Design Pattern Comportamentali

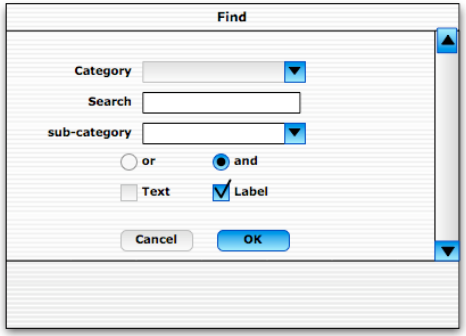
- Focalizzano sul controllo del flusso tra oggetti
- Descrivono le comunicazioni tra oggetti
- Aiutano a valutare le responsabilità assegnate agli oggetti
- Suggeriscono modi per incapsulare algoritmi dentro classi

Mediator

- Intento
  - Definisce un oggetto che incapsula come un gruppo di oggetti interagisce
  - Promuove lasco accoppiamento tra oggetti poiché essi non interagiscono direttamente
- Motivazione
  - La distribuzione delle responsabilità tra gli oggetti può risultare in molte connessioni tra oggetti
  - Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico
  - Diminuire le dipendenze di una classe e renderla più generale

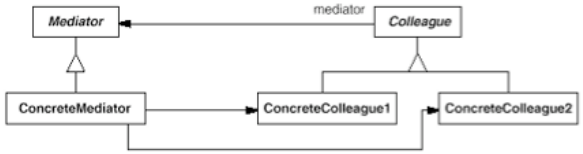
Mediator

- Per la finestra di ricerca mostrata
  - Ogni elemento visualizzato (testo, lista, bottone) è controllato da una corrispondente classe
  - Ciascuna classe deve comunicare il suo stato alle altre per far aggiornare la visualizzazione
  - Senza Mediator ciascuna classe dovrà invocare i metodi di tutte le altre classi



Mediator

- Soluzione
  - Isolare le comunicazioni (complesse) tra oggetti dipendenti creando una classe separata per esse
  - `Mediator`
    - Definisce un interfaccia tra oggetti che comunicano
  - `ConcreteMediator`
    - Implementa il comportamento cooperativo e coordina oggetti `Colleague`
  - `Colleague`
    - Ognuno conosce il `Mediator` e comunica con il `Mediator` quando avrebbe comunicato con gli altri `Colleague`



Mediator

- Conseguenze
  - La maggior parte della complessità che risulta nella gestione di dipendenze è spostata dagli oggetti cooperanti al `Mediator`. Questo rende gli oggetti più facili da implementare e mantenere
  - Le classi `Colleague` sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con codice che gestisce le dipendenze
  - Il codice del `Mediator` non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione

Mediator

```
// classe che implementa un ConcreteMediator
public class WindowFind implements Mediator {
  // istanze di vari Colleague
  private TextButton fine = new TextButton("OK");
  private TextButton canc = new TextButton("Cancel");
  private TextBox searcher = new TextBox();
  private ListBox categ = new ListBox(categs, 2);

  // metodo che attiva i controlli
  public void selectCateg(String s) {
    if (s.compareTo(categs[0]) == 0) {
       subcateg.activate();
       fine.deactivate();
       categ.show();
    }
    ...
  }
  public void init() {
    fine.deactivate();
    canc.activate();
    searcher.show();
    categ.deactivate();
  }
}
```

```
public interface Mediator {
  // metodo che i Colleague possono invocare
  public void selectCateg(String s);
}
```

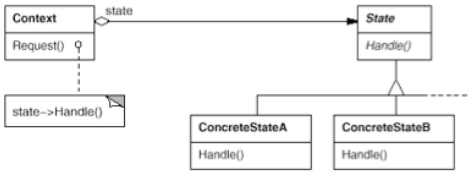
```
// classe che implementa un ConcreteColleague
public class TextButton extends Colleague {
  private String nome;
  private boolean active = false;

  public void activate() {
    active = true;
    show();
  }
  public void deactivate() {
    active = false;
    show();
  }
  public void show() {
    if (active) console.bold();
    else console.normal();
    console.print(x, y, nome);
  }
}
```


State

- Intento
  - Permette ad un oggetto di alterare il suo comportamento quando il suo stato cambia. L'oggetto appare come aver cambiato la sua classe
- Applicabilità
  - Il comportamento di un oggetto dipende dal suo stato e il comportamento cambia a run-time in dipendenza del suo stato
  - Le operazioni hanno condizioni che dipendono dallo stato

State

- Soluzione
  - Inserire ogni ramo condizionale in una classe separata
  - `Context`
    - Definisce l'interfaccia di interesse per i client
    - Mantiene un'istanza di un `ConcreteState` che definisce lo stato corrente
  - `State`
    - Definisce una interfaccia che incapsula il comportamento associato con un particolare stato
  - `ConcreteState`
    - Implementa il comportamento associato con uno stato



State

- Conseguenze
  - Inserisce il comportamento associato ad uno stato in una sola classe (`ConcreteState`)
  - Permette di incorporare la logica che gestisce il cambiamento di stato separatamente ed in una sola classe (`Context`), anziché (con istruzioni `if` o `switch`) sulla classe che implementa i comportamenti
  - Aiuta ad evitare stati inconsistenti poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
  - Incrementa il numero di classi

Anti-pattern

- Anti-pattern
  - Sono soluzioni (per il design) che si sono rivelate inefficaci
  - Sono descritti da
    - Nome
    - Problema: situazione ricorrente che ha conseguenze negative
    - Soluzione: come evitare o minimizzare il problema
- Anti-pattern God class (chiamato anche Blob)
  - Problema
    - Si ha una classe `Controller` di grandi dimensioni che usa tante classi di piccole dimensioni, le quali contengono solo dati
    - Il codice della classe `Controller` è difficile da comprendere
    - `Controller` invoca molti metodi di altre classi, `Controller` non contiene i dati su cui lavora
  - Soluzione
    - Refactoring: distribuire le responsabilità tra varie classi
    - Una classe dovrebbe contenere i dati che permettono ad essa di prendere delle decisioni

Excessive Dynamic Allocation

- Antipattern Excessive Dynamic Allocation
- Problema
  - Creazione e distruzione frequente di oggetti di una stessa classe
    - Decadimento delle prestazioni, se avviene su un grande numero di oggetti
    - Per la creazione: allocazione di memoria, inizializ. codice, inizializ. oggetto
    - Per la distruzione: esecuzione del garbage collector
- Soluzione
  - Cambiare il codice per eliminare molte esecuzioni di new
    - Ad es. se new è dentro un ciclo, potrebbe essere portato fuori
  - Riciclare oggetti anziché crearne di nuovi (gestione con object pool)
  - Eliminare la necessità di avere nuovi oggetti (usando ad es. il pattern Flyweight)
    - Individuare e condividere lo stato intrinseco immutabile (oggetto Flyweight) e separarlo dallo stato estrinseco (dipendente dal contesto)

Bug pattern

- Bug pattern
  - Sono correlazioni ricorrenti tra errori che vengono segnalati e bug che sono stati inseriti nel codice
  - Conoscerli aiuta a diagnosticare bug ed a correggerli più velocemente
  - Descritti da: Nome, Sintomo, Causa, Soluzione
- Bug pattern Rogue Tile
  - Sintomo
    - Il programma esegue come se il bug che è stato corretto fosse ancora presente
  - Causa
    - Esiste più di una copia dello stesso frammento di codice (è stato usato copy-and-paste), e solo una copia è stata corretta
  - Soluzione
    - Evitare il copy-and-paste, fattorizzare il codice comune