Il refactoring – 3

Stefano Mizzaro

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

PAOO, Lezione 18
19/5/2004

Riassunto

- Esempio
- Definizione e principi generali
- I primi refactoring

Scaletta

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

3. Organizzazione dei dati

- Semplificare la gestione dei dati

1. Self Encapsulate Field
2. Replace Data Value with Object
3. Change Value to Reference
4. Change Reference to Value
5. Replace Array with Objects
6. Duplicate Observed Data
7. Change Unidirectional Association to Bidirectional
8. Change Bidirectional Association to Unidirectional
9. Replace Magic Number with Symbolic Constant
10. Encapsulate Field
11. Encapsulate Collection
12. Replace Record with Data Class
13. Replace Type Code with Class
14. Replace Type Code with Subclasses
15. Replace Type Code with State/Strategy
16. Replace Subclass with Fields

3.1 Self Encapsulate Field

- → Creo metodi `getXxx` e `setXxx` per accedere all'attributo `Xxx`
- 2 scuole di pensiero
  1. Accedere direttamente all'interno della classe (più leggibile, …)
  2. Accedere sempre tramite `get`/`set` (sottoclassi possono sovrascrivere, lazy initialization, …)
- Con Self Encapsulate Field, si può cominciare con 1 e passare a 2 quando serve…

3.2 Replace Data Value with Object

- Una classe ha un attributo di un certo tipo (spesso un tipo base)
- a cui aggiunge comportamento/dati
- → Creo per l'attributo una classe apposita in cui metto comportamento/dati aggiuntivi

3.3 Change Value to Reference

- Classe con molte istanze uguali, che però rappresentano una singola entità
- → le rimpiazzo con una singola istanza
- Es. tipici:
  - `Cliente`, `ContoCorrente`, …
- Contro: può essere difficile da gestire…

© S. Mizzaro - Refactoring - 3 7

3.4 Change Reference to Value

- Faccio fatica a mantenere un'unica istanza per tutte le entità di un certo tipo uguali fra di loro
- → le rimpiazzo con più istanze
- Es. tipici:
  - `Date`, `Soldi`, …
- Inverso del precedente

© S. Mizzaro - Refactoring - 3 8

3.5 Replace Array with Objects

- Un array contiene dati concettualmente diversi

```
String[] indirizzo = new String[4];
indirizzo[0] = "Roma";
indirizzo[1] = "17";
...
```

- → Trasformo l'array in un oggetto con un attributo per ogni elemento dell'array

```
Indirizzo indirizzo = new Indirizzo();
indirizzo.setVia("Roma");
indirizzo.setNumero("17");
...
```

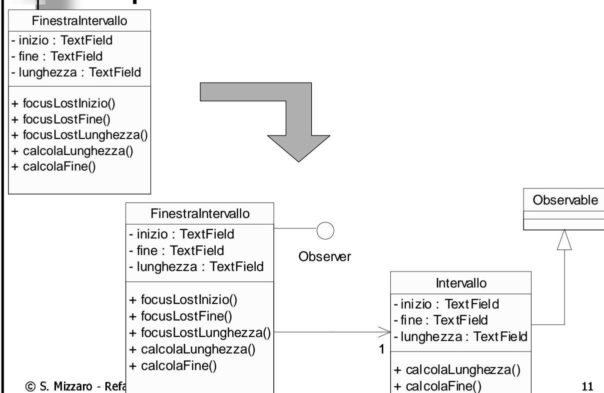
© S. Mizzaro - Refactoring - 3 9

3.6 Duplicate Observed Data

- Dati di dominio sono contenuti solo nelle classi della GUI, e le classi di dominio vi accedono
- → Copio di dati di dominio in un oggetto di dominio. Uso il pattern Observer per sincronizzare

© S. Mizzaro - Refactoring - 3 10

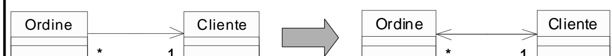
Duplicate Observed Data



© S. Mizzaro - Refa 11

3.7 Change Unidirectional Association to Bidirectional

- Ho due classi, una delle quali fa riferimento all'altra ma non viceversa. Ho bisogno anche del riferimento nell'altro senso
- → Aggiungo il "back pointer"
- Non è banale…



© S. Mizzaro - Refactoring - 3 12

3.8 Change Bidirectional Association to Unidirectional

- Inverso del precedente
- Le associazioni bidirezionali hanno costi:
  - Una maggiore complessità di gestione
  - Interdipendenze fra classi, accoppiamento ↑
  - Fonte di errori
- ⇒ Usarle solo se servono!

3.9 Replace Magic Number with Symbolic Constant

- USARE COSTANTI, NON LETTERALI!!!

```
double energiaPotenziale(double m, double h){
  return m * 9.81 * h;
}
```

```
double energiaPotenziale(double m, double h){
  return m * COSTANTE_GRAVITAZIONALE * h;
}
static final double COSTANTE_GRAVITAZIONALE = 9.81;
```

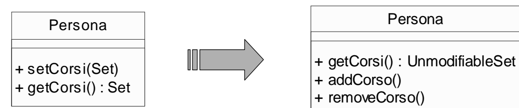
- Cfr. Replace Type Code with Class
- Usare `array.length`

3.10 Encapsulate Field

- Attributo `public` → Renderlo `private` e aggiungere i metodi `get`/`set`
- Poi controllare se ha senso Move Method
  - (altrimenti è una classe "stupida")
- Differenza da self encapsulate:
  - Qua l'attributo è `public`, là era `private`

3.11 Encapsulate Collection

- Un metodo di una classe restituisce una collection
- → Rendo immutabile la collection restituita
  - `Collections.unmodifiableCollection(Collection)`
- e aggiungo metodi add/remove

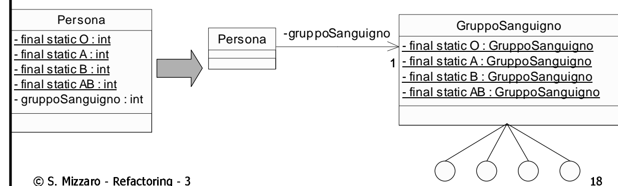


3.12 Replace Record with Data Class

- → Trasformo un record in una classe, con un attributo privato per ogni campo del record e metodi set/get per ogni attributo
- In Java non ci sono i record…
  - Altri linguaggi (C++, …)
  - Porting di un programma
  - Comunicazione con un linguaggio/API che ha i record
  - …

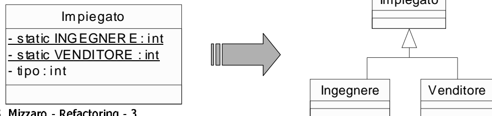
3.13 Replace Type Code with Class

- Attributo di tipo numerico
- → Lo trasformo in un attributo d'istanza di una nuova classe
  - I "type code" numerici non hanno controllo di tipo…



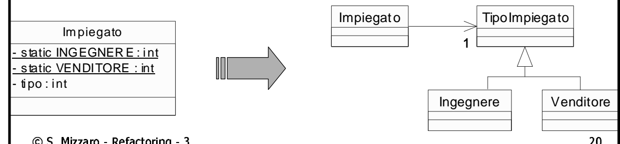
3.14 Replace Type Code with Subclasses

- Rispetto al precedente, qui il "type code" influenza il comportamento (ci sono if e switch) → usare polimorfismo
- Spesso premessa di
  - 4.6 Replace Conditional with Polymorphism
  - 6.4/5 Push Down Method/Field



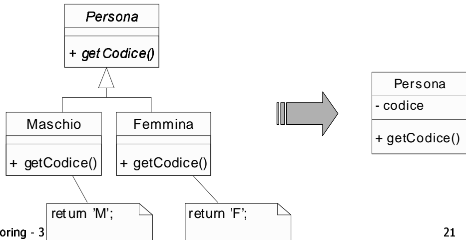
3.15 Replace Type Code with State/Strategy

- Simile al precedente, ma da usare se il "type code" cambia durante la vita dell'oggetto
- Usa il pattern State o Strategy



3.16 Replace Subclass with Fields

- Ho sottoclassi che differiscono solo per metodi che restituiscono dati costanti
- → Aggiungo un attributo nella superclasse ed elimino le sottoclassi



4. Semplificazione di espressioni condizionali

- Logica condizionale spesso intricata…
  1. Decompose Conditional
  2. Consolidate Conditional Expression
  3. Consolidate Duplicate Conditional Fragments
  4. Remove Control Flag
  5. Replace Nested Conditional with Guard Clauses
  6. Replace Conditional with Polymorphism
  7. Introduce Null Object
  8. Introduce Assertion

4.1 Decompose Conditional

- if/then/else complicato → estraggo metodi per condizione, ramo if e ramo else

```
if (data.primaDi(INIZIO_ESTATE)||
    data.dopoDi(FINE_ESTATE))
  costo=qta*_tariffaInverno*_tariffaServizioInverno;
else
  costo = qta * _tariffaEstate;
```

```
if (!inEstate(data))
  costo = costoInvernale(qta);
else
  costo = costoEstivo(qta);

... def metodi privati...
```


4.2 Consolidate Conditional Expression

- Sequenza di condizioni con lo stesso risultato → le combino in un'unica espressione condizionale e la estraggo in un metodo

```
double quantitaDisabilita() {
  if (_anzianita < 2) return 0;
  if (_mesi > 12) return 0;
  if (_partTime) return 0;
  // calcola la quantita' di disabilita'
}
```

```
double quantitaDisabilita() {
  if (nonHaDiritto()) return 0;
  // calcola la quantita' di disabilita'
}
... def. Metodo/i...
```

4.3 Consolidate Duplicate Conditional Fragments

- Lo stesso frammento di codice in tutti i rami di un if → Lo sposto al di fuori

```
if (affareSpeciale()) {
  tot = prezzo * 0.95;
  spedisci();
} else {
  tot = prezzo * 0.98;
  spedisci();
}
```

```
if (affareSpeciale())
  tot = prezzo * 0.95;
else
  tot = prezzo * 0.98;
spedisci();
```


4.4 Remove Control Flag

- Variabile di controllo → `break` o `return`

```
void m (String[] persone)
  boolean trovato = false;
  for (int i = 0; i < persone.length; i++)
    if (!trovato) {
      if(persone[i].equals("Meni")) {
        attenzione();
        trovato = true;
      }
      if(persone[i].equals("Toni")) {
        attenzione();
        trovato = true;
      }
    }
}
```

```
void m (String[] persone)
  for (int i = 0; i < persone.length; i++)
    if(persone[i].equals("Meni")) {
      attenzione();
      break;
    }
    if(persone[i].equals("Toni")) {
      attenzione();
      break;
    }
}
```

`return;`

4.5 Replace Nested Conditional with Guard Clauses

- `if/then/else` con "pesi" diversi ai 2 rami
  - (es.: situazione normale e anomala)
- → Usare `if+return` ("Guard clause")

```
double getAmmontare ()
  double res = 0;
  if(_morto) res = ammontareMorto();
  else {
    if (_separato) res = ammontareSeparato();
    else {
      if (_pensionato) res = ammontarePensionato();
      else res = ammontareNormale();
    }
  }
  return res;
}
```

```
double getAmmontare ()
  if (_morto) return ammontareMorto();
  if (_separato) return ammontareSeparato();
  if (_pensionato) return ammontarePensionato();
  return ammontareNormale();
}
```


4.6 Replace Conditional with Polymorphism

- Istruzione condizionale con comportamento ≠ a seconda del ramo
- → Sposto ogni ramo in un metodo sovrascrivente in una sottoclasse e rendo astratto il metodo originale nella superclasse

Replace Conditional with Polymorphism

```
double getVelocita() {
  switch (_tipo) {
    case EUROPA: return getVelocitaBase();
    case AFRICA:
      return getVelocitaBase() - getCarico() -
             getNumeroNociDiCocco();
    case NORVEGESE:
      return (_inverno ? 0 : getVelocitaBase();
  }
  throw new RuntimeException("Troppo veloce?");
}
```

Uccello
+ getVelocita()

Europeo
+ getVelocita()

Africano
+ getVelocita()

Norvegese
+ getVelocita()

4.7 Introduce Null Object

- Ho parecchi test per vedere se istanze di una certa classe sono in realtà `null`
- → Sostituisco il valore `null` con un "oggetto null"
  - Che è un Singleton

```
if (cliente == null) piano = PianoAddebiti.base();
else piano = cliente.getPiano();
```

Cliente
+ getPiano()

ClienteNull
+ getPiano()

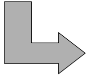
return PianoAddebiti.base();

`cliente.getPiano();`

4.8 Introduce Assertion

- Una parte di codice fa qualche assunzione sullo stato del programma
- → Rendo esplicita l'assunzione con un'asserzione

```
// x deve essere positivo
if (x >= 0) return Math.sqrt(x);
```



```
assert(x >= 0, "Errore: x negativo");
return Math.sqrt(x);
```


5. Semplificazione di invocazioni di metodi

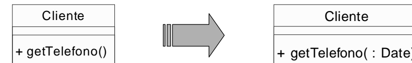
1. Rename Method
2. Add Parameter
3. Remove Parameter
4. Separate Query from Modifier
5. Parameterize Method
6. Replace Parameter with Explicit Methods
7. Preserve Whole Object
8. Replace Parameter with Method
9. Introduce Parameter Object
10. Remove Setting Method
11. Hide Method
12. Replace Constructor with Factory Method
13. Encapsulate Downcast
14. Replace Error Code with Exception
15. Replace Exception with Test

5.1 Rename Method

- → Modifico il nome di un metodo per renderlo più comprensibile
- I nomi sono importanti
  - Ancora di più se si usano metodi brevi
- Quando vedete un nome non ottimale, cambiatelo!

5.2 Add Parameter

- Un metodo ha (ora) bisogno di più dati
- → Aggiungo un parametro
- Alternative (spesso migliori)
  - Controllare che il metodo stia "bene" nella classe
  - Usare/aggiungere metodi di altri oggetti "disponibili" al metodo
  - Usare 5.9 Introduce Parameter Object



5.3 Remove Parameter

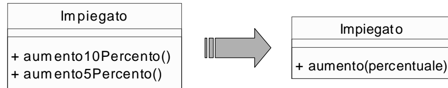
- Parametro non (più) usato
- → Lo rimuovo
  - (lasciarlo ↓ leggibilità)
- Inverso del precedente
- Attenzione ai metodi sovrascritti…

5.4 Separate Query from Modifier

- Ho un singolo metodo che restituisce un valore e modifica un oggetto
- → Creo 2 metodi, uno get e uno set
- I metodi che restituiscono un valore non dovrebbero avere side effect osservabili
  - (get successivi danno valori diversi)
  - Ad es., "memoization" è un effetto collaterale non osservabile

5.5 Parameterize Method

- Ho molti metodi che fanno cose simili, che differiscono solo per un valore nel corpo
- → Creo un unico metodo con un parametro per quel valore



5.6 Replace Parameter with Explicit Methods

- Ho un metodo che esegue codice ≠ a seconda del valore di un parametro
- → Creo un metodo separato per ogni valore del parametro
- Inverso del precedente
- ↑ chiarezza:
  - `interruttore.on()` vs. `interruttore.set(true)` o `interruttore.set(ON)`

5.7 Preserve Whole Object

- Ottengo (metodi get) vari valori da un oggetto e li passo tutti come parametri
- → Passo tutto l'oggetto come parametro (eventualmente **`this`**)
- Alternativa: Move Method (posto sbagliato?)
- Contro: dipendenza (**`z`** dipende da **`p`**…)

```
int x = p.getX();
int y = p.getY();
z.m(x,y);
```

→

```
z.m(p);
```


5.8 Replace Parameter with Method

- Ho un oggetto che invoca un metodo **`m`** e ne passa il risultato come parametro a un altro metodo **`m'`**
- → Rimuovo il parametro e lascio che sia il metodo **`m'`** a invocare il metodo **`m`**
- Lunghe liste di parametri ↓ leggibilità
  - Se un metodo può prendersi la responsabilità di ottenere un valore, non serve passarglielo come parametro

5.9 Introduce Parameter Object

- Ho un gruppo di parametri che "stanno bene insieme" (tendono a essere passati insieme)
- → Li rimpiazzo con un oggetto
  - Es.: **`m(inizio: Date, fine: Date)`** → **`m(intervallo: Intervallo)`**
- Benefici:
  - Lunghe liste di argomenti ⇒ leggibilità ↓
  - Il raggruppamento aiuta a vedere comportamento che sta bene nella nuova classe
    - Si eliminano duplicazioni di codice…

5.10 Remove Setting Method

- Ho un attributo **`x`** a cui va assegnato un valore solo alla costruzione
- → Rimuovo il metodo **`setX`**
- → Se possibile/sensato rendo **`x final`**

5.11 Hide Method

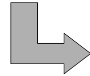
- Ho un metodo pubblico che non viene usato da nessun'altra classe
- → Lo rendo privato
- L'ambiente di programmazione potrebbe/dovrebbe evidenziare i metodi pubblici non usati…

5.12 Replace Constructor with Factory Method

- → Sostituisco il costruttore con un Factory Method

```
class Impiegato {
  public Impiegato(int tipo) {
    _tipo = tipo;
  }
}
```

O, eventualmente, sottoclasse



```
class Impiegato {
  private Impiegato(int tipo) {
    _tipo = tipo;
  }
  public static Impiegato crea(int tipo) {
    return new Impiegato(tipo);
  }
}
```

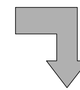

5.13 Encapsulate Downcast

- Ho un metodo che restituisce un oggetto che deve essere "downcasted" al tipo corretto
  - Es.: collections
- → Sposto il downcast dentro il metodo
  - È un metodo diverso, non sovrascrivo (più)…

5.14 Replace Error Code with Exception

- Ho un metodo che restituisce un codice speciale per indicare un errore
- → Gli faccio gettare un'eccezione

```
int preleva(int ammontare) {
  if (ammontare > _saldo)
    return -1;
  else {
    _saldo -= ammontare;
    return 0;
  }
}
```

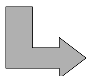


```
void  preleva(int ammontare) throws EccezioneSaldo{
  if (ammontare > _saldo)
    throw new EccezioneSaldo();
  else
    _saldo -= ammontare;
}
```


5.15 Replace Exception with Test

- In un metodo viene gettata un'eccezione su una responsabilità del chiamante
- → Modifico il chiamante per fargli fare il test
  - Progetto per contratto…

```
double getIesimo(int i) {
  try {
    return _array[i];
  } catch (ArrayIndexOutOfBoundsException e) {
    return 0;
  }
}
```



```
double getIesimo(int i) {
  if(i >= _array.length) return 0;
  return _array[i];
}
```

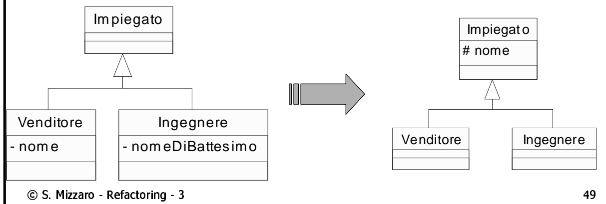

6. Gestione della generalizzazione

- Spostamenti in una gerarchia, collegamenti con i pattern di progetto

1. Pull Up Field
2. Pull Up Method
3. Pull Up Constructor Body
4. Push Down Method
5. Push Down Field
6. Extract Subclass
7. Extract Superclass
8. Extract Interface
9. Collapse Hierarchy
10. Form Template Method
11. Replace Inheritance with Delegation
12. Replace Delegation with Inheritance

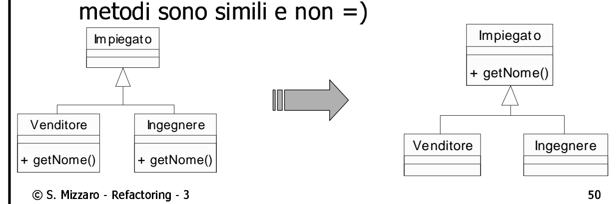
6.1 Pull Up Field

- Due sottoclassi con lo stesso attributo
- → Lo sposto nella sopraclasse
  - Anche se con nomi ≠ (purché significati =)
  - Se `private`, → `protected`



6.2 Pull Up Method

- Ho due metodi con risultati identici in due sottoclassi (o in sopraclasse e sottoclasse)
- → Li sposto nella superclasse
  - Alternativa: 6.10 Form Template Method (se i 2 metodi sono simili e non =)

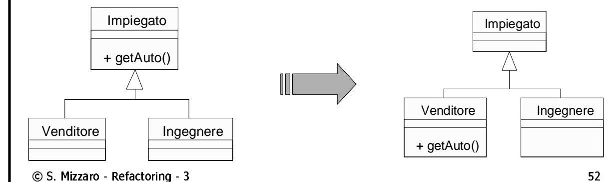


6.3 Pull Up Constructor Body

- Ho due costruttori quasi uguali in due sottoclassi
- → Creo un costruttore nella superclasse e lo chiamo (`super`) dalle sottoclassi
  - ≠ da 6.2 Pull Up Method perché i nomi dei costruttori sono ≠
  - Alternativa: 5.12 Replace Constructor with Factory Method

6.4 Push Down Method

- Ho comportamento non pertinente a una superclasse
- → Lo sposto nella/e sottoclasse/i
- Inverso di Pull Up Method



6.5 Push Down Field

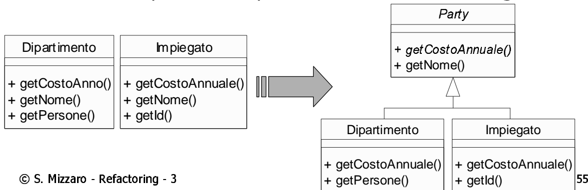
- Ho un attributo usato solo in alcune sottoclassi
- → Lo sposto in quelle sottoclassi
- Inverso di Pull Up Field

6.6 Extract Subclass

- Ho una classe con qualche caratteristica usata solo in alcune istanze
  - Classe con coesione a istanza mista (la peggiore)
- → Creo una sottoclasse per quel sottoinsieme di caratteristiche
- Alternativa: 2.1 Extract Class (delega invece di eredità)

6.7 Extract Superclass

- Ho 2 classi con caratteristiche simili
- → Creo una superclasse e vi sposto le caratteristiche comuni
  - Eventualmente rinomino, rendo astratto, sovrascrivo…
  - Uso Pull Up Field/Method/Constructor Body
- Alternativa: 2.1 Extract Class (delega)
  - e c'è sempre 6.11 Replace Inheritance with Delegation



© S. Mizzaro - Refactoring - 3 55

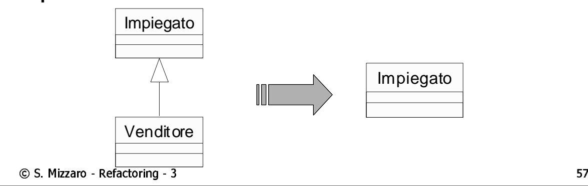
6.8 Extract Interface

- Ho molti clienti che usano solo una parte dell'interfaccia di una classe, oppure
- Due classi che hanno una parte dell'interfaccia in comune
- → Estraggo la parte comune in un'interfaccia
- Cfr. con Extract Superclass
  - Duplicazione codice, ereditarietà singola

© S. Mizzaro - Refactoring - 3 56

6.9 Collapse Hierarchy

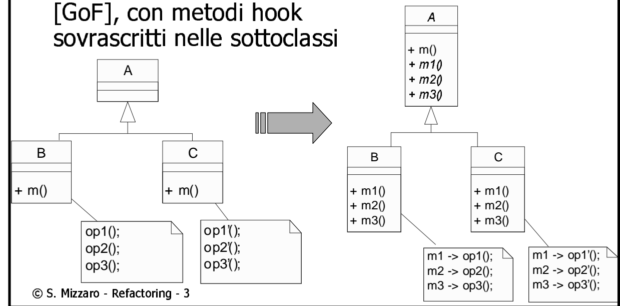
- Ho superclasse e una sottoclasse molto simili
- → Le fondo insieme
- Inverso di Extract Superclass/Subclass
- Pull Up/Push Down Field/Method possono portare a classi molto simili…



© S. Mizzaro - Refactoring - 3 57

6.10 Form Template Method

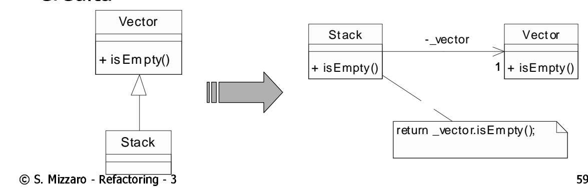
- Ho in due sottoclassi due metodi che eseguono nello stesso ordine passi simili anche se differenti
- → Costruisco nella superclasse un Template Method [GoF], con metodi hook sovrascritti nelle sottoclassi



© S. Mizzaro - Refactoring - 3

6.11 Replace Inheritance with Delegation

- Ho una sottoclasse che usa solo parte dell'interfaccia della superclasse, oppure
- sarebbe meglio se non ereditasse certi dati
- → Creo un attributo per la superclasse, modifico i metodi per far delegare e rimuovo la relazione di eredità



© S. Mizzaro - Refactoring - 3 59

6.12 Replace Delegation with Inheritance

- Ho "troppa" delegazione banale (tanti metodi semplici che delegano e basta)
- → Rendo il delegante una sottoclasse del delegato
- Inverso del precedente
- Ovviamente deve aver senso il subclassing
  - Es.: ok per `Studente` e `Persona`

© S. Mizzaro - Refactoring - 3 60

Scaletta

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