LEZIONE 10 - Design Patterns

Catia Trubiani
Gran Sasso Science Institute (GSSI), L'Aquila
catia.trubiani@gssi.infn.it

Indice

- Contesto: cos'è un design pattern?
- Elementi definiti in design patterns:
  - → Nome, Problema, Soluzione, Conseguenze
- Catalogo di design patterns:
  - → Creational Patterns
  - → Structural Patterns
  - → Behavioral Patterns
- Conclusioni e riferimenti bibliografici

2

Contesto

- I design patterns rappresentano soluzioni a problemi che sorgono durante lo sviluppo del software, infatti, si riferiscono a coppie (problema, soluzione) all'interno di un particolare contesto
- Descrivono i pattern di progettazione ricorrenti facilitando il riuso di architetture e modelli di design software
- I design patterns catturano la struttura statica, dinamica, e collaborativa tra le entità coinvolte nel design software

3

Origine di Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice"

Christopher Alexander, A Pattern Language, 1977,

Context: City Planning and Building architectures

4

Un po' di storia

1987: Cunningham e Beck usano le idee di Alexander per sviluppare un pattern language per Smalltalk

1990: Gang of Four (Gamma, Helm, Johnson e Vlissides) iniziano a creare un catalogo di design patterns

1991: Bruce Anderson presenta i primi design patterns alla Conferenza su Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)

1993: Kent Beck e Grady Booch organizzano il gruppo Hillside, che è stato il primo meeting in cui sono state discusse le challenges sui design patterns

1994: Prima Conferenza su Pattern Languages of Programs (PLoP)

1995: The Gang of Four pubblica un libro su "Design Patterns"

5

Elementi definiti in Design Patterns

I design patterns hanno 4 elementi essenziali:

- Nome
- Problema
- Soluzione
- Conseguenze

6

Design patterns - nome (1)

Il nome è un modo per descrivere:

- → un problema di design
- → le sue soluzioni
- → le sue conseguenze

Migliora il vocabolario di design

Abilita la possibilità di fare design ad un livello più alto di astrazione

Migliora la comunicazione

"The Hardest part of programming is coming up with good variable [function and type] names."

7

Design Patterns - problema (2)

Descrive quando applicare un pattern

Spiega il problema ed il suo contesto

Descrive problemi di design specifici e/o strutture ad oggetto

Contiene una lista di precondizioni che devono essere verificate prima che ha senso applicare il design pattern

8

Design Patterns - soluzione (3)

Descrive gli elementi che constituiscono il pattern:

- design
- relazioni
- responsabilità
- collaborazioni

Non descrive una specifica implementazione concreta

Si tratta di una descrizione astratta dei problem di design e come il pattern li “risolve”

9

Design Patterns - conseguenze (4)

Descrive i risultati e i trade-offs dovuti all’applicazione del design pattern

Serve per:

- valutare alternative di design
- comprendere i costi
- comprendere i benefici dovuti all’applicazione del pattern

Comprende la descrizione delle modifiche del pattern sul sistema software in termini di:

- flexibility
- extensibility
- portability

10

Cosa non sono i Design Patterns

- Design che possono essere codificati in classi e riutilizzati così come sono (come linked lists e hash tables)

- Design specifici di domini complessi (per un'intera applicazione o per un sottosistema)

I design patterns sono "descrizioni di oggetti e classi che comunicano e che vengono personalizzate per risolvere un problema di progettazione generale in un contesto particolare"

11

Come descrivere design patterns

- La notazione grafica non è generalmente sufficiente

- La parte di conoscenza critica è quella legata al riutilizzo di decisioni progettuali alternative e trade-offs che hanno portato alle decisioni

- Esempi concreti sono anche importanti

- La storia del perché, quando e come diventa la prerogativa per il contesto di utilizzo

12

Design patterns summary

Descrivono una struttura di design ricorrente

- Definiscono un vocabolario comune
- Sono astratti da design concreti
- Identificano classi, collaborazioni, e responsabilità
- Descrivono applicabilità, compromessi, e conseguenze

13

Categorizzazione dei termini

«Scope» è il dominio su cui si applica un pattern:

Class scope: relazioni tra le classi base e le loro sottoclassi (semantica statica)
Object scope: relazioni tra oggetti peer

Alcuni patterns si applicano ad entrambi gli ambiti

14

Tipi di Design Patterns

Creational patterns:

- Riguardano l'inizializzazione e la configurazione di classi e oggetti

Structural patterns:

- Riguardano il "decoupling" (disaccoppiamento) delle interfacce e l'implementazione di classi o oggetti
- Riguardano la composizione di classi o oggetti

Behavioral patterns:

- Riguardano le interazioni dinamiche tra classi o oggetti
- Come distribuire la responsabilità

15

Alcuni esempi di Design Patterns

	Creational	Structural	Behavioral
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)

16

Descrizione del «Decorator» pattern

Obiettivo:

- agganciare responsibilità aggiuntive ad un oggetto dinamicamente. I Decorators forniscono un'alternativa flessibile al subclassing per estendere le funzionalità
- conosciuto anche come "Wrapper"

Motivazione:

- Vogliamo aggiungere delle responsabilità ai singoli oggetti, non ad un'intera classe. Un toolkit per una interfaccia utente grafica, per esempio, dovrebbe permettere di aggiungere proprietà come bordi o comportamenti come lo scrolling a qualsiasi componente dell'interfaccia utente

17

Goal: aggiungere responsabilità a singoli oggetti

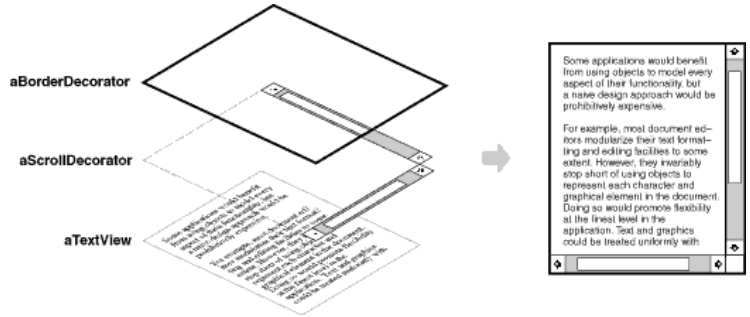
Un modo per aggiungere responsabilità è con l'ereditarietà. Ereditare un bordo da un'altra classe implica che verrà applicato un bordo attorno a ogni istanza della sottoclasse. Questo è inflessibile, però, perché la scelta di inserire un bordo è fatta staticamente. Un utente non può controllare come e quando decorare la componente con un bordo.

Un approccio più flessibile è quello di racchiudere il componente in un altro oggetto che aggiunge il bordo. L'oggetto che racchiude il bordo viene chiamato un decoratore. Il decoratore è conforme all'interfaccia del componente che decora quindi la sua presenza è trasparente agli utenti del componente. Il decoratore inoltra le richieste al componente e può eseguire azioni aggiuntive (come ad esempio disegnare un bordo) prima o dopo l'inoltro.

18

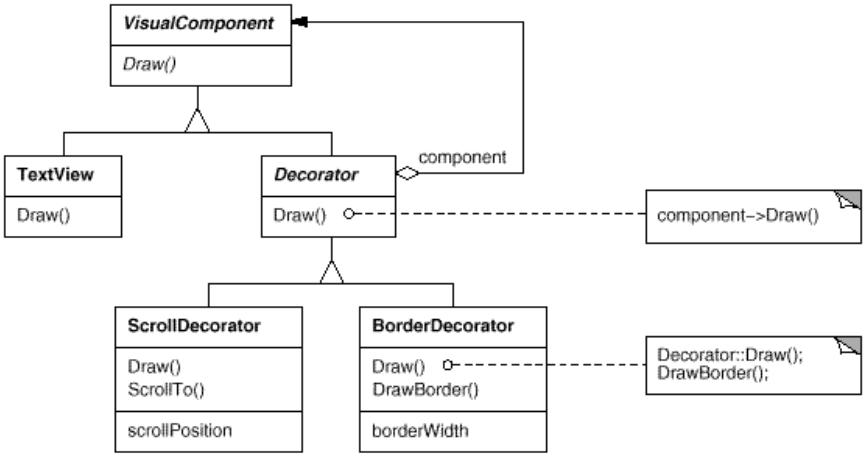
Perché usare il Decorator Pattern

Supponiamo di avere un oggetto TextView che visualizza il testo in una finestra. *TextView* non ha barre di scorrimento di default, perché potrebbe non averne bisogno. Se necessario, possiamo usare una **ScrollDecorator** per aggiungerle. Se vogliamo anche aggiungere un bordo nero spesso intorno alla TextView, possiamo usare un **BorderDecorator**.



19

Decorator pattern structure



20

Decorator pattern: applicabilità

Utilizzare il Decorator pattern per:

- aggiungere le responsabilità di singoli oggetti in modo dinamico e trasparente, cioè, senza influenzare gli altri oggetti.
- responsabilità che possono essere ritirate.
- quando l'estensione di sottoclassi è impraticabile. Talvolta un gran numero di estensioni indipendenti sono possibili e producono un'esplosione di sottoclassi per supportare qualsiasi combinazione. Oppure una classe può essere nascosta o non disponibile per l'ereditarietà.

21

Decorator pattern: partecipanti

Component (VisualComponent)

→ definisce l'interfaccia per gli oggetti che possono avere responsabilità a loro aggiunte in modo dinamico

ConcreteComponent (TextView)

→ definisce un oggetto a cui possono essere assegnate responsabilità aggiuntive

Decorator

→ mantiene un riferimento ad un oggetto di un componente e definisce un'interfaccia conforme all'interfaccia del componente

ConcreteDecorator (BorderDecorator, ScrollDecorator)

→ aggiunge responsibilità al componente

22

Decorator Pattern: conseguenze (1/3)

Più flessibilità rispetto all'ereditarietà statica.

- Il pattern Decorator fornisce un modo più flessibile per aggiungere responsabilità agli oggetti rispetto all'ereditarietà. Con i decorators, le responsabilità possono essere aggiunte e rimosse a runtime, semplicemente 'attaccandoli' e 'staccandoli'. Al contrario, l'ereditarietà richiede la creazione di una nuova classe per ogni responsabilità supplementare e questo dà luogo a molte classi e aumenta la complessità del sistema.

23

Decorator Pattern: conseguenze (2/3)

Evita classi con carichi elevati nella gerarchia.

- Il Decorator pattern offre un approccio «pay-as-you-go» nel processo di aggiunta delle responsabilità. Invece di cercare di aggiungere tutte le funzionalità prevedibili in un'unica classe (complessa), è possibile definire una semplice classe ed aggiungere funzionalità in modo incrementale con oggetti Decorator. Le funzionalità possono essere composte da pezzi semplici. Di conseguenza, un'applicazione non deve pagare per caratteristiche che non utilizza.

24

Decorator Pattern: conseguenze (3/3)

Un decorator e la sua componente non sono identici.

- Un componente decorato non è identico al componente stesso. Quindi non si dovrebbe fare affidamento su un oggetto quando si utilizzano i decoratori.

Tanti piccoli oggetti.

- Un design che utilizza un Decorator si traduce spesso in sistemi composti da tanti piccoli oggetti che sembrano tutti uguali. Gli oggetti si differenziano solo per il modo in cui sono interconnessi, non nella loro classe o nel valore delle loro variabili.

25

Decorator Pattern: sample code (1/4)

```
class VisualComponent {
  public: VisualComponent();
  virtual void Draw();
  virtual void Resize();
  // ...
}
```

26

Decorator Pattern: sample code (2/4)

```
class Decorator : public VisualComponent {
   public: Decorator(VisualComponent*);
   virtual void Draw();
   virtual void Resize();
   // ...
   private: VisualComponent* _component;

   void Decorator::Draw () {
         _component->Draw();
   }
   void Decorator::Resize () {
         _component->Resize();
   }
}
```

27

Decorator Pattern: sample code (3/4)

```
class BorderDecorator : public Decorator {
   public: BorderDecorator(VisualComponent*, int borderWidth);

   virtual void Draw();

   private: void DrawBorder(int);
   private: int _width;

   void BorderDecorator::Draw () {
         Decorator::Draw();
         DrawBorder(_width);
   }
}
```

28

Decorator Pattern: sample code (4/4)

```
void Window::SetContents (VisualComponent* contents){
        // ...
}

Window* window = new Window;
TextView* textView = new TextView;

window->SetContents(textView);

window->SetContents(
        new BorderDecorator(
                new ScrollDecorator(textView), 1);
```

29

Conclusioni

Come usare un Design Pattern? Un Pattern deve:

- Risolvere un problema ed essere utile
- Avere un contesto e descrivere dove la soluzione può essere usata
- Ricorrere in situazioni rilevanti
- Fornire comprensione sufficiente per 'ritagliare' la soluzione corrispondente
- Avere un nome ed essere riferito in modo consistente

30

Design Patterns - vantaggi

- I design patterns consentono il riutilizzo su larga scala di architetture software ed inoltre aiutano in fase di documentazione dei sistemi software

- I design patterns catturano esplicitamente la conoscenza di esperti e i trade-offs di design, e la rendono disponibile

- I design patterns contribuiscono a migliorare la comunicazione tra gli sviluppatori

- I nomi di design patterns costituiscono un vocabolario comune

- I design patterns contribuiscono ad alleviare la transizione alla tecnologia OO

31

Design Patterns - svantaggi

- I design patterns non consentono il riutilizzo diretto del codice
- I design patterns sono ingannevolmente semplici
- Team di sviluppo possono risentire del sovraccarico dovuto ai patterns
- I design patterns sono convalidati dall'esperienza e dalla discussione piuttosto che da test automatizzati
- L'integrazione di design patterns in un processo di sviluppo del software è un'attività umana che richiede un grande effort

32

Suggerimenti per l'utilizzo

- Non considerare tutto come un pattern
  - → Al contrario, conviene sviluppare patterns strategici di dominio e riutilizzare patterns tecnici esistenti
- Fornire ricompense per lo sviluppo di patterns
- Coinvolgere direttamente autori di patterns con gli sviluppatori di applicazioni e gli esperti del dominio
- Documentare chiaramente quando i patterns si applicano e quando non si applicano
- Gestire con attenzione le aspettative

33

Ulteriori letture e riferimenti bibliografici

Altri Design Patterns:

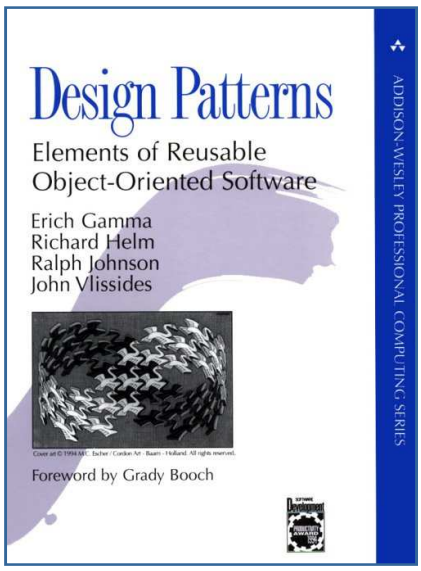
- A Creational Pattern: "Abstract Factory"
- A Behavioral Pattern: "Observer"

Reference principale:

- E.Gamma et al. "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.

Altre references:

- Sommerville, "Software Engineering", section 7.2



34

Questions?

catia.trubiani@gssi.infn.it

35