# Il refactoring – 2

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 17
17/5/2004

## Commenti sui 2 seminari [1/2]

- Applicazioni Web
- Java "lato server": J2EE, introduzione
- Servlet
  - Programma Java, scrive su un flusso di out la pagina HTML
- JSP
  - Si scrive in HTML, con Java immerso in HTML, eseguito sul server (traduzione in servlet, …)
- Tomcat
  - Contenitore di servlet (e JSP)



## Commenti sui 2 seminari [2/2]

- JavaBean e JDBC
  - Two-tier e three-tier
  - Disaccoppiamento
- In realtà
  - Persistence layer (Torque)
  - Enterprise Java Beans (EJB)
  - … e molto altro in J2EE…
- Non solo Java: CGI, PHP, ASP, .NET, XSLT, …
- UML x Web



## Riassunto

- Il Refactoring
  - "rifattorizzazione", "ristrutturazione", "re-ingegnerizzazione", …)
- Definizione breve
- Refactoring al lavoro: un esempio
- La lezione più importante: Il "ritmo" del refactoring
  - Modifica (piccola), seguendo una procedura definita in modo preciso, compilare ed eseguire i test



## Scaletta

- Definizione
- Principi generali
  - I due cappelli
  - Perché e Quando rifattorizzare
  - Problemi con il refactoring
  - Relazioni con progetto e prestazioni
- Cataloghi
  - I refactoring (72 – 93)
  - Quando fare refactoring ("puzze")
  - Associazioni puzze/refactoring



## Refactoring: Definizione/i

- Nome
  - Un cambiamento alla struttura interna di un software
  - per renderlo più comprensibile e modificabile
  - senza modificare il suo comportamento osservabile
- Verbo
  - Ristrutturare un software, attraverso applicazioni di una serie di refactoring (nome!), senza cambiarne il comportamento

## Refactoring: principi generali

- Refactoring e "pulizia codice"
- I due cappelli
- Perché fare refactoring?
- Quando fare refactoring?
- Problemi con il refactoring
- Refactoring e progetto
- Refactoring e prestazioni



## Refactoring = "pulizia codice"?

- Sì e no
- Pulizia del codice che funziona già in modo controllato e più efficiente
  - Il "refactorer" esperto fa pulizia in modo molto più efficace
  - IMPORTANZA dei test automatici!
- Obiettivo del refactoring: codice che sia più facile da capire e modificare
  - Ben ≠ da ottimizzazione delle prestazioni!



## I due cappelli

1. Aggiunta funzioni
   - Non modificate il codice esistente
   - Aggiungete funzioni e test
2. Refactoring
   - Ristrutturate il codice e basta
   - Non aggiungete nuovi casi di test (a meno che non ne troviate di mancanti)

- Indossate solo un cappello e siate sempre consapevoli di quale state indossando



## Perché fare refactoring?

- Non è una panacea, è uno strumento utile
- Migliora la qualità del codice
- 4 motivi
  1. Migliora il progetto del software
  2. Rende il software più semplice da capire
  3. Aiuta a trovare i bug
  4. Vi aiuta a programmare più velocemente (!)



## Perché 1: progetto migliore

- Il progetto di un software "decade" col tempo
  - Aggiunte funzionalità
    - in tutta fretta
    - senza visione d'insieme
  - … disordine, entropia, …
- Refactoring rimette un po' in ordine
  - Es.: elimina codice duplicato, il codice deve dire ogni cosa una volta e una sola volta



## Perché 2: software più semplice da capire

- Programmare
  - Conversazione con il calcolatore
  - Conversazione con altri umani
- "Programmi che parlano" (…XP…)
- Cos'è peggio:
  - 1 sec. macchina in più
  - 1 giorno/uomo speso a capire il codice
- Refactoring quando si cerca di capire cosa fa un software

## Perché 3: aiuta a trovare i bug

- Con il refactoring chiarifico quello che fa (e dovrebbe fare) un software
- e quindi è più facile trovare errori
- 2 tipi di programmatori
  - "Grandi" programmatori, istinto
  - Buoni programmatori con grandi abitudini



## Perché 4: aiuta a programmare più velocemente

- Contro-intuitivo?
- Pensate alle nottate passate alla ricerca di bug…
- Passare il tempo ad aggiungere funzioni, non a cercare bug!
- Programmate più velocemente perché il refactoring evita il decadimento del progetto…



## Quando fare refactoring? [1/2]

- Non pianificate "2 settimane ogni 2 mesi"
- Rifattorizzate quando
  - volete aggiungere qualcosa
  - e il refactoring vi aiuta a farlo meglio e più velocemente
  - Rifattorizzate a piccoli pezzetti
- "Regola del tre" (?)
  - La prima volta che fate qualcosa, fatelo
  - La seconda, duplicate (turandovi il naso)
  - La terza rifattorizzate



## Quando fare refactoring? [2/2]

- Quando aggiungete una funzionalità
  - Perché è più veloce…
- Durante il debugging
  - Per capire meglio il codice
- Durante un "code review"
  - Risultato concreto
  - XP, Pair programming



## Problemi con il refactoring

- È giovane, ancora da capire appieno
- Il capo
  - Quality-oriented: ok
  - Altrimenti, non diteglielo e fatelo "di nascosto"… è comunque programmazione…
- Database: coerenza attributi in classi e tabelle
- Modifica delle interfacce (ripercussioni)
- Refactoring vs. rifare
  - Se il codice è pessimo, non rifattorizzate: rifate!
  - Refactoring come debito



## Refactoring e progetto [1/2]

- Refactoring e progetto:
  - Complementari, alternativi
- Visioni estreme:
  - classica (e diffusa)
    - Il progetto è la cosa difficile/importante (ingegnere)
    - La programmazione è "meccanica" (muratore)
  - alternativa (XP):
    - il software è ben diverso dalle macchine "fisiche", …
    - progetto non conta, è importante la programmazione

## Refactoring e progetto [2/2]

- Progetto comunque IMP., ma col refactoring:
  - Semplicità del progetto
  - Se so che il progetto semplice può essere poi rifattorizzato in quello più flessibile, non perdo tempo a fare il progetto più flessibile!
- Motivazioni psicologiche
  - Spesso tutta la flessibilità si rivela inutile, e ciò è frustrante…
  - Non "ho paura" di fare il progetto sbagliato



## Refactoring e prestazioni [1/2]

- Molti dei refactoring
  - Semplificano il codice
  - Lo rendono meno efficiente
- Demitizziamo l'importanza dell'efficienza
- Regola del 90 – 10%
  - I programmi passano il 90% del tempo nel 10% del codice
- Se ottimizzo le prestazioni di tutto il codice, il 90% dell'ottimizzazione è sprecato!



## Refactoring e prestazioni [2/2]

- Profiling!!!!
  - Quando si giudica "a occhio" dove bisogna migliorare le prestazioni, si sbaglia sempre!
- L'ottimizzazione va fatta alla fine!
- Ottimizzare un software intricato è difficile…
  - … e spesso porta a un software sbagliato…
- Il refactoring aiuta a scrivere software più efficiente



## Scaletta

- Definizione
- Principi generali
  - I due cappelli
  - Perché e Quando rifattorizzare
  - Problemi con il refactoring
  - Relazioni con progetto e prestazioni
- Cataloghi
  - I refactoring (72 – 93)
  - Quando fare refactoring ("puzze")
  - Associazioni puzze/refactoring



## I refactoring

- Sono tanti, non li vediamo tutti bene
  - Alcuni in dettaglio
  - Altri più velocemente
- Descrizione di ogni refactoring [Fowler 2000]
  - Nome
  - Riassunto
  - Motivazione
  - Meccanica
  - Esempi



## Divisi in 6 gruppi

1. Composizione di metodi
2. Spostamenti fra oggetti
3. Organizzazione dei dati
4. Semplificazione di espressioni condizionali
5. Semplificazione di invocazioni di metodi
6. Gestione della generalizzazione

## 1. Composizione di metodi

- Come arrivare a metodi ben fatti
  1. Extract Method
  2. Inline Method
  3. Inline Temp
  4. Replace Temp with Query
  5. Introduce Explaining Variable
  6. Split Temporary Variable
  7. Remove Assignments to Parameters
  8. Replace Method with Method Object
  9. Substitute Algorithm



## 1.1 Extract Method: def. ed es.

- → Raggruppo un frammento di codice in un metodo nella stessa classe

```
void stampaRendiconto(double ammontare) {
  stampaIntestazione();

  // stampa dettagli
  System.out.println("nome:" + _nome);
  System.out.println("ammontare:" + ammontare);
}
```

```
void stampaRendiconto(double ammontare) {
  stampaIntestazione();
  stampaDettagli(ammontare);
}

void stampaDettagli(double ammontare) {
  System.out.println("nome:" + _nome);
  System.out.println("ammontare:" + _ammontare);
}
```

## Extract Method: commenti

- Importanza dei nomi
  - Vanno scelti bene
  - Possono sostituire un commento
- Abituarsi a metodi corti
  - Più comprensibili
  - Più riusabili
- Quand'è che un metodo è lungo?
  - Boh?
  - "Distanza semantica" fra il nome e il corpo



## Extract Method: meccanica

1. Creare un nuovo metodo (stessa classe)
   1. Nome sulla base di cosa fa, non di come lo fa…
2. Copiare codice da metodo vecchio a nuovo
3. Cercare uso di variabili locali/parametri
   1. Se usate solo nel codice estratto e in lettura ⇒ variabili locali/parametri nel nuovo metodo
   2. Se modificate nel codice estratto ⇒
      1. Se solo una variabile ⇒ valore restituito dal metodo
      2. Altrimenti non si può, servono altri refactoring
4. Compilare
5. Invocazione nel vecchio metodo (rimuovendo variabili locali non più usate)
6. Compilare e testare



## 1.2 Inline Method: def. ed es.

- Metodo il cui corpo è chiaro quanto il nome: → elimino il metodo
- Inverso di Extract Method (!)

```
int getPuntualita() {
  return (piuDiCinqueSpedizioniInRitardo())?2:1:
}

boolean piuDiCinqueSpedizioniInRitardo() {
  return _numeroDiSpedizioniInRitardo > 5;
}
```

```
int getPuntualita() {
  return (_numeroDiSpedizioniInRitardo>5)?2:1:
}
```



## Inline Method: meccanica

1. Controllate che il metodo non sia sovrascritto
   - Se lo rimuovo, non posso più sovrascriverlo…
2. Trovate tutte le invocazioni
3. Rimpiazzarle con il corpo
4. Compilare e test
5. Rimuovere la definizione del metodo
6. Compilare e test

## 1.3 Inline Temp

- → Elimino una variabile temporanea a cui è assegnato un unico valore
  - La variabile può dar fastidio ad altri refactoring, ad es. Extract Method

```
double prezzoBase = ordine.prezzoBase();
return (prezzoBase > 100);
```

```
return (ordine.prezzoBase() > 100);
```



## Inline Temp: meccanica

1. Dichiarare la variabile temporanea come **`final`**
   - Compilare e testare
   - Verifica che c'è un unico assegnamento
2. Trovare tutti i riferimenti e rimpiazzarli con la parte dx. dell'assegnamento
3. Rimuovere la dichiarazione di variabile
4. Compilare e test



## 1.4 Replace Temp with Query: def.

- Quando viene usata una variabile temporanea per contenere il risultato di un'espressione
- → Estraggo l'espressione in un metodo
- e sostituisco tutti i riferimenti alla variabile temporanea con invocazioni del metodo



## Replace Temp with Query: es.

```
double prezzoBase = _quantita * _prezzoArticolo;
if (prezzoBase > 1000)
  return prezzoBase * 0.95;
else
  return prezzoBase * 0.98;
```

```
if (prezzoBase() > 1000)
  return prezzoBase() * 0.95;
else
  return prezzoBase() * 0.98;

...

double prezzoBase() {
  return _quantita * _prezzoArticolo;
}
```



## 1.5 Introduce Explaining Variable

- Se ho un'espressione complicata
- → Metto il risultato dell'espressione (o una sua parte) in una variabile temporanea con nome esplicativo
- Inverso di Inline Temp
- In alternativa a Extract Method
  - (variabile temporanea invece di metodo)
  - Ma Extract Method può essere usato in tutta la classe, e se pubblico anche da altre classi



## 1.6 Split Temporary Variable

- Se ho una variabile temporanea a cui vengono assegnati due valori scorrelati
  - (la variabile fa 2 cose ≠)
- → Creo due variabili temporanee separate, una per ogni assegnamento

## 1.7 Remove Assignments to Parameters

- Se il corpo di un metodo assegna un valore a un parametro
- → Evito l'assegnamento e uso una variabile temporanea

```
int sconto(int valInput) {
  if (valInput > 50)
    valInput -= 2;
  return valInput;
```

```
int sconto(int valInput) {
  int result = valInput;
  if (valInput > 50)
    result -= 2;
  return result;
```



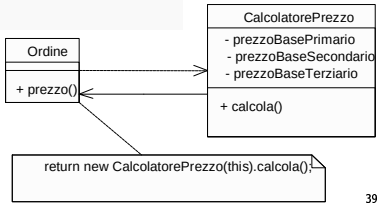
## 1.8 Replace Method with Method Object: def.

- Se ho un metodo lungo che usa parecchie variabili locali
  - (e quindi Extract Method è difficile/complicato)
- → Creo una nuova classe che incapsula il metodo
  - E in cui tutte le variabili locali sono attributi
  - (Così ora posso usare Extract Method senza preoccuparmi delle variabili locali!)



## Replace Method with Method Object: es.

```
class Ordine
  double prezzo (){
  double prezzoBasePrimario;
  double prezzoBaseSecondario;
  double prezzoBaseTerziario;
  //... Calcolo luuuuuuuungo
  }
}
```






## 1.9 Substitute Algorithm

- Rimpiazzare un algoritmo complicato con uno semplice
- Fare attenzione che i test funzionino…



## 2. Spostamenti fra oggetti

- Per decidere dove mettere le responsabilità
  1. Move Method
  2. Move Field
  3. Extract Class
  4. Inline Class
  5. Hide Delegate
  6. Remove Middle Man
  7. Introduce Foreign Method
  8. Introduce Local Extension



## 2.1 Move Method

- Metodo che usa più caratteristiche di un'altra classe che di quella in cui è
- → Creo un nuovo metodo, con corpo simile a quello di partenza, nella nuova classe
  - O rimuovo il vecchio metodo
  - O lo lascio e uso la delega
- Fare attenzione a
  - Sovrascrittura
  - Attributi/metodi privati usati dal vecchio metodo

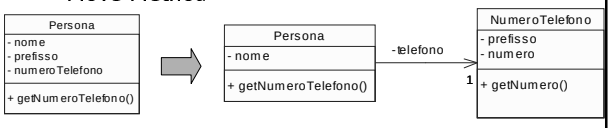
## 2.2 Move Field

- Attributo nella classe sbagliata
  - Perché, ad es., viene usato più nelle altre classi che in quella in cui è…
- → Lo sposto nella classe "giusta"



## 2.3 Extract Class

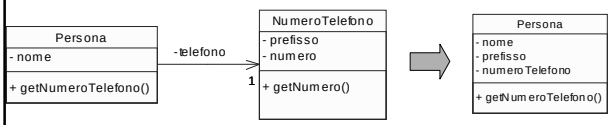
- Quando ho una classe che fa due cose distinte
- → Creo una nuova classe e vi sposto gli attributi e metodo opportuni
  - Move Field
  - Move Method






## 2.4 Inline Class

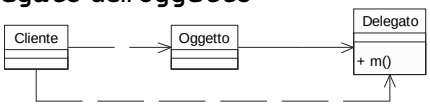
- Classe che non fa molto
- → Sposto tutti i suoi attributi e metodi e la rimuovo
- Inverso di Extract Class



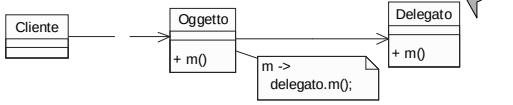



## 2.5 Hide Delegate

- Cliente che usa un **`Oggetto`** e anche un **`Delegato`** dell'**`Oggetto`**




- → Aggiungo un metodo a **`Oggetto`** per nascondere il delegato






## 2.6 Remove Middle Man

- Inverso di Hide Delegate
- Prezzo di Hide Delegate:
  - "Ingrasso" l'interfaccia di **`Oggetto`**
  - Se ho tanti metodi… **`Oggetto`** fa essenzialmente da "middle man" → È meglio che **`Cliente`** chiami direttamente **`Delegato`**



## 2.7 Introduce Foreign Method

- Un **`Cliente`** sta usando una classe **`Servitore`**
- e gli farebbe comodo che **`Servitore`** avesse un metodo in più, ma non la può modificare
- → Creo un metodo "estraneo" in **`Cliente`** con, come primo argomento, un'istanza di **`Servitore`**

## Introduce Foreign Method: es.

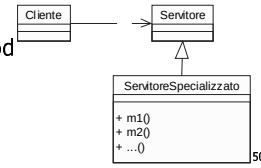
```
Date inizio2 = new Date(fine1.getYear(),
               fine1.getMonth(), fine1.getDay()+1);
```

```
Date inizio2 = giornoDopo(fine1);
...
private static Date giornoDopo(Date arg) {
  return new Date(arg.getYear(), arg.getMonth(),
                  arg.getDay()+1);
}
```



## 2.8 Introduce Local Extension

- Cliente sta usando una classe Servitore
- e gli vorrebbe aggiungere vari metodi, ma non la può modificare
- → Creo una nuova classe con i nuovi metodi
  - Come sottoclasse o come wrapper
  - Con più di 1-2 metodi, Introduce Foreign Method diventa scomodo…






## Riassunto

- Definizione
- Principi generali
  - I due cappelli
  - Perché e Quando rifattorizzare
  - Problemi con il refactoring
  - Relazioni con progetto e prestazioni
- Cataloghi
  - I refactoring (72 – 93)
  - Quando fare refactoring ("puzze")
  - Associazioni puzze/refactoring