12. Principi di Progettazione Software, Metodologie Agile e Design Patterns

Questo capitolo finale integra i concetti di progettazione del software, le metodologie di sviluppo agili e l'applicazione dei design pattern, essenziali per la costruzione di sistemi software robusti, flessibili e manutenibili.

6.1. Fasi del Processo di Sviluppo Software

Lo sviluppo di un sistema software è un processo complesso che si articola in diverse fasi interconnesse, spesso iterate in cicli di sviluppo moderni (come quelli agili).

6.1.1. Panoramica delle Fasi

Le fasi principali del processo di sviluppo software includono:

1. **Analisi**: In questa fase si definiscono i requisiti del sistema, comprendendo cosa il software deve fare per soddisfare le esigenze degli utenti e degli stakeholder. Si distinguono i requisiti funzionali (cosa fa il sistema) e non funzionali (come lo fa, es. prestazioni, sicurezza).
2. **Design (Progettazione)**: Si traduce la fase di analisi in una struttura del software. Si decide come il sistema sarà costruito, definendo l'architettura, i moduli, le classi, le interfacce e le relazioni tra essi. Questa fase si divide spesso in:
    - **Architettura**: Definizione delle componenti di alto livello e delle loro interazioni.
    - **Design di Dettaglio**: Definizione delle classi, dei metodi e delle strutture dati specifiche.
3. **Implementazione**: Si traduce il design in codice eseguibile, scrivendo il software vero e proprio.
4. **Collaudo (Testing)**: Si verifica che il software funzioni correttamente e soddisfi i requisiti. Include diverse tipologie di test (unità, integrazione, sistema, accettazione).
5. **Deployment (Messa in Produzione)**: Il software viene rilasciato e reso disponibile agli utenti finali.
6. **Manutenzione**: Attività post-rilascio che include la correzione di bug, l'aggiunta di nuove funzionalità e l'adattamento a nuovi ambienti.

6.1.2. Problem Space vs. Solution Space

Nel contesto della progettazione, è utile distinguere tra:

- **Problem Space (Dominio/Logica Business)**: Riguarda il problema che il software intende risolvere. Descrive le esigenze, i processi e le entità del **mondo reale** che il sistema deve modellare.
- **Solution Space (Scelte Realizzative)**: Riguarda il modo in cui il **software** viene costruito per risolvere il problema. Include le decisioni tecnologiche, le strutture dati, gli algoritmi e i pattern di progettazione utilizzati.

La progettazione efficace implica un'astrazione dal problem space per creare soluzioni nel solution space che siano flessibili e manutenibili. Il concetto di "livello di astrazione" è cruciale per gestire la complessità.

6.2. Concetti di Qualità Interna del Software

La qualità del software non si limita alla sua funzionalità esterna, ma include anche la sua "qualità interna", ovvero quanto è ben costruito, leggibile e flessibile il codice.

6.2.1. Modularità, Dipendenza, Accoppiamento, Coesione

Questi sono principi fondamentali per una buona progettazione del software:

- **Modularità**: La capacità di un sistema di essere **scomposto** in componenti discreti e **indipendenti (moduli)** che possono essere sviluppati, testati e mantenuti separatamente. Un buon design promuove moduli con **responsabilità ben definite**.
- **Dipendenza**: La relazione tra due moduli in cui un modulo richiede l'altro per funzionare. Idealmente, le dipendenze dovrebbero essere **minime e unidirezionali**.
- **Accoppiamento (Coupling)**: Misura la forza delle dipendenze tra i moduli. Un accoppiamento **basso** è desiderabile, poiché indica che i moduli sono **meno interdipendenti**, rendendoli più facili da modificare e riutilizzare. Un

accoppiamento alto (es. due classi che dipendono fortemente l'una dall'altra) rende il sistema più rigido e fragile.

- **Coesione (Cohesion)**: Misura quanto gli **elementi** all'interno di un modulo sono **correlati** tra loro e contribuiscono a un **singolo scopo ben definito**. Una coesione **alta** è desiderabile, poiché indica che un modulo è focalizzato su una singola responsabilità, rendendolo più comprensibile e manutenibile.

6.2.2. Riuso del Codice

Il riuso è un obiettivo chiave nella progettazione software e può essere ottenuto principalmente in due modi:

- **Composizione (utilizzo di altri oggetti)**: Un oggetto è ottenuto combinando istanze di altre classi. Questo promuove un accoppiamento più debole e una maggiore flessibilità rispetto all'ereditarietà.
- **Estensione (ereditarietà)**: Una nuova classe è ottenuta riusando il codice di una classe pre-esistente. Questo crea una relazione "è un tipo di" (is-a) e promuove il riuso del comportamento. Tuttavia, l'ereditarietà può portare a un accoppiamento forte e problemi di gerarchia se non usata con attenzione.

6.3. Agile Software Development

Le metodologie agili sono un insieme di principi e **pratiche per lo sviluppo software** che promuovono la consegna iterativa, la collaborazione del team e la risposta al cambiamento.

6.3.1. Cos'è Agile?

Agile è un **approccio iterativo e incrementale** allo sviluppo software. Invece di un lungo ciclo di sviluppo "a cascata" (waterfall), Agile **suddivide il lavoro in piccole iterazioni (sprint)**, ognuna delle quali produce un incremento di software funzionante. L'obiettivo è massimizzare il valore per il cliente attraverso la consegna continua e l'adattamento ai requisiti che evolvono.

6.3.2. Il Manifesto Agile

Il Manifesto per lo Sviluppo Agile del Software è stato scritto nel 2001 e definisce i valori e i principi fondamentali delle metodologie agili.

Valori del Manifesto Agile:

- **Individui e interazioni** più che processi e strumenti.
- **Software funzionante** più che documentazione esaustiva.
- **Collaborazione del cliente** più che negoziazione dei contratti.
- **Rispondere al cambiamento** più che seguire un piano.

Ciò significa che, sebbene ci sia valore negli elementi a destra, si valorizzano maggiormente gli elementi a sinistra.

6.3.3. Principi Agile

Il Manifesto Agile è supportato da **dodici principi**, tra cui:

- La nostra massima priorità è soddisfare il cliente attraverso la consegna rapida e continua di software di valore.
- Accogliere i requisiti che cambiano, anche in fase avanzata di sviluppo. I processi agili sfruttano il cambiamento per il vantaggio competitivo del cliente.
- Consegnare frequentemente software funzionante, da un paio di settimane a un paio di mesi, con una preferenza per il periodo più breve.
- Le persone di business e gli sviluppatori devono lavorare insieme quotidianamente durante tutto il progetto.
- Costruire progetti attorno a individui motivati. Dare loro l'ambiente e il supporto di cui hanno bisogno e fidarsi di loro per portare a termine il lavoro.
- Il metodo più efficiente ed efficace per comunicare informazioni a e all'interno di un team di sviluppo è la conversazione faccia a faccia.
- Il software funzionante è la misura principale del progresso.
- I processi agili promuovono lo sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere un ritmo costante a tempo indeterminato.
- L'attenzione continua all'eccellenza tecnica e al buon design migliora l'agilità.

- La semplicità – l'arte di massimizzare la quantità di lavoro non fatto – è essenziale.
- Le migliori architetture, requisiti e design emergono da team auto-organizzati.
- A intervalli regolari, il team riflette su come diventare più efficace, quindi sintonizza e adatta il suo comportamento di conseguenza.
- *Provenienza*: Web (Agile Manifesto Principles)

6.3.4. Metodologie Agile Comuni

- **Scrum**: Un framework leggero per la gestione di progetti complessi, basato su sprint (iterazioni a tempo fisso), ruoli (Product Owner, Scrum Master, Development Team) ed eventi (Daily Scrum, Sprint Planning, Sprint Review, Sprint Retrospective).
- **Kanban**: Un metodo per visualizzare il lavoro, limitare il lavoro in corso (WIP) e massimizzare l'efficienza del flusso.
- **Extreme Programming (XP)**: Un insieme di pratiche ingegneristiche (es. programmazione a coppie, Test-Driven Development, integrazione continua) che mirano a migliorare la qualità del software e la capacità di rispondere ai requisiti che cambiano.
- *Provenienza*: Web (integrazione con concetti dei PDF)

6.4. Introduzione ai Design Pattern

I Design Pattern sono **soluzioni riutilizzabili** a problemi comuni che si presentano nella progettazione del software. Non sono soluzioni dirette, ma "ricette" o schemi che possono essere adattati a contesti specifici.

6.4.1. Cos'è un Design Pattern?

Un design pattern è una descrizione di una soluzione a un problema ricorrente che si verifica in un contesto specifico. Non è una classe o una libreria specifica, ma una **descrizione di come risolvere** un problema di progettazione. I pattern sono stati formalizzati dal "Gang of Four" (GoF) nel loro libro "Design Patterns: Elements of Reusable Object-Oriented Software".

Elementi di un Design Pattern:

- **Nome del Pattern**: Un nome descrittivo e riconoscibile.
- **Problema**: Descrizione del problema che il pattern risolve.
- **Soluzione**: Descrizione astratta della soluzione, includendo gli elementi di design, le loro relazioni e le responsabilità.
- **Conseguenze**: I pro e i contro dell'applicazione del pattern, inclusi compromessi e impatti sulla flessibilità, riusabilità, prestazioni, ecc.

6.4.2. Classificazione dei Design Pattern (GoF)

I pattern GoF sono classificati in tre categorie principali:

1. **Creazionali**: Riguardano la **creazione di oggetti**, fornendo meccanismi per creare oggetti in modo flessibile e disaccoppiato dal codice client.
   - Esempi: Factory Method, Abstract Factory, Singleton, Builder, Prototype.
2. **Strutturali**: Riguardano la **composizione di classi e oggetti** per formare strutture più grandi.
   - Esempi: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
3. **Comportamentali**: Riguardano l'**interazione e la distribuzione delle responsabilità** tra gli oggetti.
   - Esempi: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

6.5. Esempi di Design Pattern

Abbiamo già incontrato alcuni design pattern nei capitoli precedenti. Qui li riassumiamo e ne introduciamo un altro fondamentale.

6.5.1. Pattern Strategy (Comportamentale)

- **Problema**: Si ha un algoritmo che può variare o che può essere sostituito in base al contesto. Si vuole definire una famiglia di algoritmi, incapsularli e renderli intercambiabili.

- **Soluzione**: Definire un'**interfaccia comune** per la famiglia di algoritmi (la "strategia"). Ogni algoritmo concreto implementa questa interfaccia. La classe che utilizza l'algoritmo (il "contesto") mantiene un riferimento all'interfaccia della strategia e delega l'esecuzione dell'algoritmo all'oggetto strategia.
- **Vantaggi**: Permette di cambiare il comportamento di un oggetto a run-time, elimina le istruzioni condizionali complesse, e promuove il principio "Open/Closed" (aperto all'estensione, chiuso alla modifica).

```
interface GeneircRespository {}

class UsersRepository implements GenericRepository {}
```

Esempio: Strategie di Calcolo Tasse
Invece di avere un `if-else` per ogni tipo di calcolo tasse, si definisce un'interfaccia `TaxStrategy` e diverse implementazioni (es. `ItalianTaxStrategy`, `USTaxStrategy`). La classe `Invoice` (contesto) usa un oggetto `TaxStrategy` per calcolare le tasse.

6.5.2. Pattern Factory Method (Creazionale)

- **Problema**: Una classe non può prevedere la classe degli oggetti che deve creare. Si vuole delegare la responsabilità di istanziare gli oggetti a sottoclassi.
- **Soluzione**: Definire un'interfaccia per la creazione di un oggetto (il "prodotto"), ma lasciare alle sottoclassi la decisione su quale classe istanziare. La classe base (il "creatore") dichiara un **"metodo factory" astratto** che le sottoclassi devono implementare per restituire un'istanza del prodotto concreto.
- **Vantaggi**: Disaccoppia la creazione di oggetti dal codice client, promuove la flessibilità e l'estendibilità.

Esempio: Creazione di Documenti
Una classe `Application` può avere un metodo `createDocument()` che restituisce un `Document`. Le sottoclassi di `Application` (es. `DrawingApplication`, `TextApplication`) implementano `createDocument()` per restituire rispettivamente un `DrawingDocument` o un `TextDocument`.

```
// Classe base astratta per i Documenti
abstract class Document {
    private String name;
}

// Implementazione specifica per un Documento di Disegno
class DrawingDocument extends Document {
    public DrawingDocument(String name) {
        super(name);
    }
}

// Implementazione specifica per un Documento di Testo
class TextDocument extends Document {
    public TextDocument(String name) {
        super(name);
    }
}

// Classe base astratta per l'Applicazione
abstract class Application {
    public abstract Document createDocument(String name);
}

// Implementazione specifica per un'Applicazione di Disegno
class DrawingApplication extends Application {
    @Override
    public Document createDocument(String name) {
        return new DrawingDocument(name);
    }
}
```

```
// Implementazione specifica per un'Applicazione di Testo
class TextApplication extends Application {
    @Override
    public Document createDocument(String name) {
        return new TextDocument(name);
    }
}

// Classe di esempio per l'utilizzo
public class Main {
    public static void main(String[] args) {
        Application drawingApp = new DrawingApplication();
        Application textApp = new TextApplication();

        drawingApp.newDocument("My Drawing");
        textApp.newDocument("My Text");
    }
}
```

6.5.3. Pattern Decorator (Strutturale)

- **Problema**: Si vuole aggiungere dinamicamente nuove responsabilità a un oggetto, senza modificare la sua struttura. Si vuole evitare l'esplosione di sottoclassi per ogni combinazione di funzionalità.
- **Soluzione**: Definire un'interfaccia comune per il componente e il decoratore. Il decoratore incapsula un'istanza del componente (o di un altro decoratore) e aggiunge funzionalità prima o dopo aver delegato la chiamata al componente incapsulato.
- **Vantaggi**: Flessibilità nell'aggiunta di responsabilità, evita l'ereditarietà multipla, promuove il principio "Open/Closed".

Esempio: I/O in Java
Come discusso nel Capitolo 4, le classi di I/O in Java (es. `InputStream`, `OutputStream`) sono un esempio classico del pattern Decorator. `FileInputStream` è un componente base, mentre `BufferedInputStream` è un decoratore che aggiunge il buffering.

6.5.4. Pattern Template Method (Comportamentale)

- **Problema**: Si vuole definire lo scheletro di un algoritmo in un'operazione, delegando alcuni passi alle sottoclassi.
- **Soluzione**: Definire un metodo (il "template method", spesso `final`) in una classe astratta che contiene la struttura dell'algoritmo. Questo metodo invoca uno o più metodi astratti (o "hook methods") che devono essere implementati dalle sottoclassi.
- **Vantaggi**: Permette di riutilizzare la struttura comune di un algoritmo, garantendo che i passi specifici siano implementati correttamente dalle sottoclassi.

Esempio: `BankAccount` (dal Capitolo 2)
La classe astratta `BankAccount` ha un `withdraw()` (template method) che delega il calcolo della commissione (`operationFee()`) a un metodo astratto. Le sottoclassi (es. `BankAccountWithConstantFee`) forniscono l'implementazione specifica di `operationFee()`.

```
public abstract class BankAccount {

    protected double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public double withdraw(double amount) {
        double fee = operationFee(amount);
        if (balance >= amount + fee) {
            balance -= (amount + fee);
```

```
            return amount;
        } else {
            return 0; // Or throw an exception - insufficient funds
        }
    }

    protected abstract double operationFee(double amount);

    public double getBalance() {
        return balance;
    }
}

public class BankAccountWithConstantFee extends BankAccount {

    private final double constantFee;

    public BankAccountWithConstantFee(double initialBalance, double constantFee) {
        super(initialBalance);
        this.constantFee = constantFee;
    }

    @Override
    protected double operationFee(double amount) {
        return constantFee;
    }
}

public class BankAccountWithPercentageFee extends BankAccount {

    private final double percentageFee;

    public BankAccountWithPercentageFee(double initialBalance, double percentageFee) {
        super(initialBalance);
        this.percentageFee = percentageFee;
    }

    @Override
    protected double operationFee(double amount) {
        return amount * percentageFee;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccountWithConstantFee account1 = new BankAccountWithConstantFee(100, 5);
        System.out.println("Withdraw 20 from account1: " + account1.withdraw(20)); // Output: 20.0
        System.out.println("Account1 balance: " + account1.getBalance()); // Output: 75.0

        BankAccountWithPercentageFee account2 = new BankAccountWithPercentageFee(100, 0.1);
        System.out.println("Withdraw 20 from account2: " + account2.withdraw(20)); // Output: 20.0
        System.out.println("Account2 balance: " + account2.getBalance()); // Output: 78.0
    }
}
```