Il refactoring

Stefano Mizzaro

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

PAOO, Lezione 15
10/5/2004

Riassunto

- OO
- Intro UML
- Criteri per buon OO
- I Design pattern

Scaletta e bibliografia

- Il Refactoring
  - "rifattorizzazione", "ristrutturazione", "re-ingegnerizzazione", …)
- Definizione breve
- Refactoring al lavoro: un esempio
- Poi
  - Definizione completa
  - Puntualizzazioni, principi generali, …
- M. Fowler. *Refactoring – Improving the Design of Existing Code*, Addison Wesley, 2000
  - Oggi: capp. 1, 2
  - Un po' blabla, ma ben fatto
- http://www.refactoring.com/

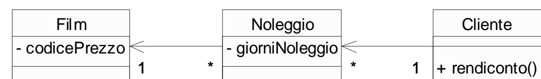
Refactoring: definizione

- Il processo di modifica di un sistema software in modo da:
  - non alterare il comportamento esterno
  - migliorare la sua struttura
- Non è detto che sia OO… ma è nato in ambiente OO (Smalltalk)
- È un procedimento sistematico

Un esempio giocattolo: negozio di noleggio video

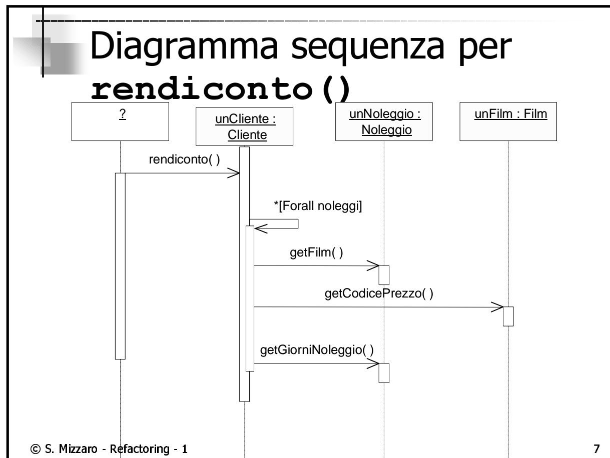
- Programma per calcolare e stampare il rendiconto di un cliente
- Il programma ha in input:
  - Quanti film sono stati noleggiati dal cliente
  - Per quanto tempo
- Il programma calcola (secondo certe regole)
  - L'ammontare totale dovuto
  - I punti per noleggi frequenti
- 3 tipi di film: regolare, per bambini, novità

Diagramma UML classi



- (Vedi il codice versione 1.0)

Diagramma sequenza per `rendiconto()`



Commenti

- "quick and dirty simple program"… OK!
- Ma facciamo finta che sia più complicato
- Non OO
- **`rendiconto()`**
  - Trooooppo lungo
  - Fa troppe cose, che andrebbero in altre classi

2 nuovi requisiti

- Il committente vuole:
  - rendiconto in HTML (cut&paste, duplicazione)
  - cambiare le regole di classificazione film
    - Non ha ancora deciso come… ne ha un po' in mente…
    - Ma tanto sappiamo che fra 2 mesi cambierà idea ☺
- Momento giusto per il refactoring
  - Devo aggiungere nuove funzionalità…
  - … ma per come è fatto il programma, è difficile
  - Se non rifattorizzo, mi aspetta un incubo…

Insieme di test

- Quindi, rifattorizziamo
- 1a cosa: preparare un insieme di TEST!
- JUnit
  - Creo qualche cliente
  - Gli associo qualche noleggio di film di vario tipo
  - Genero le stringhe di rendiconto
  - Le confronto con stringhe corrette e controllate/generate a mano
  - IMPORTANTE: risposta dei test: OK/KO
- Vedi codice 2.0 (che sarebbe da fare meglio)

Primo refactoring

- Obiettivo (ovvio): il metodo luuuuuungo (**`rendiconto`** in **`Cliente`**)
  - Scomporlo in pezzi più corti e gestibili
  - Per rendere più semplice la generazione in HTML
- Estraiamo un pezzo di codice e mettiamolo in un metodo
  - Che assegna a **`questoAmmontare`** il valore corretto
  - "Extract Method"
  - Sullo **`switch/case`**

Extract method passo 1

- Attenzione alle variabili locali!
  - **`ognuno`**
    - Non modificata
    - La passo come argomento
  - **`questoAmmontare`**
    - Viene modificata
    - Ce n'è solo una → valore restituito dal metodo
- Vedi codice 3.0
- N.B. Esistono tool automatici che aiutano… ma facciamolo a mano

Importante

- Ricompilare e rieseguire i test
- … guarda caso non funzionano…
- …perché il tipo restituito di **`ammontarePer`** deve essere **`double`**, non **`int`**. Pistola.
- Correggo (3.1), ricompilo, rieseguo, tutto ok.
- Morale
  - È facile fare errori scemi, è importante controllare
  - Siccome i cambiamenti sono piccoli, gli errori sono facili da trovare

I nomi sono importanti

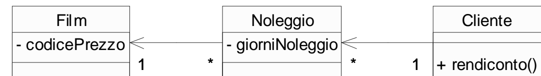
- Cambiamo alcuni nomi di variabile in **`ammontarePer`** (3.2)
- … e rieseguiamo i test
- Sull'importanza dei nomi:
  - Ogni mona è capace di scrivere codice che solo un computer può capire. I bravi programmatori scrivono codice che gli umani capiscono…
  - Non abbiate paura ("coraggio" di XP): search&replace, typecheck del compilatore, test.

Secondo refactoring

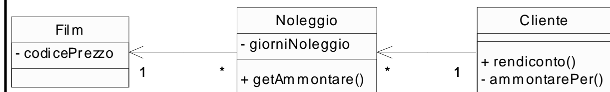
- Guardiamo **`ammontarePer`**: non ci piace molto
- Usa informazioni da **`Noleggio`** (e anche da **`Film`**), ma non da **`Cliente`**!
- Indizio che è nella classe sbagliata
  - "Move Method"
- Mettiamolo in **`Noleggio`** (4.0)
  - Gli cambiamo nome (**`getAmmontare`**) e lo rendiamo **`public`**
  - Eliminiamo il parametro e usiamo **`this`** (opzionale)
  - Invochiamolo da **`ammontarePer`** di **`Cliente`**
  - Ricompiliamo, ritestiamo, …

Diagramma UML

- Era



- Ora è (Extract Method + Move Method)



Move Method: 2o passo [opzionale]

- Trovare le invocazioni al vecchio metodo e sostituirle con invocazioni al nuovo metodo
- Ricompilare & ritestare
- Eliminare il vecchio metodo
- Ricompilare & ritestare
- Qui è facile, un unico punto
- Opzionale perché si può lasciare la delega dal vecchio metodo al nuovo
  - Ad es. se il vecchio è parte dell'interfaccia

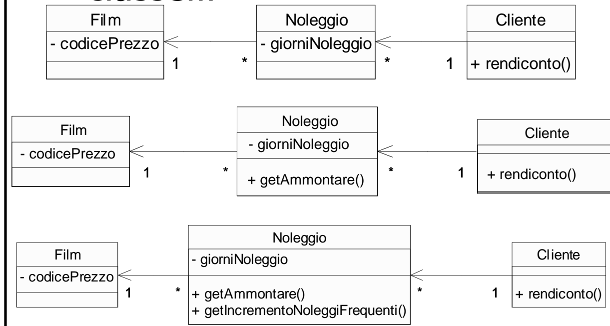
Terzo refactoring

- In **`rendiconto`**, la variabile temporanea **`questoAmmontare`** è superflua
  - Le assegno un valore che poi non cambio
- Eliminiamola (5.0)
  - "Replace Temp with Query"
- (Sull'efficienza):
  - 2 invocazioni invece di una
  - Perdita minima
  - Prima rifattorizzare per bene, poi pensare all'efficienza

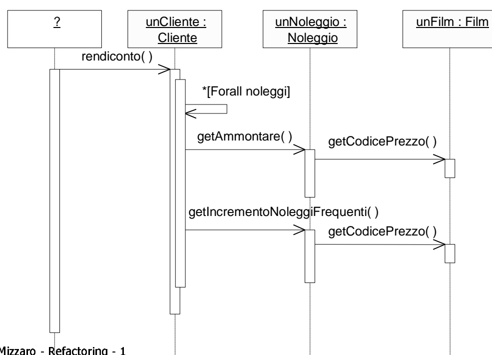
Quarto refactoring

- Sempre in **`rendiconto`** di **`Cliente`**, qualcosa di analogo con i punti per noleggi frequenti
- "Extract method"
  - Metto in un metodo privato il codice per calcolare l'incremento dei punti (6.0)
  - Ricompilo & ri-test
  - Metto il codice nel posto giusto (**`Noleggio`**)
    - Eliminando il metodo privato (6.1)
  - Ricompilo & ri-test

Ricapitoliamo: diagramma di classe…



…e diagramma di sequenza



Quinto refactoring

- Eliminiamo altre variabili temporanee
  - Incoraggiano metodi lunghi
- Sempre in **`rendiconto(),`** ci sono
  - **`ammontareTotale`**
  - **`puntiNoleggiFrequenti`**

  usate come "accumulatori"
- "Replace Temp with Query"
  - Cominciamo a sostituire **`ammontareTotale`** con **`getAmmontareTotale()`** (7.0)
  - E poi **`puntiNoleggiFrequenti`** con **`getPuntiNoleggiFrequenti()`** (7.1)

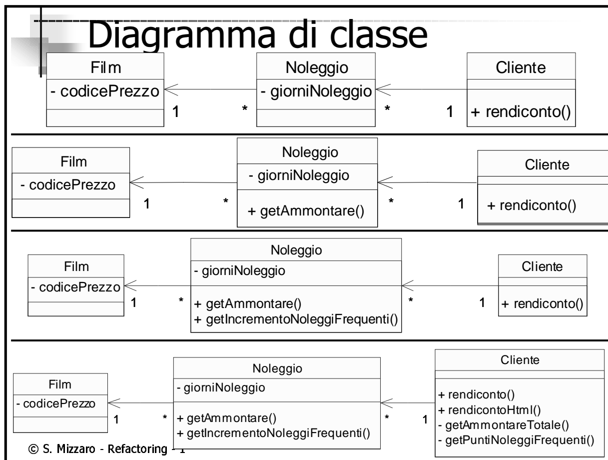
Commenti

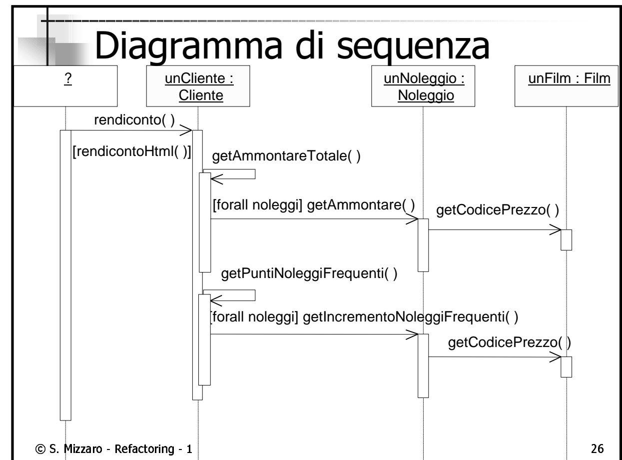
- Lunghezza codice
  - Con gli ultimi 2 refactoring ho aggiunto righe di codice
  - Di solito non è così
- Efficienza
  - 2 esecuzioni di ciclo invece di una
- Non preoccupiamocene:
  - Il codice è più chiaro
  - Questo ci consentirà una riorganizzazione più semplice

Due cappelli

1. "Refactoring"
2. "Aggiunta funzione"

- Importante distinguere i due ruoli
- Togliamo il primo e mettiamo il secondo: finalmente
  - **`rendicontoHtml()`** (8.0)





Ri-refactoring, su altro

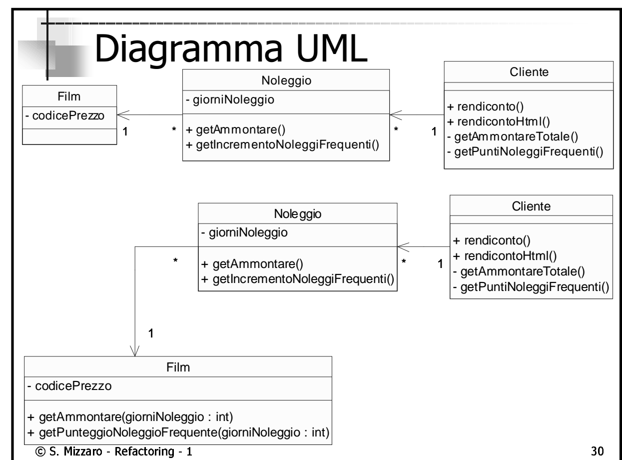
- Finora abbiamo lavorato su:
  - Rifattorizzare **`rendiconto()`** e
  - Aggiungere **`rendicontoHtml()`**
  - Ci sarebbere ancora "Extract Method",
    - (inizio e fine del rendiconto)
  - ma…
- Adesso pensiamo al secondo requisito:
  - Cambiare le regole di classificazione film
    - Non sappiamo ancora come… nuove classificazioni, modifica di quelle esisitenti, …
    - Ma sappiamo che fra 2 mesi saranno diverse…

Regole di classificazione

- In **`Noleggio`**,
  - Metodo **`getAmmontare()`**
    - **`switch/case`**
- Istruzioni condizionali ⇒ cambiamenti scomodi (almeno)
- Rifattorizziamo
  - Passiamo da logica condizionale a uso del polimorfismo
  - Ci mettiamo un po'…

Metodo nel posto sbagliato…

- Lo switch è su:
  - **`getFilm().getCodicePrezzo()`**
  - un attributo di un'altra classe
- Legge di Demeter…
- Brutto, spostiamolo in **`Film`**: "Move Method" (9.0)
  - Ricordiamoci di passare l'argomento **`giorniNoleggio`**
  - In questo modo ciò che riguarda i tipi di film (interessati dai nuovi reqs.) è tutto in **`Film`**…
  - …a parte **`getIncrementoNoleggiFrequenti()`** in **`Noleggio`** ⇒ spostiamo anche quello, in **`getPunteggioNoleggiFrequenti()`** di **`Film`** (9.1)



Tipi di film

- Vari tipi di film con risposte diverse alle stesse richieste
  - (solo **getAmmontare()**, per ora…)
- Ereditarietà, ma come?
- Prima possibilità
- … ma i film possono cambiare classificazione…

Film
- codicePrezzo
+ getAmmontare()
+ getPunteggioNoleggioFrequente()

Regolare: + getAmmontare()
Bambini: + getAmmontare()
Novita: + getAmmontare()

State pattern!

Noleggio
- giorniNoleggio
+ getAmmontare()
+ getIncrementoNoleggiFrequenti()

Cliente
+ rendiconto()
+ rendicontoHtml()
- getAmmontareTotale()
- getPuntiNoleggiFrequenti()

Film
- codicePrezzo
+ getAmmontare()

-prezzo

Prezzo
+ getAmmontare()

getAmmontare() -> return prezzo.getAmmontare();

Prezzo Regolare: + getAmmontare()
Prezzo Bambini: + getAmmontare()
Prezzo Novita: + getAmmontare()

Come arrivarci?

- Con calma… 3 passi
  1. "Replace Type Code with State/Strategy"
     - Creo la gerarchia di **Prezzo**…
     - … attributo **prezzo** in **Film**
  2. "Move Method"
     - Sposto lo switch di **getAmmontare()** da **Film** a **Prezzo**
  3. "Replace Conditional with Polymorphism"
     - Elimino lo switch grazie al polimorfismo

1. Replace Type Code with State/Strategy

- Anche qui vari passi
- In **Film**, evito accessi diretti a **_codicePrezzo**
  - L'unico accesso è nel costruttore, lo sostituisco usando **setCodicePrezzo()** (10.0)
  - "Self Encapsulate Field"
  - Compilo e test…
- Poi creo la gerarchia per i prezzi
  - 4 nuove classi, e le compilo… (10.1)
- Poi in Film aggiungo l'attributo **_prezzo** di tipo **Prezzo** e modifico **set/getCodicePrezzo** per usare le nuove classi (10.2)

2. Move Method

- **getAmmontare()** di **Film**
  - Spostiamolo in **Prezzo**… (non serve nessuna modifica)
  - …e delega in **Film**
- (10.3)

3. Replace Conditional with Polymorphism

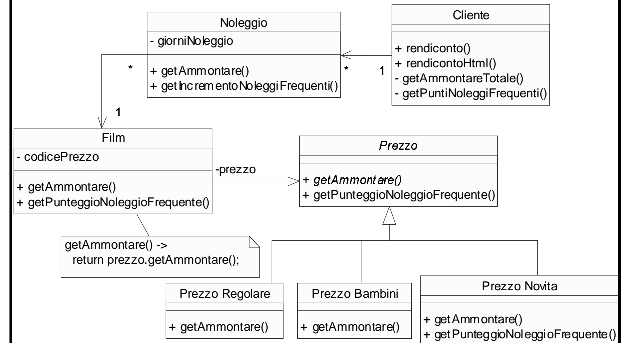
- Prendiamo un ramo dello switch alla volta e
  - sovrascriviamo **getAmmontare()** nella sottoclasse corrispondente
  - (senza modificare **getAmmontare()** di **Prezzo**, che però non viene più usato per i tipi di film che hanno il loro **getAmmontare()**)
  - Compiliamo e testiamo (e magari inseriamo qualche bug di proposito…)
- Ripetiamo per 3 volte e poi dichiariamo astratto **getAmmontare()** di **Prezzo** (10.4)

Anche l'altro metodo…

- Dopo **`getAmmontare()`**, anche **`getPunteggioNoleggiFrequenti()`**
- Stessa procedura
  1. "Replace Type Code with State/Strategy"
     - Creo la gerarchia di **`Prezzo`**… (già fatto)
  2. "Move Method"
     - Sposto l'if in **`Prezzo`**
  3. "Replace Conditional with Polymorphism"
     - Per eliminare l'if
     - Ma questa volta in **`Prezzo`** rimane il default, non astratto
- (10.5) – finale.

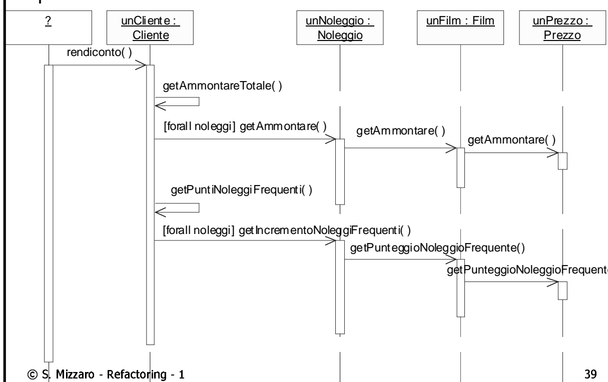
© S. Mizzaro - Refactoring - 1 37

Cosa abbiamo ottenuto



© S. Mizzaro - Refactoring - 1 38

Diagramma di sequenza



© S. Mizzaro - Refactoring - 1 39

Quindi

- Inserire il pattern State ci è costato un po' di fatica…
- Guadagno: nuovi cambiamenti sul tipo di film sono indolori
  - Comportamenti diversi per i tipi di prezzi già definiti
  - Nuovi tipi di prezzi
  - Nuovi comportamenti dipendenti dal prezzo
- In un esempio così banale non ne valeva la pena…

© S. Mizzaro - Refactoring - 1 40

Ricapitoliamo

- Abbiamo fatto due cambiamenti principali e aggiunto funzionalità
- Usando
  - Extract Method
  - Move Method
  - Replace Conditional with Polymorphism
  - Replace Type Code with State/Strategy
  - Self Encapsulate Field
  - Replace Temp with Query
- Più tempo a spiegarlo che a farlo…

© S. Mizzaro - Refactoring - 1 41

La lezione più importante: Il "ritmo" del refactoring

- Modifica (piccola),
- seguendo una procedura definita in modo preciso,
- compilare ed eseguire i test
- Morali:
  - Non fare il passo più lungo della gamba…
  - Andare con i piedi di piombo…
  - La pazienza è la virtù dei forti…
  - Chi va piano va sano e va lontano…

© S. Mizzaro - Refactoring - 1 42

Il refactoring

- Cosa abbiamo visto
  - Un esempio giocattolo
- Cosa bisogna sapere:
  - Principi generali, fondamenti, cosa, come, quando, perché, problemi potenziali, …
  - Cataloghi
    - Quando fare refactoring ("puzze")
    - I refactoring (72 – 93)
    - 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 in modo controllato e più efficiente
  - Il "refactorer" esperto fa pulizia in modo molto più efficace
  - IMP. dei test automatici!
- Obiettivo del refactoring: codice che sia più facile da capire e modificare
  - Ben ≠ da ottimizzazione delle prestazioni!

I due cappelli

- Aggiunta funzioni
  - Non modificate il codice esistente
  - Aggiungete funzioni e test
- Refactoring
  - Ristrutturate il codice e basta
  - Non aggiungete nuovi casi di test (a meno che non ne trovate 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
- 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
- "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

© S. Mizzaro - Refactoring - 1 55

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

© S. Mizzaro - Refactoring - 1 56

Refactoring e progetto [2/2]

- Progetto comunque IMP., ma il 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

© S. Mizzaro - Refactoring - 1 57

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!

© S. Mizzaro - Refactoring - 1 58

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

© S. Mizzaro - Refactoring - 1 59

Riassunto

- Abbiamo visto
  - Refactoring per migliorare la qualità del software (comprensibilità, efficienza, …)
  - Esempio giocattolo
  - Definizione, Principi generali, I due cappelli, Perché e Quando rifattorizzare, Problemi con il refactoring, relazioni con progetto e prestazioni
- Dobbiamo ancora vedere:
  - I cataloghi (Puzze e Refactoring)
  - Come si fa il refactoring in concreto

© S. Mizzaro - Refactoring - 1 60