Refactoring

Valter Crescenzi
crescenz@dia.uniroma3.it
http://www.dia.uniroma3.it/~crescenz

Cos'è il Refactoring?

- Il processo di modificare un sistema software senza alterarne il comportamento esterno pur migliorandone la struttura interna
- Un modo disciplinato di ripulire del codice esistente minimizzando le possibilità di introdurre nuovi bugs

Come?!?

- Ma il codice non si scriveva solo dopo la fase di progettazione?
  - alcune “scuole di pensiero” hanno rivisitato questa visione tradizionale anticipando la scrittura del codice (TDD, XP)
  - inoltre il refactoring può intervenire quando l’integrità del sistema, e la qualità del codice svanisce a causa di modifiche progressive

Software Decay & Refactoring

- Talvolta un sistema degenera al punto che ogni modifica diventa proibitiva, si ricorre ad “hack”
- Il Refactoring è una tecnica per fronteggiare questo scadimento del codice rispetto ai requisiti, attuali e futuri

Tanti Piccoli Passi = Molta Strada

- Il processo di refactoring è composto da passi minimali di modifica del codice
  - es. spostare un campo da un classe ad un'altra
- ciascuna modifica è talmente semplice da risultare agevole e priva di inconvenienti
  - non deve introdurre nuovi bug
- l'effetto cumulativo di numerose piccole modifiche può migliorare drasticamente il progetto

Esempi di Passi di Refactoring

- spostamento di un metodo da un classe ad un'altra
- incapsulamento di un campo con metodi accessori
- sostituzione di condizionali tramite polimorfismo
- estrazione di un nuovo metodo da un blocco di codice
- sostituzione di ereditarietà con composizione
- estrazione di una gerarchia da una classe
- risalita/discesa di un metodo in una gerarchia
- sostituzione di variabili temporanee con chiamate

Il Termine Refactoring

- Doppio significato:
  - Singolo passo di ristrutturazione del codice
    - per migliorarne la qualità senza alterarne il comportamento osservabile
  - Il processo di ristrutturazione del codice
    - cumulando tanti piccoli passi di ristrutturazione senza alterare il comportamento osservabile

Come?!? (2)

- Tanta fatica per non alterarne il comportamento osservabile?
  - anche l'ottimizzazione del codice non altera comportamento esterno pur modificandone la struttura interna
- L'obiettivo del refactoring è di rendere il codice più semplice da "maneggiare" e comprendere

Perché fare Refactoring?

- Per migliorare la progettazione del software
  - contrastando il “Software Decay”
    - spesso gli obiettivi di breve termine contrastano quelli di medio-lungo termine
- Per rendere il codice più comprensibile
  - Pensando allo sviluppatore futuro del tuo codice
    - P.S. soprattutto se questo sei sempre te
- Per trovare e correggere bugs
- Per programmare velocemente e con qualità

Come Fare Refactoring: con i Test

- Un prerequisito fondamentale è la disponibilità di solidi test che:
  - minimizzano la probabilità di introdurre bug
  - consentono di “misurare” l’affidabilità del codice
  - contrastano la “paura” di fare cambiamenti
- Risulta fondamentale mantenere il controllo del codice che non deve mai *allontanarsi troppo dal funzionare* >>

Come Fare Refactoring (2)

- Per ogni singolo passo (cfr. TDD)
  - si costruiscono i test per il risultato del refactoring
  - si ristruttura il codice
  - si eseguono i test e si fanno correzioni fino a ripristinare il completo funzionamento del codice
- Il codice deve ritornare a funzionare dopo ogni singolo passo di refactoring

Quando Fare Refactoring?

- Continuamente, con piccole sessioni unicamente dedicate al refactoring
- In genere l'esigenza del refactoring scaturisce quando si vuole fare qlc altro ed una sessione di refactoring aiuta in tal senso. Ad es.:
  - prima di aggiungere una funzionalità
  - per correggere un bug
  - durante una revisione del codice

Bad Smells in Code

- L'esigenza di un refactoring è usualmente denunciata da parti di codice di scarsa qualità
  - duplicazioni nel codice
  - metodo molto lungo
  - classe molto grande
  - lista di parametri lunga e complessa
  - logica condizionale sofisticata
  - classi che non fanno niente (solo dati)
  - gerarchie di classi parallele
  - troppa astrazione speculativa
  - … …

Un Catalogo di Refactoring

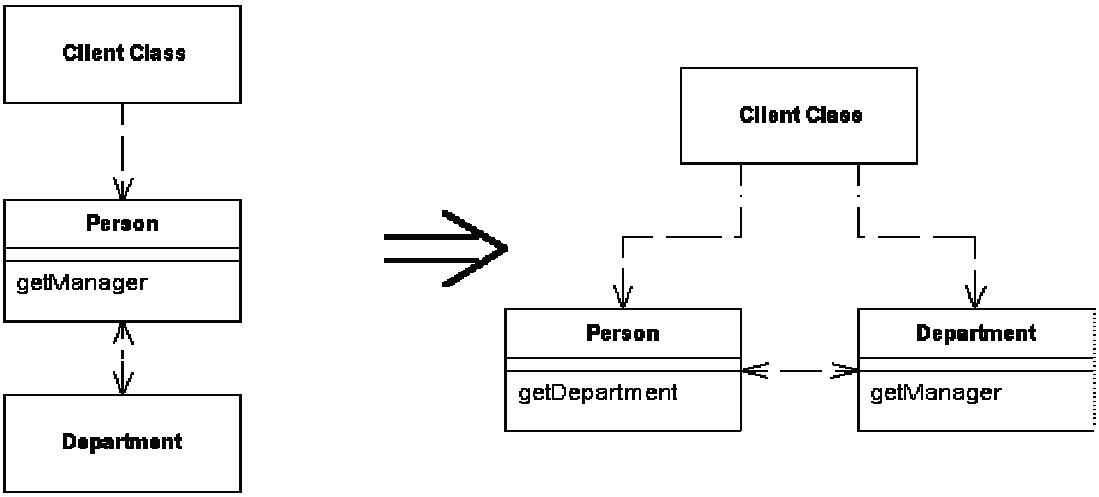
- Martin Fowler ha organizzato un catalogo di refactoring raggruppati secondo queste tipologie:
  - Composing Method
  - Moving Features Between Objects
  - Organizing Data
  - Simplifying Conditional Expressions
  - Making Method Calls Simpler
  - Dealing with Generalization

Formato delle Voci del Catalogo

- Name
- Summary
- Motivation
- Mechanics
- Examples

Un Esempio dal Catalogo (1)

- Remove Middle Man
  - una classe fa solo deleghe



  - Il suo inverso è “Hide Delegate”

Un Esempio dal Catalogo (2)

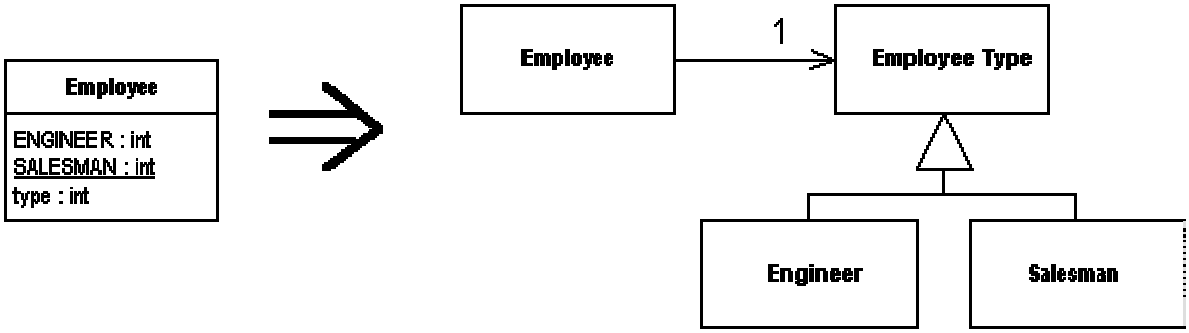
- Split Loop
  - Un ciclo fa più cose contemporaneamente

```
void printValues() {
  double averageAge = 0;
  double totalSalary = 0;
  for (int i=0;i<people.length;i++) {
    averageAge += people[i].age;
    totalSalary += people[i].salary;
  }
  averageAge=averageAge/people.length;
  System.out.println(averageAge);
  System.out.println(totalSalary);
}
```

```
void printValues() {
  double totalSalary = 0;
  for (int i=0;i<people.length;i++) {
    totalSalary += people[i].salary;
  }
  double averageAge = 0;
  for (int i=0;i<people.length;i++) {
    averageAge += people[i].age;
  }
  averageAge=averageAge/people.length;
  System.out.println(averageAge);
  System.out.println(totalSalary);
}
```

Un Esempio dal Catalogo (3)

- Replace Type Code with State/Strategy
  - Un codice di tipo determina il comportamento di una classe
  - Utile in alternativa a “Replace Type Code with Subclasses”



Refactoring e Performance (1)

- Il Refactoring privilegia esplicitamente la semplicità, ed alcuni refactoring sembrano introdurre inefficienze
- Tra le varie motivazioni primeggia:
  - la progressiva diminuzione del costo delle risorse hardware in rapporto al costo delle risorse umane
  - ed inoltre…

Refactoring e Performance (2)

- Ha sempre meno senso affidarsi al proprio intuito per individuare le porzioni di codice da ottimizzare
  - i modelli di esecuzione del programmatore possono non corrispondere a quelli reali, sempre più complessi
- Le ottimizzazioni che si fanno durante la scrittura del codice 9 volte su 10 non hanno alcun effetto pratico finale
  - tranne il continuo dispendio di energie del programmatore
- Il codice di qualità è più facile da ottimizzare in una fase finale esclusivamente dedicata a questo e basata su misurazioni sperimentali

Refactoring & Design Pattern

- Il target dei refactoring spesso sono Design Pattern
  - ovvero soluzioni riusabili già dimostrate efficaci in tanti differenti contesti tutti simili tra loro
- Il refactoring aiuta ad trovare il giusto livello di sofisticazione della soluzione
  - contro il pericolo di Over/Under-Engineering

Modulazione dei Refactoring

- Modulazione del refactoring, un compromesso tra
  - dimensione del refactoring
  - rischi connessi
- il refactoring permette di ridurre e controllare i rischi che un progetto comporta
- assieme ai test, permette di far evolvere il codice lungo un binario di correttezza e qualità assicurando un rapido feedback

Refactoring Tool

- Esistono alcuni strumenti che supportano semplici refactoring (semi-)automatici. Ad es.:
  - Rename Method
  - Extract Method
  - Encapsulate Field
  - …
- I tool devono mantenere una rappresentazione dei sorgenti per coglierne almeno parzialmente gli aspetti semantici oltre che sintattici
- E' importante lavorare il più possibile con sorgenti corretti anche durante i passi di refactoring

Refactoring Tool (2)

- Vedi demo Flash di RefactorIT

Limiti del Refactoring

- Non è stata fatta sufficiente esperienza per essere sicuri dei limiti
- Alcuni sembrano più che altro ereditati dalla complessità intrinseca del modello OO
  - molti non sono completamente automatizzabili
  - servono programmatori sempre più capaci

Riferimenti Bibliografici

- "*Refactoring – Improving the Design of Existing Code*", Martin Fowler, Addison-Wesley *www.refactoring.com*
- "*eXtreme Programming eXplained: Embrace Change*", Kent Beck, Addison-Wesley, 2000
- JUnit Open-Source Testing Framework. Kent Beck and Erich Gamma. *www.junit.org*
- "*Design Patterns: Elements of Reusable Object Oriented Software*", E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1997
- refactorIT: *www.refactorit.com*