Capitolo 22
Primi esempi di refactoring

In questo capitolo mostriamo un paio di esempi di refactoring. Si tratta di toccare con mano alcuni aspetti nella trasformazione del codice. Gli esempi sono da considerare semplici perché sono automatizzabili e sono compresi nelle liste dei refactoring offerte da alcuni sistemi di sviluppo. Rappresentano comunque un buon esempio per un primo approccio al tema dei refactoring.

Aggiunta di un metodo factory

Come primo esempio consideriamo il procedimento denominato Replace Constructor with Factory Method, attraverso cui si introduce nel codice il pattern Factory Method.

Mostro questo procedimento perché ci permette di completare l'esempio proposto nella prima parte del libro. È comodo che la creazione di sottoclassi della classe astratta `TipoSci` avvenga utilizzando questo pattern.

Scopo

Isolare la creazione di oggetti in modo che questa sia completamente sotto controllo e sia possibile aggiungere operazioni alla semplice creazione.

Motivazione

Analogamente al catalogo dei pattern, anche i refactoring hanno una motivazione. Si usa introdurre un metodo factory quando l'oggetto da creare potrebbe appartenere a classi differenti, ognuna delle quali sottoclasse di un tipo comune. L'idea è quella di inserire lo switch di scelta nel metodo factory, piuttosto che nel codice dell'oggetto chiamante. Potrebbe quindi essere la continuazione del refactoring Replace type Code with Strategy/State, già visto nella prima parte del libro, quando abbiamo sostituito gli switch con la gerarchia di classi TipoSci.

Oppure, più semplicemente, si usa un metodo factory perché si vuole aggiungere un'operazione alla creazione dell'oggetto, o gestire all'interno del metodo eventuali eccezioni che potrebbero venir lanciate.

Schema di trasformazione

Il codice che segue mostra in modo schematico la trasformazione, considerando per il momento un'unica classe semplice, non una gerarchia. Supponiamo di avere la classe TipoSci, gestita da costanti, senza ancora l'ausilio di sottoclassi. Il costruttore potrebbe apparire in questo modo:

```
public TipoSci(int tipo){
  fTipo = tipo;
}
```

Senza metodo factory, il costruttore viene richiamato ogni volta che nel programma è necessario creare un nuovo oggetto di tipo TipoSci. Se utilizziamo un metodo factory, la creazione dell'oggetto viene invece isolata nel metodo stesso e si presenta in questo modo:

```
public static TipoSci create(int tipo){
  return new TipoSci(tipo);
}
```

Dove prima c'era una chiamata al costruttore, ora si dovrà avere la chiamata al metodo factory. In questo modo la chiamata al costruttore appare un'unica volta nel codice dell'intero programma.

Procedimento

Questo è il procedimento "meccanico" per realizzare in modo sicuro ogni singolo passaggio del refactoring.

- Creare un metodo factory per l'oggetto in questione e inserire nel metodo la costruzione dell'oggetto.
- Rimpiazzare nel programma tutte le chiamate al costruttore con chiamate al metodo factory.

- Compilare e eseguire i test dopo ogni modifica
- Dichiarare il costruttore privato
- Compilare

Esempio

Consideriamo l'esempio della classe TipoSci, tanto per vedere un caso semplice, cioè un caso in cui un'unica classe è coinvolta.

Questa è la situazione di partenza:

```
public class TipoSci
{
  private int fTipo;

  static final int NORMALE = 0;
  static final int CARVING = 1;
  static final int BAMBINI = 2;

  public TipoSci(int tipo){
    fTipo = tipo;
  }

  …
}
```

Vediamo ora, passo per passo, le modifiche da effettuare nel codice. Per prima cosa realizziamo il metodo factory, come mostrato in precedenza.

```
public static TipoSci create(int tipo){
  return new TipoSci(tipo);
}
```

In seguito dobbiamo rimpiazzare nel programma tutte le chiamate al costruttore con chiamate al metodo factory. Dove troviamo codice di questo tipo:

```
TipoSci sci = new TipoSci(TipoSci.BAMBINI);
```

sostituiamo la linea con la seguente:

```
TipoSci sci = TipoSci.create(TipoSci.BAMBINI);
```

A questo punto dobbiamo compilare ed eseguire i test. La compilazione non ci dice ancora molto, mentre i test verificano che la funzionalità precedente sia rimasta intatta.

In seguito, sia per evitare in futuro l'utilizzo diretto del costruttore, sia per poter verificare se abbiamo effettivamente sostituito tutte le chiamate attuali, dichiariamo il costruttore private:

```
public class TipoSci
{
  ...
  private TipoSci(int tipo){
    fTipo = tipo;
  }

  ...
}
```

Ora la semplice compilazione è sufficiente per verificare se abbiamo dimenticato chiamate dirette al costruttore. Terminata la compilazione, abbiamo terminato i passaggi di refactoring.

Vantaggi

Fin qui i vantaggi di questo refactoring non sono ancora evidenti, perché al codice di costruzione non abbiamo associato altro codice, come chiamate ad altri metodi, gestione di eccezioni o sostituzione del codice di tipo con uso di polimorfismo.

Vediamo di trattare proprio quest'ultimo caso. Supponiamo allora che le differenze tra tipi di sci non vengano gestite da un campo intero all'interno di TipoSci, ma dall'utilizzo di sottoclassi, come mostrato nell'esempio introduttivo del libro. Queste vengono create nel metodo create():

```
static TipoSci create(int tipoSci){
  switch(tipoSci){
    case NORMALE:
      return new TipoNormale ();
    case CARVING:
      return new TipoCarving ();
    case BAMBINI:
      return new TipoBambini ();
    default:
      throw new IllegalArgumentException("...");
  }
}
```

Se pensiamo all'esempio iniziale, vediamo che di nuovo abbiamo il problema dello switch. È comunque un problema relativo, perché è in ogni caso l'unico luogo in cui verrebbe usato codice condizionale, visto che è solo qui che vengono scelte le classi. Operazioni specifiche

verrebbero poi implementate nelle singole classi e discriminate attraverso lo sfruttamento del polimorfismo. Si potrebbe comunque pensare di evitarlo anche qui. Come? Ad esempio, in Java, con il meccanismo di Reflection.

Reflection

In questo caso sfruttiamo la capacità di Java di creare l'oggetto di una classe partendo dal nome della classe rappresentato in una stringa.

```
public static TipoSci create(String name){
  try{
    return (TipoSci)Class.forName(name).newInstance();
  }catch(Exception e){
    throw new IllegalArgumentException("...");
  }
}
```

La chiamata deve quindi prevedere il passaggio del nome della stringa come parametro.

```
TipoSci sci = TipoSci.create(tipoDaCreare);
```

Il grosso vantaggio nell'utilizzare un meccanismo del genere è evidentemente quello di non dover modificare il metodo `create()` nemmeno quando si aggiunge al programma una nuova sottoclasse di `TipoSci`. Lo svantaggio, per contro, è quello di non avere nessun controllo in fase di compilazione (ma questo è lo stesso discorso sui vantaggi e gli svantaggi dei linguaggi *strong typed* rispetto a quelli senza controlli di tipo).

Modifica della relazione tra classi

In questo secondo esempio consideriamo il procedimento denominato Replace Inheritance with Delegation. Pur trattando il caso della relazione tra classi, che potrebbe sembrare a prima vista qualcosa di ostico, questo refactoring è automatizzabile.

Scopo

Rimpiazzare in un programma una relazione di ereditarietà tra due classi con una relazione di composizione.

Motivazione

Quando una sottoclasse utilizza solo parte dell'interfaccia della superclasse, oppure non ha intenzione di utilizzare variabili ereditate, è la relazione di ereditarietà quella corretta?

In generale l'ereditarietà è un meccanismo semplice, che si sfrutta velocemente. Non sempre è però la soluzione migliore. Succede spesso di ereditare per comodità, ma poi, durante la *review* del codice si realizza che la maggior parte della funzionalità della superclasse non serve. Si ha quindi un'interfaccia sbagliata sulla propria classe.

Un altro caso critico è quello in cui la superclasse utilizza e carica tutta una serie di dati inutili alla sottoclasse, rallentandone la creazione (con delegation è possible gestire un meccanismo di *lazy loading*).

Sono parecchie le applicazioni che sfruttano in modo non coerente l'ereditarietà. Si può sicuramente convivere con situazioni del genere, ma si ha un codice che non rispecchia in modo esatto la situazione reale del programma. Se possibile è quindi meglio sostituirlo.

Un caso classico di utilizzo sbagliato dell'ereditarietà, citato più volte come cattivo esempio, si può trovare nelle classi della distribuzione standard di Java, a partire dalla versione 1.0. La classe Stack eredita da Vector, pur utilizzando una minima parte dell'interfaccia di Vector. Non solo, ma attraverso l'interfaccia di Vector è possible manipolare il funzionamento dello stack rendendolo del tutto inconsistente.

Questo è un esempio di sfruttamento errato dell'ereditarietà. Meglio sarebbe stato realizzare la classe Stack con un meccanismo di delega alla classe Vector, come mostrato nello schema di figura 22.1.

Usando la composizione (delegation) risulta chiaro che si utilizza solo parte del codice dell'oggetto delegato. Si ha così il pieno controllo sugli aspetti dell'interfaccia da tenere e su quelli da ignorare, perché l'interfaccia non si eredita, ma viene definita in Stack.

Il costo in più consiste nei metodi di delega, cioè i metodi che al loro interno non fanno altro che chiamare i loro corrispondenti sull'oggetto delegato (i metodi, in altre parole, che prima si ereditava direttamente). Questi metodi, proprio perché sono una semplice chiamata a un metodo dell'oggetto delegato, sono comunque molto semplici da implementare.

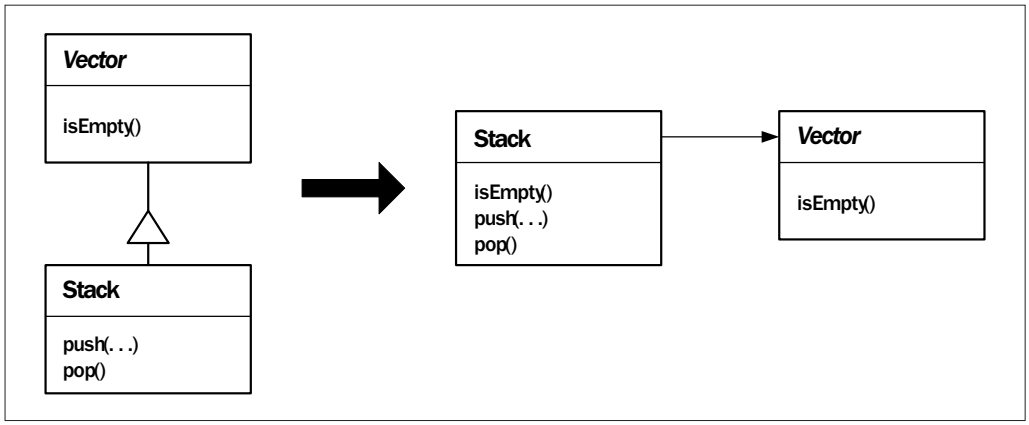

Figura 22.1 – Schema di passaggio da ereditarietà a delegation.

Procedimento

Questo è il procedimento "meccanico" per realizzare in modo sicuro ogni singolo passaggio del refactoring, così come descritto in [Fowler-1999].

- Creare un campo nella sottoclasse riferito a un'istanza di superclasse. Inizializzarlo con this.
- Cambiare ogni metodo definito nella sottoclasse in modo che ogni chiamata a un metodo ereditato utilizzi ora il campo delegato.
- Compilare ed eseguire i test.
- Togliere la specificazione di sottoclasse e rimpiazzare l'inizializzazione dell'oggetto delegato con un nuovo oggetto di superclasse.
- Aggiungere un nuovo metodo per ogni metodo della superclasse usato direttamente dall'applicazione client.
- Compilare ed eseguire i test.

Esempio

Vediamo allora l'esempio di Stack. L'implementazione presente nell'ambiente di sviluppo standard è qualcosa del genere:

```
public class Stack extends Vector
{
  public void push(Object element){
    insertElementAt(element,0);
  }

  public Object pop(){
    Object result = firstElement();
    removeElementAt(0);
    return result;
  }
}
```

Se pensiamo ora alle operazioni richieste da un oggetto client che intendesse utilizzare Stack, ci accorgeremmo che, oltre a push() e pop(), l'oggetto potrebbe richiedere isEmpty() e, ma non necessariamente, size(). Le prime due sono state implementate nella classe Stack, le altre due sono invece esempi di metodi ereditati da Vector. Se consideriamo che Vector mette a disposizione una cinquantina di altri metodi, tra cui set() e get(), che danno la possibilità di accedere

direttamente alla lista e modificare gli elementi del vettore, ben ci rendiamo conto di come l'idea di ereditare da Vector sia sbagliata.

Vediamo ora, passo per passo, le modifiche da effettuare nel codice per passare dalla relazione di ereditarietà alla delega dei compiti. Prima di tutto dobbiamo creare in Stack (sottoclasse) un campo che si riferisca a un'instanza di Vector (superclasse). Il campo va inizializzato con this.

```
private Vector fVector = this;
```

Ora dobbiamo adattare ogni metodo definito nella sottoclasse facendo in modo che ogni chiamata a metodi ereditati venga sostituita con chiamate a metodi del campo delegato.

```
public void push(Object element){
  fVector.insertElementAt(element,0);
}

public Object pop(){
  Object Result = fVector.firstElement();
  fVector.removeElementAt(0);
  return result;
}
```

A questo punto siamo in grado di compilare e verificare se i test funzionano ancora. In realtà stiamo ancora utilizzando i metodi ereditati in Stack da Vector, ma vi accediamo in modo indiretto, attraverso la variabile fVector.

Il prossimo passaggio diventa quello di togliere da Stack la specificazione di sottoclasse e rimpiazzare l'inizializzazione dell'oggetto delegato con un nuovo oggetto di tipo Vector.

```
public class Stack
{
  private Vector fVector = new Vector();
  ...
}
```

Ora Stack non eredita più l'interfaccia di Vector. Un oggetto client può quindi riferirsi a un oggetto Stack unicamente attraverso i suoi metodi di interfaccia.

Avendo tolto la gerarchia, non abbiamo più a disposizione i metodi ereditati da Vector, che ora vanno implementati. La realizzazione è però molto semplice, perché consiste unicamente nella chiamata del corrispondente metodo dell'oggetto delegato.

```
public int size(){
  return fVector.size();
}
```

```
public boolean isEmpty(){
   return fVector.isEmpty();
}
```

In questo capitolo…

In questo capitolo abbiamo visto con due esempi pratici cosa significa applicare i passaggi di refactoring. Gli esempi in questione sono da considerare semplici, perché introducono modifiche che toccano una o al massimo due classi, ma possono già avere un impatto importante nel design, come ha mostrato l'esempio del passaggio da un tipo di relazione a un'altra. I singoli passi servono a mostrare come si possa mantenere il codice il più consistente possibile, riducendo il rischio di errore, anche durante modifiche che intaccano il design.