Capitolo 20
Test-driven development

L'idea di test-driven development (TDD), cioè sviluppo a partire dal test, nasce dai movimenti di sviluppo agile e, più in particolare, dalle considerazioni che stanno alla base dell'eXtreme Programming (XP).

XP definisce una serie di buone abitudini che incoraggiano valori ritenuti importanti nello sviluppo, come feedback e semplicità. Il feedback lo si ottiene attraverso test e iterazioni di sviluppo brevi. La semplicità deriva dall'idea che il software deve eseguire solo quanto è richiesto in un dato momento, senza prevedere troppe varianti. Richieste future verranno integrate solo successivamente nel codice esistente attraverso iterazioni di refactoring.

XP e la programmazione in coppia

Tra le cosiddette "buone abitudini" che vengono proposte in XP e in altre metodologie agili, la programmazione in coppia (*pair programming)* è senza dubbio una delle più discusse.

Nessuno mette in dubbio i vantaggi dello sviluppare in coppia (in due, seduti davanti a unico computer) da un punto di vista del programmatore, cioè dell'impiegato. Il punto cruciale è però la produttività. L'imprenditore, il capo progetto si chiedono: in due si produce almeno il doppio di quanto produrrebbero le due persone separate? Se no, quali sono i vantaggi per l'azienda?

Esistono articoli e libri sull'argomento (un autore su tutti, naturalmente, Kent Beck). Vorrei qui semplicemente parlare della mia esperienza. Prima di tutto non è mai a vantaggio dell'azienda che sia un unico impiegato ad avere il know-how sul codice di un'applicazione, per evidenti motivi. Già questo basterebbe a spiegare la necessità di lavorare in coppia almeno su alcune parti molto critiche dell'applicazione. Non è però tutto. Lavorando in due si anticipano discussioni, si valuta insieme in modo critico il design e il codice dell'applicazione, si prendono decisioni più coraggiose. Il risultato è un codice più stabile perché è stato sviluppato in coppia e ha quindi già superato la critica incrociata. Le modifiche da apportare, magari dopo aver discusso la soluzione con altri membri del team, saranno sicuramente minori.
Inoltre la leggibilità del codice sarà maggiore, perché non una, ma due persone l'hanno valutata durante lo sviluppo. Se uno dei due sviluppatori fosse tentato di scrivere un metodo troppo lungo, un'espressione poco chiara o una classe che non si integra nel design, ecco che l'altro si opporrebbe, perché non gli sono chiari i motivi di certe scelte. Una migliore leggibilità del codice è la premessa per una buona manutenzione.

La teoria

Kent Beck, il maggiore promotore e divulgatore di XP, ha estratto l'essenza delle buone abitudini maturate in progetti gestiti secondo i metodi di eXtreme Programming dal punto di vista del test e le ha pubblicate nel libro *Test-Driven Development* [Beck-2003].

Il punto principale di TDD consiste nella capacità di estrarre la funzionalità esatta richiesta al software, piuttosto che iniziare lo sviluppo cercando di immaginarsi cosa un utente si aspetti dal programma. Il metodo usato sembra poco intuitivo, ma ci si abitua molto velocemente. Si inizia a scrivere il codice client come se il codice che dobbiamo sviluppare esistesse già, immaginandoci un utilizzo molto semplice.

Scriviamo quindi prima la parte di codice che utilizzerà il nostro programma. L'abbiamo visto nel capitolo precedente con l'utilizzo delle diverse varianti di `add()`. Questo rappresenta il test, che ci aiuta a modellare in modo programmatico la migliore API per i bisogni del codice client. In altre parole partiamo dai casi d'uso. Nei test possiamo già inserire degli `assert()`, dato che sappiamo esattamente cosa il codice utilizzatore si attende dal sistema. Naturalmente il codice non potrà essere compilato, visto che vengono usati oggetti e chiamati metodi non ancora esistenti.

Il passaggio successivo consiste nello scrivere il codice minimo indispensabile affinché il test possa almeno essere compilato (qui può giocare un ruolo importante il sistema di sviluppo usato: sistemi come IntelliJ IDEA o Eclipse, tanto per citare i primi che hanno introdotto questa possibilità, sono in grado di generare le classi e i metodi mancanti). Basta la compilazione, per permettere di eseguire il test e verificare che non passa (evidentemente, visto che la funzionalità non è ancora stata implementata).

Solo a questo punto si scrive il codice applicativo che permette di soddisfare il test. Appena il test funziona correttamente, bisogna intervenire nel codice (refactoring) per renderlo più generico, cioè non solo in grado di far passare i casi di test previsti, ma in grado di coprire il dominio del problema che ci si aspetta debba risolvere.

Questo, in sintesi, deve diventare il ritmo di sviluppo in TDD:

1. scrivi il test;
2. scrivi il codice in grado di far superare il test;
3. esegui refactoring per generalizzare la soluzione.

Scrivere test prima di scrivere il codice dell'applicazione (l'abbiamo già sperimentato con le chiamate di `add()`) permette di concentrare la mente sulla funzionalità essenziale dell'applicazione. Anche il processo di sviluppo si limiterà a soddisfare le richieste del test, così che il sistema che si trova in fase di evoluzione sarà in grado di eseguire unicamente quanto gli è richiesto, niente di più. Nuove funzionalità vanno aggiunte solo quando si rendono necessarie, sempre precedute ("condotte", *driven*) dai test corrispondenti. Naturalmente, anche con TDD, l'intera sequenza di test va eseguita regolarmente per verificare eventuali anomalie durante la fase di refactoring e di aggiunta di nuove funzionalità.

La pratica

La differenza tra quanto vedremo ora e quanto già visto nel capitolo precedente consiste nel fatto che qui si tratta di applicare in modo sistematico l'idea di scrivere il test prima del codice dell'applicazione. Ogni funzionalità va preceduta da un test. Anche questo, come ogni metodo che arriva da Kent Beck, va portato all'estremo.

Problema

Sfruttiamo il fatto che abbiamo lavorato da un paio di capitoli sul problema delle valute, per inserire una nuova funzionalità al sistema. Vogliamo aggiungere la moltiplicazione. Ci serve, ad esempio, a calcolare il prezzo globale di un portfolio di fondi di investimento, in cui ogni singola parte viene espressa in `IMoney`.

Procedimento

Pur desiderando aggiungere la funzionalità in interfaccia (`IMoney`), ci accorgiamo subito che la stessa avrà due casi ben distinti: l'implementazione in `Money` e quella in `MoneyBag`. Separiamo quindi i due casi e iniziamo da quello più semplice, cioè la moltiplicazione in `Money`.

E adesso? Invece di andare nella classe `Money` e vedere come aggiungere la prima funzionalità della lista, iniziamo dal test. È scrivendo il test che riusciamo a immaginare meglio l'interfaccia della nostra operazione.

```
public void testMultiplication(){
  Money money = new Money(10, "CHF");
```

```
    money.multiply(3);
    assertEquals(30,money.getAmount());
}
```

Ci sarebbero alcune considerazioni da fare. Una di queste è legata all'effetto collaterale che abbiamo introdotto implicitamente, visto che il test presuppone che il singolo oggetto money venga modificato da multiply(), metodo di tipo void. Vedremo fra un attimo questo aspetto.

Al momento abbiamo un test che non funziona (anzi, non viene nemmeno compilato) e vogliamo arrivare al più presto al suo corretto funzionamento. Il problema è il metodo multiply() che non esiste ancora. Dobbiamo crearlo (un buon sistema di sviluppo ci genererebbe con un unico click un primo stub); per il momento è più che sufficiente un metodo vuoto nella classe Money:

```
public void multiply(int factor){
}
```

Ora siamo in grado di compilare. Otteniamo un errore di test, del resto ampiamente previsto, ma anche questo in TDD è da considerare un passo avanti. Ora trasformare la barra da rossa a verde diventa il nostro prossimo scopo.

Il metodo più semplice per farlo è inserire nel codice il risultato atteso:

```
public void multiply(int factor){
    fAmount = 30;
}
```

A inserire il valore 30 come costante sembra di fare qualcosa di poco corretto, ed è vero, ma è il primo passaggio verso la soluzione finale. Abbiamo infatti scritto il test (punto 1 del ritmo di sviluppo), abbiamo aggiunto del codice in grado di far passare il test (punto 2), ora dobbiamo generalizzare (punto 3). Generalizzare, nel nostro caso, significa fare in modo che ogni chiamata a multiply() calcoli il valore corretto. In questo caso non c'è molto da ragionare. Siamo in grado di passare dalla soluzione specifica a quella generale senza troppi problemi.

```
public void multiply(int factor){
    fAmount = fAmount * factor;
}
```

L'esempio è molto semplice. Questi passaggi sembrano inutili per coloro che (tutti) sarebbero in grado di arrivare direttamente alla soluzione finale. L'idea di TDD non è però quella di obbligare lo sviluppatore a passare dalle soluzioni intermedie. L'idea è invece quella di permettergli di farlo quando lo ritiene necessario. I casi pratici sono più complessi di questo. Se si è in grado con un unico passaggio di arrivare alla soluzione finale, tanto meglio. Quando però non si è in grado, meglio tornare ai piccoli passi, procedere con attenzione e poi riprendere il proprio ritmo.

Prima strategia: implementazione "fasulla"

Abbiamo visto con questo esempio una prima strategia utilizzata per arrivare al test corretto: implementazione "fasulla", cioè inserimento di una costante e graduale passaggio verso l'utilizzo di variabili per ottenere il codice corretto. L'implementazione fasulla serve ad arrivare velocemente alla barra verde. Il refactoring che segue, che può comprendere più passaggi, ha invece lo scopo di ottenere la versione definitiva in modo più semplice. Con le ulteriori modifiche che apporteremo al codice nei prossimi paragrafi, vedremo altre due strategie usate in TDD.

Nuovi passi

Abbiamo visto come procedere. Ora possiamo continuare con altri compiti oppure adattare ciò che non ci convinceva completamente, come l'idea che multiply() fosse di tipo void. È il modo corretto di lavorare con TDD. Si parte con un obiettivo, si fa in modo che funzioni, si generalizza e poi si torna a controllare se il codice realizzato è veramente pulito. Se ci pensiamo bene è l'esatto opposto dell'approccio orientato all'architettura: lì prima si pensa al design pulito, poi si implementa la funzionalità.

Effetti collaterali

La cosa strana nella soluzione precedente consiste nel fatto che l'oggetto Money viene modificato. Aggiungiamo codice al test per verificare questo effetto collaterale.

```
public void testMultiplication(){
  Money money = new Money(10, "CHF");
  money.multiply(3);
  assertEquals(30, money.getAmount());
  money.multiply(2);
  assertEquals(20, money.getAmount());
}
```

Infatti il test non passa. Il primo multiply() modifica Money, così che il secondo genera una soluzione diversa da quella attesa. Una codifica più pulita del codice richiederebbe la restituzione di un nuovo oggetto Money da parte di multiply(), e quindi, indirettamente, una modifica dell'interfaccia. Dobbiamo perciò prima modificare il test.

```
public void testMultiplication(){
  Money money = new Money(10, "CHF");
  Money result = money.multiply(3);
  assertEquals(30, result.getAmount());
  result = money.multiply(2);
```

```
    assertEquals(20, result.getAmount());
    assertEquals(10, money.getAmount());
}
```

Oltre ai risultati delle operazioni, verifichiamo anche che l'oggetto Money iniziale abbia lo stesso valore che aveva in partenza. Anche qui otteniamo subito un problema di compilazione, perché il nuovo test si basa su un'interfaccia non esistente. Questo ci obbliga a modificare subito la dichiarazione di multiply():

```
public Money multiply(int factor){
    return new Money(getAmount() * factor, getCurrency());
}
```

Abbiamo in questo modo pulito il codice, evitando effetti collaterali durante la chiamata dell'operazione di moltiplicazione.

Esattamente come fatto in precedenza per l'implementazione di add(), dovremo preoccuparci della generalizzazione dell'interfaccia per moltiplicazioni con Money e moltiplicazioni con MoneyBag. Ci occuperemo di questo al momento di realizzare la moltiplicazione in MoneyBag.

Seconda strategia: realizzazione diretta

L'ultima implementazione di multiply() ci mostra la seconda strategia usata in TDD per passare dal test all'implementazione: la realizzazione diretta. Si usa sia in casi semplici, come questo, sia in casi più complessi, se ci si sente sicuri. Se poi il test dimostra che questa non era la soluzione corretta, allora significa che vale la pena ricominciare, suddividere il compito in passaggi più brevi, usando l'implementazione fasulla, strategia vista in precedenza.

Controllo di uguaglianza

Nello scrivere il test abbiamo unicamente controllato la parte intera (amount) di Money. In realtà sappiamo benissimo che abbiamo già a disposizione il metodo equals(), implementato in un capitolo precedente. Vale la pena utilizzarlo, perché esegue un controllo più preciso del risultato, considerando anche il tipo di valuta.

```
public void testMultiplication(){
    Money money = new Money(10, "CHF");
    Money result = money.multiply(3);
    assertEquals(new Money(30, "CHF"), result);
    result = money.multiply(2);
    assertEquals(new Money(20, "CHF"), result);
    assertEquals(new Money(10, "CHF"), money);
}
```

Terza strategia: triangolazione

Kent Beck definisce *triangulation* la terza strategia. Per mostrarla torniamo all'inizio dell'implementazione di multiply(), pensando però già di restituire un nuovo oggetto Money (senza effetti collaterali). Questo sarebbe il primo test proposto:

```
public void testMultiplication(){
  Money money = new Money(10, "CHF");
  assertEquals(new Money(30, "CHF"), money.multiply(3));
}
```

Supponiamo ora che la prima implementazione di multiply() venga realizzata con l'utilizzo di una costante.

```
public Money multiply(int factor){
  return new Money(30, "CHF");
}
```

Come fare a passare alla realizzazione definitiva? La strategia usata in precedenza è stata di passare direttamente dalla versione fasulla, contenente la costante, alla versione finale. La triangolazione parte invece di nuovo dal test. L'idea di base consiste nel creare un nuovo test, o aggiungere un nuovo assert nel test esistente, che dimostri l'inesattezza dell'implementazione attuale.

```
public void testMultiplication(){
  Money money = new Money(10, "CHF");
  assertEquals(new Money(30, "CHF"), money.multiply(3));
  assertEquals(new Money(50, "CHF"), money.multiply(5));
}
```

Il secondo esempio serve a generalizzare il test ed eventualmente a dimostrare, come in questo caso, che il programma non è generico. Ora si tratta di modificare il codice di conseguenza.

```
public Money multiply(int factor){
  return new Money(getAmount() * factor, getCurrency());
}
```

Abbiamo così visto anche la triangolazione, la terza strategia usata in TDD. Questa strategia prevede un passaggio in più, che serve sia a dimostrare il problema, sia a confermare la soluzione successiva e rendere lo stesso test più solido. Se la soluzione è ovvia, non serve passare dalla triangolazione. Quando però ci sono dubbi riguardo alla realizzazione finale, la triangolazione permette di osservare il problema da un diverso punto di vista.

Valute diverse

La moltiplicazione con valute diverse è relativamente semplice, in ogni caso più semplice della somma, trattata nel capitolo 19. Iniziamo comunque a preparare un test adatto.

```
public void testMoneyBagMultiplication(){
  MoneyBag moneyBag = new MoneyBag(new Money[]{new Money(10,"EUR"), new Money(20,"CHF")});
  MoneyBag expected = new MoneyBag(new Money[]{new Money(30,"EUR"), new Money(60,"CHF")});
  assertEquals(expected, moneyBag.multiply(3));
}
```

Il test non può essere compilato per la mancanza del metodo multiply(), per ora presente solo in Money. Trattandosi di una semplice operazione, proviamo a eseguire il passaggio in modo diretto, implementando subito la funzionalità per la moltiplicazione con valute multiple.

```
class MoneyBag extends AbstractMoney
{
  …
  public MoneyBag multiply(int factor){
    List tempResults = new ArrayList();
    Iterator iter = getMoneyList().iterator();
    while(iter.hasNext()){
      Money money = (Money)iter.next();
      tempResults.add(money.multiply(factor));
    }
    return new MoneyBag((Money[])tempResults.toArray(new Money[tempResults.size()]));
  }
}
```

Generalizzazione

Ora ci manca un unico punto: la generalizzazione sull'interfaccia IMoney, esattamente come già fatto per l'operazione add(). Vogliamo sfruttare il fatto che entrambe le classi coinvolte sono parte della stessa gerarchia. Questo significa che la chiamata multiply() deve poter essere inviata a un oggetto generico di tipo IMoney.

Iniziamo ad adattare il primo test, quello sulle valute semplici. Si tratta di modificare la dichiarazione delle variabili da Money (classe concreta) a IMoney (interfaccia della gerarchia):

```
public void testMultiplication(){
  IMoney money = new Money(10, "CHF");
  IMoney result = money.multiply(3);
  assertEquals(new Money(30,"CHF"), result);
```

```
    result = money.multiply(2);
    assertEquals(new Money(20,"CHF"), result);
    assertEquals(new Money(10,"CHF"), money);
  }
```

In questo modo generiamo un problema di compilazione, perché multiply() esiste in Money e MoneyBag, ma non è stato dichiarato nella loro interfaccia IMoney. Il primo passo consiste nell'adattare IMoney, inserendo la dichiarazione mancante, accanto al già presente metodo add().

```
public interface IMoney
{
  public IMoney add(IMoney aMoney);
  public IMoney multiply(int factor);
}
```

Questo passaggio, però, da solo non basta. Dichiarando multiply() in questo modo, infatti, definiamo IMoney come tipo di return. Questa dichiarazione entra in conflitto con i due metodi concreti già implementati, che restituiscono elementi di tipo Money e MoneyBag. Dobbiamo allora riprendere i due metodi e modificarli in modo che anche loro restituiscano un oggetto generico.

```
class Money extends AbstractMoney
{
  …
  public IMoney multiply(int factor){
    return new Money(getAmount() * factor, getCurrency());
  }
}

class MoneyBag extends AbstractMoney
{
  …
  public IMoney multiply(int factor){
    List tempRes = new ArrayList();
    Iterator iter = getMoneyList().iterator();
    while(iter.hasNext()){
      Money money = (Money)iter.next();
      tempRes.add(money.multiply(factor));
    }
    return new MoneyBag((Money[])tempRes.toArray(new Money[tempRes.size()]));
  }
}
```

In questo caso, essendoci unicamente un parametro di tipo intero, non abbiamo i problemi incontrati nella realizzazione di `add()`. Qui, infatti, il dispatching è semplice e si applica unicamente sull'oggetto che esegue la chiamata.

In questo capitolo…

Iniziare lo sviluppo di una funzionalità dal test permette di immaginarsi e utilizzare la API di un programma ancora prima che questa venga realizzata. Avevamo in parte già usato questo metodo nel capitolo 19.

La novità di questo capitolo riguarda la sistematicità dell'approccio. Con il metodo TDD si inizia sempre dal test. Si richiama la funzionalità prima di implementarla. Questo permette di concentrarsi sulla soluzione più essenziale, che poi viene generalizzata in un secondo tempo. Con TDD portiamo all'estremo ciò che avevamo denominato in precedenza "design by testing".