrete di WidgetFactory per ogni look-and-feel e per i widget
- Builder (object creational)
  - Crea una Factory per costruire oggetti complessi in maniera incrementale
- Prototype (object creational)
  - Crea una Factory per clonare nuove istanze da un prototipo
- Singleton (object creational)
  - Crea una Factory per assicurarsi che una classi abbia una unica istanza

Structural Patterns

- **Adapter (class, object structural)**
  - Adatta l'interfaccia di una classe (server) per farla apparire quella che il client si aspetta (wrapper)
- **Bridge (object structural)**
  - Disaccoppia un'astrazione dalla sua implementazione per poterli variare in maniera indipendente
- **Composite (object structural)**
  - Struttura per costruire aggregazioni ricorsive
- **Decorator (object structural)**
  - Estende le funzionalità di un singolo oggetto di una classe in maniera trasparente (includendolo in un altro oggetto)

Structural Patterns

- **Facade (object structural)**
  - Fornisce una interfaccia unificata (di alto livello) all'insieme delle interfacce di un sottosistema (solo client con esigenze avanzate vanno oltre la facciata)
- **Flyweight (object structural)**
  - Molto oggetti di grana fine (es. un oggetto per ogni lettera di un text editor) vengono gestiti in maniera efficiente tramite la condivisione e la distinzione tra stato intrinseco (a prescindere dal contesto) ed estrinseco (mantenuto dal contesto).
- **Proxy** (**(object structural)**
  - Un oggetto fa da punto di accesso ad un altro (protetto)

Behavioral Patterns

- **Chain Of Responsibility (object behavioral)**
  – Il mittente di una richiesta di servizio ed il ricevente sono disaccoppiati, e la richiesta scorre eventualmente lungo una catena finché non arriva ad un oggetto in grado di servirla (es. help contestuali)
- **Command (object behavioral)**
  – Incapsula i comandi in oggetti (astratti). I comandi concreti definiscono il collegamento tra un oggetto ricevente e un'azione (es. progettazione di menù)
- **Interpreter (class behavioral)**
  – Definisce una rappresentazione per la grammatica di un linguaggio insieme con un interprete adeguato (piccole grammatiche per esprimere istanze di un certo problema frequente come parole di un linguaggio)
- **Iterator (object behavioral)**
  – Definisce l'accesso sequenziale ad elementi aggregati senza scoprire i dettagli della loro organizzazione

Behavioral Patterns

- **Mediator (object behavioral)**
  – Un Mediator coordina le interazioni tra i suoi associati (es. un bottone disabilitato se un campo di testo è vuoto)
- **Memento (object behavioral)**
  – Cattura ed esternalizza lo stato interno di un oggetto (originator) in uno snapshot senza violare l'incapsulamento, in modo che questo stato possa essere ripristinato (es. per un undo)
- **Observer (object behavioral)**
  – Oggetti dipendenti (observer) ricevono automaticamente una notifica quando cambia lo stato del subject
- **State (object behavioral)**
  – Un oggetto può modificare il proprio comportamento in base al proprio stato

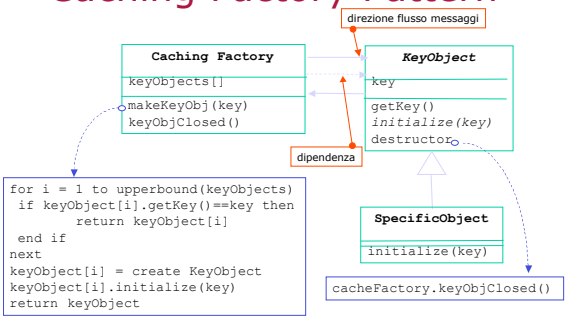
Behavioral Patterns

- **Strategy (object behavioral)**
  – Astrazione per selezionare uno di molti comportamenti (algoritmi)
- **Template Method (class behavioral)**
  – Definisce la struttura di un algoritmo per cui alcuni passi sono istanziati da sottoclassi
- **Visitor (object behavioral)**
  – Codifica operazioni che vanno applicate in maniera diversa agli elementi di una struttura eterogenea

Caching Factory Pattern

- **Problema:**
  - Aprire un singolo oggetto per ogni valore chiave
- **Soluzione:**
  - Un oggetto Factory apre gli oggetti per ogni chiave
  - Caching Factory mantiene i collegamenti con gli aggetti aperti
  - Restituisce un oggetto esistente se esiste per una chiave
  - Distrugge il collegamento quando l'oggetto viene chiuso
- **Conseguenze:**
  - Esercita alto grado di controllo su quando e come gli oggetti possono essere creati
  - Può limitare l'usabilità del sistema se applicata indiscriminatamente.

Caching Factory Pattern



Regole per flessibilità ed estendibilità

- Dare agli oggetti obiettivi ristretti
- Mantenere i metodi semplici e brevi
- Favorire Delegation rispetto Specialization
- Usare metodi di *Template* e *Hook* per massimizzare la riusabilità

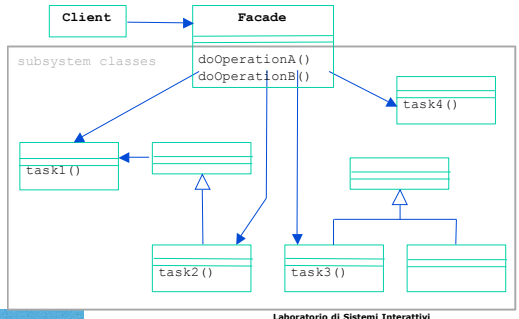
Principi chiave

1. Separare interfaccia dall'implementazione
2. Determinare cosa è *comune* e cosa è *variabile* in interfaccia e implementazione
   Comune == stabile
3. Permettere sostituzione di implementazioni variabili per mezzo di un'interfaccia comune
4. Dividere aspetti comuni da variabili dovrebbe essere rivolto allo scopo piuttosto che esaustivo

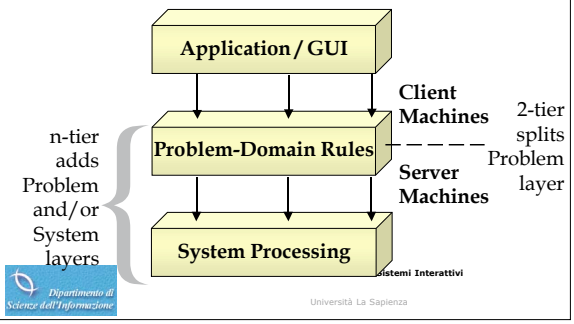
GOF Facade Pattern

- **Problema:**
  - Incapsulare un sottosistema usando un'interfaccia di alto livello, semplificando l'uso del sottosistema e nascondendo dettagli strutturali.
- **Problema:**
  - Reti complesse di oggetti sono difficili da controllare.
- **Soluzione:**
  - Clienti comunicano col sottosistema chiamando metodi di Facade.
  - Clienti non accedono mai (o il meno possibile) gli oggetti nel sottosistema.
  - Sottosistemi non mantengono conoscenza del cliente.
  - Oggetti del sottosistema non mantengono riferimenti a Facade.
- **Conseguenze**
  - Riduce il numero di oggetti a cui il cliente deve interfacciarsi.
  - Promuove basso accoppiamento che migliora la flessibilità generale.

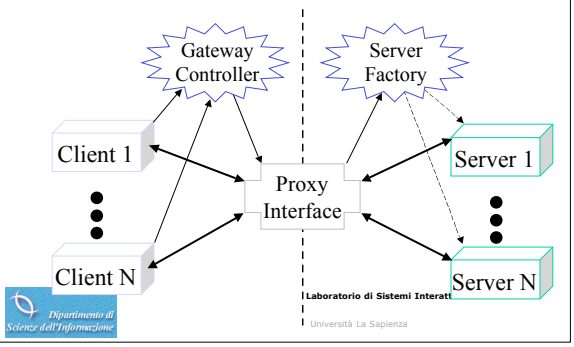
GOF Facade Pattern



Fundamental 3+-tier Partitioning Architecture



A Typical Distributed Client-Server Architecture



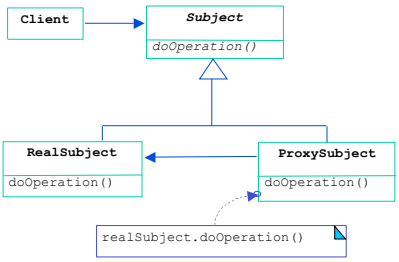
GOF Proxy Pattern

- **Problema:**
  - Fornire surrogato o segnaposto per controllare accesso a un oggetto.
- **Soluzione:**
  - Creare interfaccia pubblica o classe astratta Subject per l'oggetto di interesse.
  - Derivare classe RealSubject che implementa l'interfaccia.
  - Derivare un ProxySubject che crea il RealSubject e passa tutte le richieste ad esso.
- **Conseguenze:**
  - Adatta progetti OO inflessibili, spesso aggiungendo flessibilità.
  - Proxy può essere usata per distribuire oggetti a diversi spazi di oggetti o macchine in rete.
  - ProxySubject può arricchire il comportamento di RealSubject.

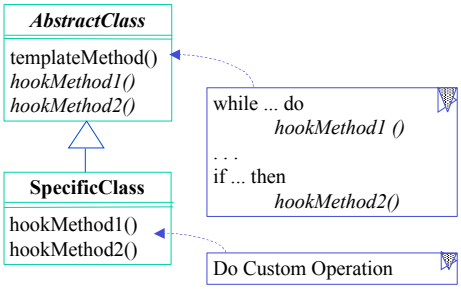
Laboratorio di Sistemi Interattivi

Università La Sapienza

GOF Proxy Pattern



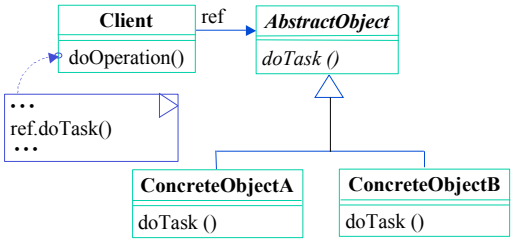
GOF Template Pattern



Zimmer's Objectifier Pattern

- **Problema:**
  - Permettere a un oggetto di variare una proprietà indipendentemente .
- **Soluzione:**
  - Creare una classe astratta che fornisce un'interfaccia comune per implementazioni specifiche della parte variabile.
  - Creare classi concrete discendenti da quella astratta per fornire l'implementazione .
  - Definire un riferimento alla classe astratta nel Client e assegnarla a un'istanza di classe concreta a tempo di esecuzione.
  - Delegare operazioni per il comportamento specifico all'oggetto specifico.
- **Conseguenze**
  - Comportamento bene incapsulato negli oggetti.
  - Strutture altamente flessibili e configurabili
  - Può penalizzare l'esecuzione a causa dell'indirezione

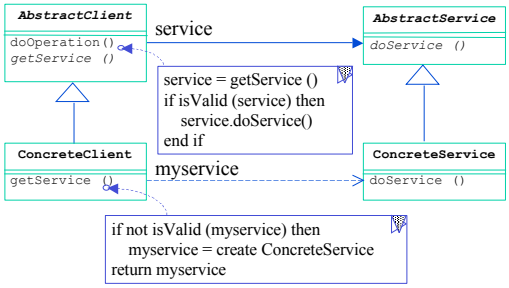
Zimmer's Objectifier Pattern



Flexible Service Pattern

- **Problema:**
  - Disaccoppiare un comportamento da una classe antenato: discendenti possono specificare comportamento flessibile.
- **Soluzione:**
  - Usare Objectifier per fornire una gerarchia di servizi variabili.
  - Usare il pattern Template per definire metodi nel client astratto che richiedono riferimento a servizio concreto con `getService()`.
  - Ridefinire `getService()` per ottenere servizi concreti.
- **Consequenze**
  - Framework molto flessibili e estendibili.
  - Client concreti controllano completamente il servizio e possono cambiare oggetti di servizio dinamicamente.
  - Penalizzazione da chiamata a `getService()` prima di ogni utilizzo.

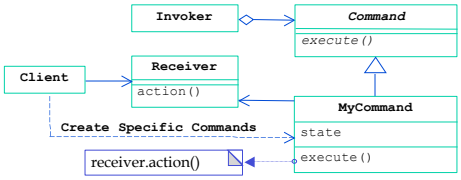
The Flexible Service Pattern



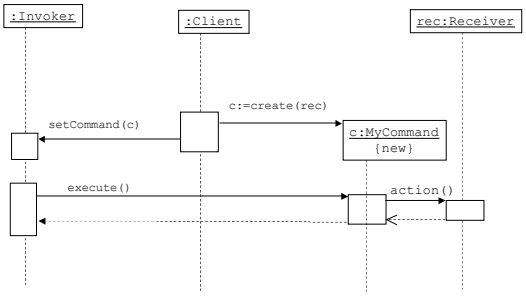
GOF Command Pattern

- **Problema:**
  - Incapsulare una richiesta come un oggetto parametrizzato, permettere richieste diverse; accodare o memorizzare richieste, supportare operazioni disfattibili.
- **Soluzione:**
  - Client crea comandi quando necessario, specificando il ricevitore e i parametri che il comando utilizzerà successivamente.
  - Ogni comando eredita da una classe astratta che definisce l'interfaccia di base di esecuzione, permettendo all'Invoker di attivare i comandi quando necessario.
  - Ogni comando conosce i metodi del suo Receiver.
- **Conseguenze**
  - Disaccoppia un oggetto dalle operazioni eseguite su di esso.
  - Comandi si possono estendere o aggregare.

GOF Command Pattern



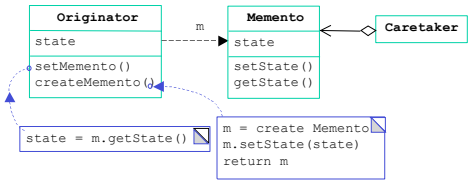
Collaborazione



Memento Pattern

- **Problema:**
  - Catturare e esternalizzare lo stato interno di un oggetto, per successivo restauro, senza violare l'incapsulamento.
- **Soluzione:**
  - Creare una classe Memento capace di memorizzare l'informazione interna dell'Originator.
  - L'Originator crea un Memento al bisogno e chiama setState() per copiare i propri attributi interni in Memento.
  - Originator chiama getState() in Memento per recuperare uno stato precedente.
  - Definire una classe Caretaker per mantenere il Memento.
- **Conseguenze**
  - Semplifica Originator che non deve mantenere lo stato precedente.
  - Occorre garantire che solo l'Originator possa accedere allo stato.
  - Può avere alti costi di memoria

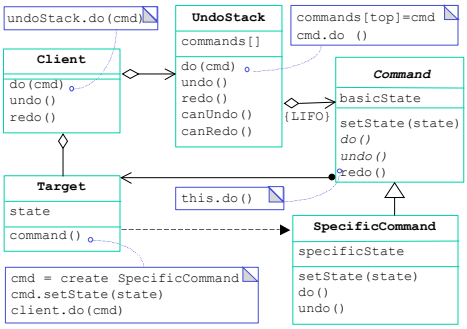
Memento Pattern



Basic Undo/Redo Pattern

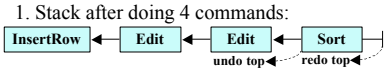
- **Problema:**
  - Fornire capacità di undo e redo per operazioni di UI che cambiano informazioni importanti.
- **Solution:**
  - Usare il pattern Command per incapsulare tutti i comandi disfattibili.
  - Incorporare il pattern Memento per permettere di memorizzare informazioni per disfare il comando.
  - Fondere l'Invoker del Command con il Caretaker del Memento per formare uno stack di undo/redo.
- **Conseguenze**
  - Disegno applicabile in molte situazioni.
  - Facilmente estendibile per aggiungere nuovi comandi disfattibili.
  - Può dover restringere i livelli di undo per risparmiare memoria.
  - Disegno più complesso.

Basic Undo/Redo Pattern

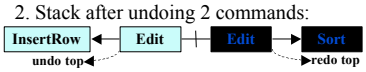


Undo / Redo Stack Operation

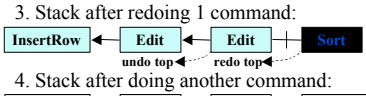
1. Stack after doing 4 commands:



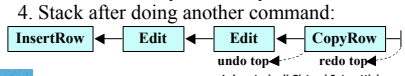
2. Stack after undoing 2 commands:



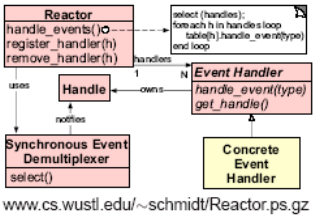
3. Stack after redoing 1 command:



4. Stack after doing another command:



The Reactor Pattern



www.cs.wustl.edu/~schmidt/Reactor.ps.gz

Intent

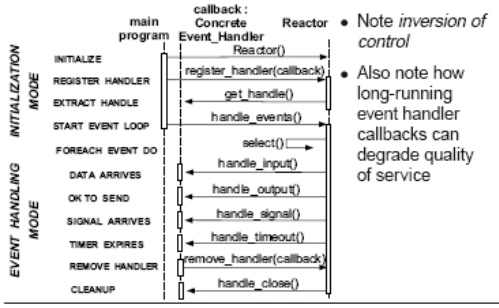
- *Decouples synchronous event demuxing & dispatching from event handling*

Forces Resolved

- Efficiently demux events *synchronously* within one thread
- Extending applications without changing demux infrastructure

Da Douglas C. Schmidt, **Introduction to Patterns and Frameworks**, UCLA Extension Courses

Collaboration in the Reactor Pattern



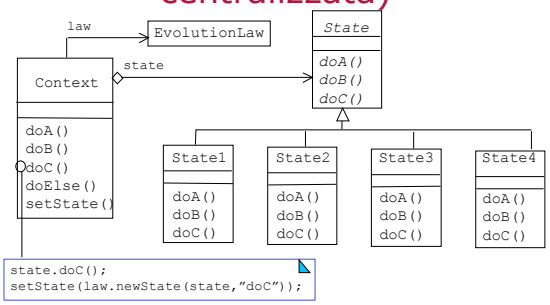
- Note *inversion of control*
- Also note how long-running event handler callbacks can degrade quality of service

Da Douglas C. Schmidt, **Introduction to Patterns and Frameworks**, UCLA Extension Courses

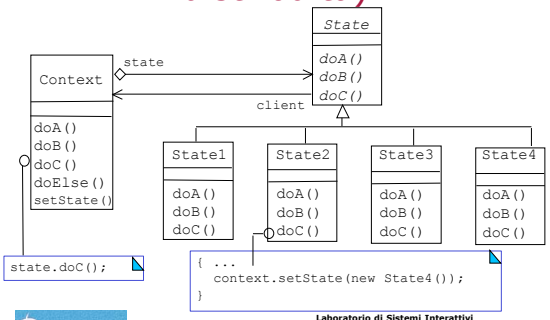
Pattern State

- **Problema:**
  - Permettere a un sistema di realizzare un insieme di comportamenti in maniera diversa in funzione del suo stato
- **Soluzione:**
  - Definire una classe astratta State la cui interfaccia dichiara i comportamenti da implementare
  - Definire una sottoclasse concreta per ogni stato da realizzare
  - Specificare i metodi che realizzano i comportamenti desiderati
  - Specificare le leggi per il cambiamento di stato
- **Conseguenze**
  - Elimina la necessità di specializzare la classe che può trovarsi nei diversi stati

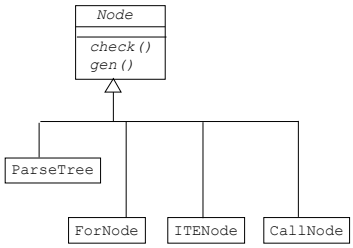
Pattern State (versione centralizzata)



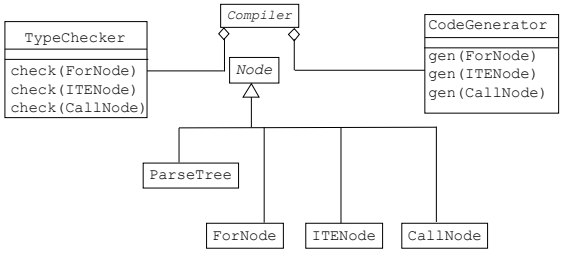
Pattern State (versione distribuita)



Visitor Pattern



Visitor Pattern



Visitor Pattern