11. Riuso del software e CBSE

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Sommario

1. Riuso
2. Approcci
3. CBSE
   - Componenti e modelli a componenti
   - Il processo di sviluppo di sistemi a componenti

Sommario

1. Riuso
2. Approcci
3. CBSE
   - Componenti e modelli a componenti
   - Il processo di sviluppo di sistemi a componenti

Riuso - generalità

Riuso tipico approccio ingegneristico alla costruzione di sistemi.
Sistemi risultano da integrazione di sottosistemi spesso non “banali”.

Motivazioni e benefici legati al riuso:

- ridotti costi di sviluppo,
- sviluppo da parte di specialisti,
- ridotti tempi di rilascio,
- aumentata qualità del software
- ridotto rischio del processo,

Riuso e tipiche problematiche

Problematiche tipiche nel riuso del software:

- Incremento nei costi di gestione del sistema
- Mancanza di strumenti di supporto al riuso
- Sindome del “NIH”
- Gestione di libreria di componenti
- Integrazione ed adattamento di componenti

Riuso - generalità

Riuso può interessare manufatti ma anche riuso a livello concettuale

Riuso di manufatti:

- Riuso di intere applicazioni
- Riuso di componenti e sottosistemi
- Riuso di oggetti e librerie

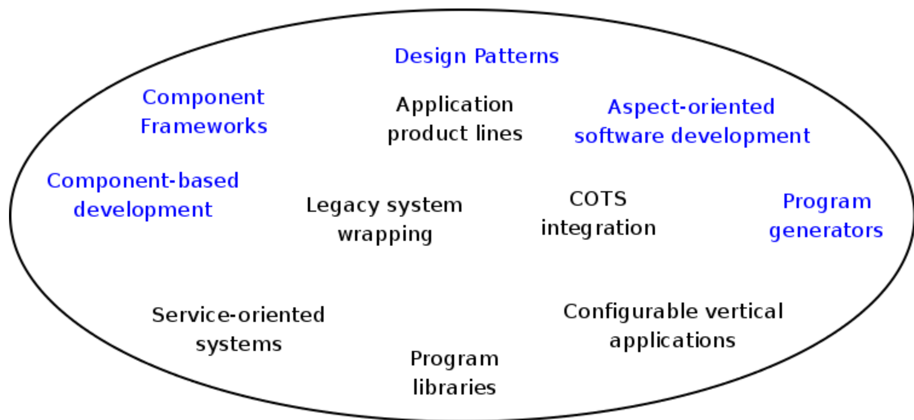
Sommario

1 Riuso

2 Approcci

3 CBSE
- Componenti e modelli a componenti
- Il processo di sviluppo di sistemi a componenti

Riuso - una panoramica



Quale approccio?

Fattori che possono guidare nella scelta di un approccio piuttosto che un'altro:

- Tempistiche dello sviluppo
- Attesa longevità dell'applicazione
- Conoscenze e capacità del team di sviluppo
- Caratteristiche di criticità del software
- Dominio applicativo
- Piattaforma di esecuzione

Design Patterns

Definizione: un pattern descrive un problema che ricorre spesso e propone una possibile soluzione in termini di organizzazione di classi/oggetti che generalmente si è rilevata efficace.

Design Pattern sono caratterizzati da quattro elementi principali:

- Nome - riferimento mnemonico che permette di aumentare il vocabolario dei termini tecnici e ci permette di identificare il problema e la soluzione in una o due parole
- Problema - descrizione del problema e del contesto a cui il pattern intende fornire una soluzione
- Soluzione - descrive gli elementi fondamentali che costituiscono la soluzione e le relazioni che intercorrono tra questi
- Conseguenze - specifica le possibili conseguenze che l'applicazione della soluzione proposta può comportare.

Design Patterns e documentazione

Una volta ben definito e generalmente accettato da una communità di sviluppatori il pattern diventa anche un'ottimo strumento per documentare il software.

Esistono esempi di applicazioni interamente descritte attraverso l'uso di design pattern (e.g. JUnit)

Riferimento:
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Pattern: elements of reusable Object Oriented software
Addison Wesley

Descrizione dei Design Pattern secondo la *GoF*

La definizione di collezioni di pattern in uno specifico dominio applicativo sono il primo passo per poter stabilire un contesto di riuso

Nel libro della GoF i pattern sono descritti attraverso i seguenti punti:

- Nome
- Intento
- Aka (Also known as)
- Motivazioni
- Applicabilità
- Struttura
- Partecipanti
- Collaborazioni
- Conseguenze
- Implementazione
- Codice sorgente di esempio
- Usi noti
- Patterns correlati

Classificazione dei Pattern

GoF ha definito 23 pattern generali e li ha classificati in base a due criteri principali:

- A cosa si riferiscono
  - Oggetti
  - Classi
- Cosa fa il pattern
  - Creazionali
  - Strutturali
  - Comportamentali

Certamente è possibile classificarli secondo altri criteri

GoF classificazione

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	***Adapter (class)***	Interpreter Template Method
	Object	***Abstract Factory*** Builder Prototype ***Singleton***	Adapter (object) Bridge Composite Decorator Façade Flyweight ***Proxy***	Chain of Responsibility Command Iterator Mediator Memento ***Observer*** State ***Strategy*** Visitor

Principi generali di buona programmazione OO

Ereditarietà definisce una classe in termini di un'altra. Principalmente meccanismo di riuso del codice di tipo white-box. Il programmatore conosce il codice.

Composizione di oggetti consiste nell'assemblare differenti oggetti al fine di ottenere il comportamente desiderato. Composizione rihiede definizione di precise interfacce.

- Programmare riferendosi alle interfacce e non alle implementazioni
- Favorire la composizione di oggetti rispetto all'ereditarietà

DP creazionali

Astraggono il meccanismo di generazione di istanze di una classe con l'obiettivo di rendere il sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati. La creazione degli oggetti viene tipicamente gestita da apposite "strutture".

Semplificano la composizione di classi e la favoriscono rispetto all'uso dell'ereditarietà.

I pattern creazionali forniscono molta flessibilità nel cosa, il come, il chi ed il quando della creazione di oggetti

Saranno considerati due pattern di questo tipo:

- Abstract Factory
- Singleton

Singleton

Scopo: Fare in modo che a run-time esista al più una sola istanza di una classe e fornire un punto globale di accesso a tale oggetto
Motivazione: É spesso importante avere una sola istanza di una classe a run-time. Si consideri ad esempio il caso di file system manager o del window manager di un sistema.
Partecipanti: la classe singleton
Conseguenze: controllo degli accessi all'istanza, spazio dei nomi ridotto, possibilità di estensione ad aver un numero n di istanze, facile manutenzione

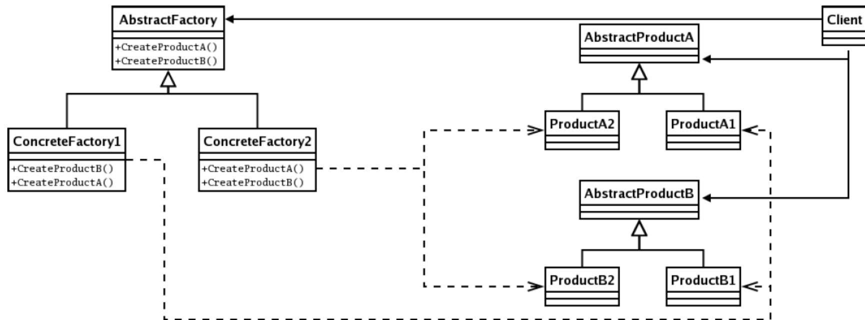
Singleton - Codice di esempio

```
public class Singleton {
  protected static Singleton instance = null;
  private static int counter = 0;

  ... // objects attributes and methods

  protected Singleton() { ... }
  public static Singleton Instance() {
    if (counter == 0) {
    instance = new Singleton(); counter++;
    }
    return instance;
  }
}
```

Abstract Factory

Scopo: fornire un interfaccia per permettere la creazione di famiglie di oggetti correlati o dipendenti, senza dover esplicitamente specificare la classe di appartenenza
Motivazione: Spesso un sistema deve essere indipendente da come gli elementi che lo compongono sono creati, composti e rappresentati. In altri casoi un sistema deve essere configurato in famiglie di prodotto.

Abstract Factory - Diagramma



Abstract Factory - Codice di esempio

```
public interface MazeFactory {
  Maze MakeMaze();
  Wall MakeWall();
  Door MakeDoor();
  Room MakeRoom();
}

public class MazeGame(MazeFactory mf) {
  Maze aMaze = mf.MakeMaze(); Room r1 = mf.MakeRoom(1);
  Room r2 = mf.MakeRoom(2); Door aDoor = mf.MakeDoor(r1,r2);
  aMaze.AddRoom(1); aMaze.addRoom(2);
  r1.SetSide(N,mf.MakeWall()); r1.SetSide(S,mf.MakeWall());
  r1.SetSide(E,mf.MakeWall()); r1.SetSide(O,aDoor);
  r2.SetSide(N,mf.MakeWall()); r2.SetSide(S,mf.MakeWall());
  r2.SetSide(E,aDoor); r2.SetSide(O,mf.MakeWall());
  return aMaze;
}
```

Abstract Factory

Conseguenze: il pattern permette di isolare i punti di creazione degli oggetti di una classe. La Factory incapsula tutti i meccanismi di creazione. Le classi concrete si trovano specificate soltanto all'interno della factory il resto si affida alla definizione delle interfacce.

Pattern Strutturali

Come comporre oggetti o classi per creare strutture più complesse

Proxy

Proxy

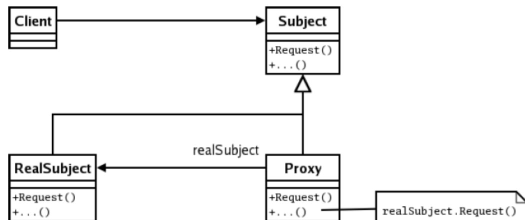
Scopo: fornire un surrogato o un segnaposto per un altro oggetto al fine di controllare l'accesso all'oggetto stesso. Allo stesso tempo può essere utile per rimandare la creazione di un oggetto ad un momento successivo per ridurre uso di risorse.

Motivazione: motivo può essere differire il costo di creazione ed inizializzazione dell'oggetto ad un momento successivo

Applicabilità:

- remote proxy
- virtual proxy
- protection proxy
- smart reference

Proxy - Codice e diagramma di esempio




```
public interface Graphic {...}

public class Image implements Gaphic {
  Image(FileInputStream fis) {...}; ...
}

public class ImageProxy implements Graphic {
  String fileName;
  public void Draw(at) { Image i = new Image(fis); i.Draw(at); }
}
```

Pattern Comportamentali

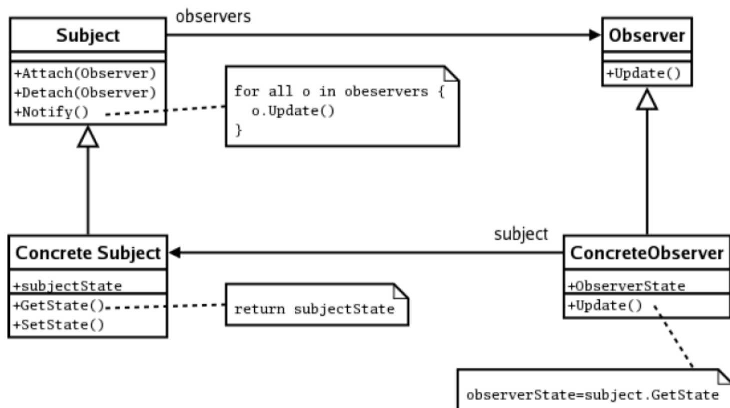
I pattern comportamentali hanno a che fare con gli algoritmi e gli assegnamenti di responsabilità tra gli oggetti. Caratterizzano sistemi di controllo complessi.
Observer

Observer

Scopo: Definisce una relazione uno-a-molti tra oggetti cosicché quando l'oggetto (uno) cambia stato tutti gli altri oggetti (molti) coinvolti vengano informati del cambiamento
Motivazione: c'è bisogno di mantenere consistenza tra gli oggetti correlati
Applicabilità: quando una modifica ad un oggetto richiede modifiche ad altri oggetti e non è noto a priori quali saranno tali oggetti
Partecipanti:

- Subject: conosce i suoi "osservatori" e fornisce meccanismi per il collegamento e lo scollegamento degli osservatori
- Observer: fornisce un'interfaccia per la notifica
- ConcreteSubject: contiene lo stato ed invia le notifiche
- ConcreteObserver: gestisce il riferimento al soggetto e mantiene l'informazione sullo stato dell'oggetto osservato.

Observer - Diagramma



Riuso basato su generatori automatici di codice

Il codice dell'applicazione viene derivato direttamente da specifici tool in cui è stata inserita la conoscenza relativa alla struttura necessaria delle applicazioni rivolte a risolvere la specifica esigenza.

Applicabile in ambiti applicativi in cui si siano manifestate precise organizzazione del software. Tali organizzazioni possono essere sfruttate al fine di generare automaticamente il codice a partire da input caratterizzante l'applicazione per lo specifico contesto.

E.g. Sviluppo di compilatori - Yacc, JavaCC
oppure tool per la generazione automatica di codice da diagrammi

Generative Programming

Generative Programming applica le idee del riuso in un ambito a componenti.

Aspect oriented programming - concetto di "*cross cutting concerns*". Il sistema dovrebbe essere progettato in modo tale che ogni elemento della struttura del sistema faccia una ed una sola cosa.

Il caso del monitoring

join point ed il weaving

Sommario

1. Riuso
2. Approcci
3. CBSE
   - Componenti e modelli a componenti
   - Il processo di sviluppo di sistemi a componenti

CBSE - generalità

Emerge alla fine degli anni 90. Riuso a partire da oggetti aveva fallito. Molte promesse al momento non completamente matenute (e.g. previsto mercato dei componenti)

Elementi e principi base di CBSE:

- Componenti e interfacce fornite
- Modello dei componenti e standards
- Middleware
- Processo di sviluppo a componenti

Sviluppo a componenti - nuove problematiche

Lo sviluppo a componenti introduce nuove problematiche che hanno richiesto e stanno richiedendo soluzioni al mondo dell'industria/ricerca:

- Component Trustworthiness
- Certificazione dei componenti
- Proprietà e predicibilità
- Aggiustamento dei requisiti

Componenti software

definizione ...

Non è ancora emersa una definizione “standard”. Tipicamente viene riportata quella di Szyperski [1998]:

Un componente software è un’unita di composizione con interfacce specificate contrattualmente e con definizione di precise dipendenze dal contesto. Un componente software può essere impiegato indipendentement ed è soggetto a composizione da parte di terzi.

Altre definizioni pongono l’accento sulla definizione di un preciso modello di componenti e sulla definizione di precisi meccanismi di composizione.

Componenti - in definitiva

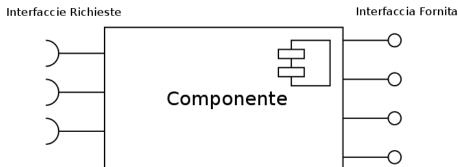
Proprietà salienti di un componente sono:

- Esistenza di “standard”
- Indipendenza del componente
- Creato per essere composto - interfacce publicamente accessibili
- Direttamente impiegabile (*Deployable* - formato binario
- Esistenza di precisa documentazione

Componenti ed interfacce

Per un componente sono definiti due tipi di interfacce:

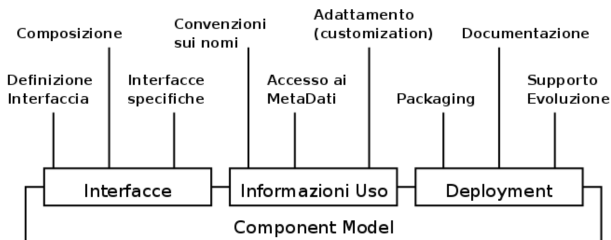
- Fornite
- Richieste



Modelli di componenti

Un modello dei componenti definisce uno “standard” da seguire per:

- l’implementazione di un componente
- documentazione
- impiego (deployment)



CCM (OMG), EJB (Sun Microsystems), COM+ (Microsoft)

Processo di sviluppo

- Requisiti in forma più generale ed identificazione di possibili componenti (provisioning)
- Raffinamento dei requisiti e adattamento dei componenti

Sono presenti almeno due nuove attività:

- Provisioning (Approviggionamento)
- Adattamento

E per le altre attività:

- Sviluppo
- Verifica e Validazione