TDD

Release 2025

Grzegorz Gazowski

(Traduzione: Baldassarre Cesarano)

08 dic 2025

Contenuti:


Grzegorz Gałęzowski
Test-Driven Development
Extensive Tutorial

CAPITOLO 1

Titolo

CAPITOLO 2

Premessa

CAPITOLO 3

Dediche

Ad Deum qui laetificat iuventutem meam. [a Dio che allieta la mia giovinezza.]

Alla mia amata moglie Monika e al nostro adorabile figlio Daniel.

CAPITOLO 4

Grazie!

Vorrei ringraziare le seguenti persone (elencate in ordine alfabetico per nome) per preziosi feedback, suggerimenti, correzioni di errori di battitura e altri contributi:

- Brad Appleton
- Borysaw Bobulski
- Chris Kucharski
- Daniel Dec
- Daniel oopa (cover image)
- Donghyun Lee
- ukasz Maternia
- Marek Radecki
- Martin Moene
- Michael Whelan
- Polina Kravchenko
- Rafa Bigaj
- Reuven Yagel
- Rémi Goyard
- Robert Pajk
- Wiktor onowski

Questo libro non è affatto originale. Presenta vari argomenti che altri hanno inventato e io ho solo raccolto. Pertanto, vorrei anche ringraziare i miei mentori e le autorità sullo sviluppo test-driven e sulla progettazione orientata agli oggetti da cui ho acquisito la maggior parte delle mie conoscenze (elencati in ordine alfabetico per nome):

- Amir Kolsky
- Dan North
- Emily Bache
- Ken Pugh

- Kent Beck
- Mark Seemann
- Martin Fowler
- Nat Pryce
- Philip Schwarz
- Robert C. Martin
- Scott Bain
- Steve Freeman

CAPITOLO 5

I codici di esempio

5.1 Note per gli utenti di C#

Il linguaggio preferito per i codici degli esempi è il C#, tuttavia, sono state fatte alcune eccezioni alle tipiche convenzioni del codice C#.

5.1.1 Eliminazione della "I" dai nomi delle interfacce

Non sono un grande fan dell'utilizzo di "IQualcosa" come nome dell'interfaccia, quindi ho deciso di eliminare la "I" anche se la maggior parte degli sviluppatori C# se lo aspetta. Spero che si possa perdonare.

5.1.2 C# idiomatico

La maggior parte del codice contenuto in questo libro non è C# idiomatico. Ho cercato di evitare proprietà, eventi e funzionalità più moderne. Il mio obiettivo è consentire agli utenti di altri linguaggi (in particolare Java) di trarre vantaggio dal libro.

5.1.3 Uso del carattere di sottolineatura nei nomi dei campi

Ad alcune persone piace, ad altre no. Ho deciso di attenermi alla convenzione di inserire un underscore (_) prima del nome di un campo di classe.

5.2 Note per gli utenti Java

Il linguaggio scelto per gli esempi di codice è il C#. Detto questo, volevo che il libro fosse il più possibile indipendente dalla tecnologia, per consentire soprattutto ai programmatori Java approfittarne. Ho provato a utilizzare un numero minimo di funzionalità specifiche di C# e in diversi punti ho anche fatto commenti mirati agli utenti Java per facilitarli. Tuttavia, ci sono alcune cose che non ho potuto evitare. Ecco perché ho scritto un elenco con molte delle differenze tra Java e C# utile soprattutto agli utenti Java durante la lettura del libro.

5.2.1 Convenzioni sui nomi

La maggior parte dei linguaggi ha le proprie convenzioni sulla nomenclatura. Ad esempio, in Java, il nome di una classe è scritto in "*Pascal case*" (ad es. `UserAccount`), metodi e campi sono scritti in "*camel case*", ad es. `payTaxes` e costanti/campi di sola lettura sono generalmente scritti in maiuscolo e con l'underscore (ad esempio `CONNECTED_NODES`).

Il C# il "*Pascal case*" sia per le classi che per i metodi (ad esempio `UserAccount`, `PayTaxes`, `ConnectedNodes`). Per i campi esistono diverse convenzioni. Ho scelto quello che inizia con l'underscore (ad esempio `_myDependency`). Ci sono altre piccole differenze, ma queste sono quelle che si incontreranno abbastanza spesso.

5.2.2 La parola chiave `var`

Per brevità, ho scelto di utilizzare la parola chiave `var` negli esempi. Questa serve come inferenza automatica del tipo, ad es.

```
var x = 123; //x inferred as integer
```

Naturalmente, questa non è assolutamente un input dinamico: tutto viene risolto in fase di compilazione.

Un'altra cosa: `var` è utilizzabile solo quando è possibile dedurre il tipo, quindi occasionalmente si vedranno dichiarare esplicitamente i tipi come in:

```
List<string> list = null; //list cannot be inferred
```

5.2.3 `string` come parola chiave

Il C# ha un tipo `String`, come Java. Tuttavia, esso permette di scriverlo come parola chiave, ad es. `string` invece di `String`. Questo è solo "zucchero sintattico" utilizzato per default dalla comunità C#.

5.2.4 Attributi invece di annotazioni

In C# gli attributi vengono utilizzati per lo stesso scopo delle annotazioni in Java. Quindi, ogni volta che si vede:

```
[Whatever]
public void doSomething()
```

si deve pensare a:

```
@Whatever
public void doSomething()
```

5.2.5 `readonly` e `const` invece di `final`

Laddove Java utilizza final per le costanti (insieme a `static`) e i campi di sola lettura, C# utilizza due parole chiave: `const` e `readonly`. Senza entrare nei dettagli, ogni volta che si vedono cose come:

```
public class User
{
  // a constant with literal:
  private const int DefaultAge = 15;

  // a "constant" object:
  private static readonly TimeSpan DefaultSessionTime
    = TimeSpan.FromDays(2);

  // a read-only instance field:
  private readonly List<int> _marks = new List<int>();
}
```

si deve pensare a:

```
public class User {
  //a constant with literal:
  private static final int DEFAULT_AGE = 15;
```

(continues on next page)

(continua dalla pagina precedente)

```
  //a "constant" object:
  private static final Duration
    DEFAULT_SESSION_TIME = Duration.ofDays(2);

  // a read-only instance field:
  private final List<Integer> marks = new ArrayList<>();
}
```

5.2.6 Una `List<T>`

Gli utenti Java, devono considerare che in C# `List<T>` non è un'interfaccia, ma una classe concreta. Viene generalmente utilizzata laddove si utilizzerebbe un `ArrayList`.

5.2.7 I generici

Una delle maggiori differenze tra Java e C# è il modo in cui trattano i generici. Innanzitutto, C# consente di utilizzare tipi primitivi in dichiarazioni generiche, quindi si può scrivere `List<int>` in C# mentre in Java si deve scrivere `List<Integer>`.

L'altra differenza è che in C# non è prevista la cancellazione [erasure] dei tipi come in Java. Il codice C# conserva tutte le informazioni generiche in fase di esecuzione. Ciò influisce sul modo in cui la maggior parte delle API generiche vengono dichiarate e usate in C#.

Definizione e creazione di una classe generica in Java e C# sono più o meno le stesse. Cè però una differenza a livello di metodo. Un metodo generico in Java è tipicamente scritto come:

```
public <T> List<T> createArrayOf(Class<T> type) {
    ...
}
```

e chiamato così:

```
List<Integer> ints = createArrayOf(Integer.class);
```

mentre in C# lo stesso metodo verrebbe definito come:

```
public List<T> CreateArrayOf<T>()
{
    ...
}
```

e chiamato così:

```
List<int> ints = CreateArrayOf<int>();
```

Queste differenze sono visibili nella progettazione della libreria utilizzata in questo libro per generare i dati di test. Mentre nella versione C#, si generano dati di test scrivendo:

```
var data = Any.Instance<MyData>();
```

la versione Java della libreria viene utilizzata in questo modo:

```
MyData data = Any.instanceOf(MyData.class);
```

CAPITOLO 6

Parte 1: Solo le nozioni di base

{{keyToDo}} **Status:** stabile. Questo capitolo riceverà principalmente correzioni di bug e modifiche estetiche.

Senza entrare molto in dettaglio, come applicare il TDD ai sistemi object-oriented in cui più oggetti collaborano (che è un argomento della parte 2), viene presentata la filosofia e le pratiche basilari del TDD. In termini di progettazione, la maggior parte degli esempi riguarderà i metodi di test di un singolo oggetto. L'obiettivo è quello di concentrarsi sul nocciolo del TDD prima di passare alle sue applicazioni specifiche e introdurre man mano alcuni concetti in modo facile da comprendere.

Dopo aver letto la parte 1, si sarà in grado di sviluppare in modo efficace classi che non hanno dipendenze da altre classi (e dalle risorse del sistema operativo) utilizzando il TDD.

CAPITOLO 7

Motivazione: il primo passo per imparare il TDD

Sto scrivendo questo libro perché sono un entusiasta del Test-Driven Development (TDD). Credo che il TDD rappresenti un notevole miglioramento rispetto ad altre metodologie di sviluppo software utilizzate per fornire software di qualità. Credo anche che questo sia vero non solo per me, ma per molti altri sviluppatori di software. Ciò fa sorgere la domanda: perché non ci sono più persone che imparano e utilizzano TDD come metodo di distribuzione del software? Nella mia vita professionale, non ho mai visto un tasso di adozione abbastanza alto da giustificare l'affermazione che il TDD sia ormai la pratica principale.

C'è da rispettare chi decide di prendere in mano un libro, piuttosto che basare la comprensione del TDD sulle leggende metropolitane e la propria immaginazione. Sono onorato e felice che sia stato scelto questo, non importa se sia il primo libro sul TDD o uno dei tanti letti per i propri studi. Per quanto spero che questo libro venga letto da cima a fondo, sono consapevole che questo non sempre accade. Questo mi fa venir voglia di porre una domanda importante che potrebbe aiutare a decidere se continuare a leggere: perché imparare il TDD?

Mettendo in discussione la motivazione, non cerco di scoraggiare la lettura di questo libro. Piuttosto, vorrei che si tenesse presente l'obiettivo da raggiungere leggendolo. Nel corso del tempo, ho notato che alcuni di noi (me compreso) potrebbero pensare che ci sia *bisogno* di imparare qualcosa (invece di *voler* imparare qualcosa) per vari motivi, come ottenere una promozione sul lavoro, ottenere un certificato, aggiungere qualcosa al nostro CV, o semplicemente "rimanere aggiornato" con le ultime novità. Sfortunatamente, la mia osservazione è che il Test-Driven Development tende a rientrare in questa categoria per molte persone. Tale motivazione può essere difficile da sostenere a lungo termine.

Unaltra fonte di motivazione potrebbe essere quella di immaginare il TDD come qualcosa che non è. Alcuni di noi potrebbero avere solo una vaga conoscenza di quali siano i costi e i benefici reali del TDD. Sapendo che il TDD è apprezzato e lodato dagli altri, possiamo concludere che deve essere positivo anche per noi. Potremmo avere una vaga comprensione delle ragioni, come ad esempio "il codice sarà più testato". Dato che non conosciamo il vero "perché" del TDD, potremmo inventare alcune ragioni per praticare lo sviluppo del "test-first", come "per garantire che i test siano scritti per tutto". Non fraintendetemi, queste affermazioni potrebbero essere parzialmente vere, ma omettono gran parte dell'essenza del TDD. Se il TDD non porta i benefici che immaginiamo possa portare, la delusione potrebbe insinuarsi. Ho sentito professionisti delusi dire "Non ho bisogno del TDD, perché ho bisogno di test che mi diano fiducia in un ambito più ampio" o "Perché ho bisogno di unit test[1] quando ho già i test di integrazione, gli smoke test, sanity test, exploration test, ecc...?". Molte volte ho visto il TDD abbandonato prima ancora che fosse compreso.

Imparare il TDD è una priorità? Sei determinato a provarlo e ad impararlo? Se non lo sei, ehi, ho sentito che la nuova serie del "Trono di Spade" è in TV, perché non dargli un'occhiata invece? Ok, sto solo scherzando, ma come dicono alcuni, il TDD è "facile da imparare, difficile da padroneggiare"[2], quindi sarà difficile senza un po' di coraggio. Soprattutto perché

[1] A proposito, TDD non riguarda solo i test unitari, a cui arriveremo alla fine.

[2] Non so chi l'ha detto per primo, ho cercato sul web e l'ho trovato in pochi posti in cui nessuno degli scrittori ha dato credito a qualcun altro per questo, quindi ho deciso di menzionare semplicemente che non sono stato io a coniare questa frase.

ho intenzione di introdurre i contenuti in modo lento e graduale in modo che si possa ottenere una spiegazione migliore di alcune pratiche e tecniche.

7.1 A cosa assomiglia il TDD

A me e a mio fratello piaceva giocare ai videogiochi da bambini, uno dei più memorabili è stato Tekken 3, un torneo giapponese di combattimenti per la Playstation della Sony. Completare il gioco con tutti i guerrieri e sbloccare tutti i bonus nascosti, i minigiochi, ecc. ha richiesto circa un giorno. Alcuni potrebbero dire che da allora il gioco non ha più avuto nulla da offrire. Perché allora ci abbiamo dedicato più di un anno?

È perché ogni combattente nel gioco aveva molte combinazioni, calci e pugni che potevano essere mischiati in vari modi. Alcuni di essi erano utilizzabili solo in determinate situazioni, altri erano qualcosa che potevo lanciare al mio avversario quasi in qualsiasi momento senza correre il rischio di essere esposto a contrattacchi. Potevo fare un passo laterale per eludere gli attacchi del nemico e, soprattutto, potevo calciare un altro combattente in aria dove non potevano bloccare i miei attacchi e potevo sferrare alcuni begli attacchi contro di loro prima che cadessero. Queste tecniche volanti erano chiamate "juggles". Alcune riviste pubblicavano elenchi di nuove juggles ogni mese ed è rimasto di moda nella comunità dei giocatori per oltre un anno.

Sì, Tekken era facile da imparare: potevo dedicare un'ora ad allenare le mosse principali di un personaggio e poi essere in grado di "usare" questo personaggio, ma sapevo che ciò che mi avrebbe reso un grande combattente era l'esperienza e la conoscenza su quali tecniche erano rischiose e quali no, quali potevano essere usate in quali situazioni, quali, se usate una dopo l'altra, davano all'avversario poche possibilità di contrattaccare, ecc. Non c'è da stupirsi che presto sorsero molti tornei, dove i giocatori potevano scontrarsi per la gloria, la fama e le ricompense. Ancora oggi si possono guardare alcune di quelle vecchie partite su YouTube.

TDD è come Tekken. Probabilmente hai sentito il mantra "red-green-refactor" o il consiglio generale "scrivi prima il test, poi il codice", forse hai anche fatto alcuni esperimenti per conto tuo cercando di implementare un algoritmo di bubble sort o altri semplici cose iniziando con un test. Ma è come esercitarsi in Tekken provando ogni mossa da sola su un avversario

fittizio, senza il contesto di problemi del mondo reale che rendono il combattimento impegnativo. E anche se penso che tali esercizi siano molto utili (in effetti, ne faccio molti), trovo un immenso vantaggio anche nel comprendere il quadro più ampio sull'uso del TDD nel mondo reale.

Alcune persone con cui parlo del TDD riassumono ciò che dico loro in questo modo: "Questo è demotivante: ci sono così tante cose a cui devo prestare attenzione, che non mi viene voglia di iniziare!". Tranquillo, niente panico: ricorda la prima volta che hai provato ad andare in bicicletta: potevi non conoscere le norme del traffico e e non seguire i segnali stradali, ma questo non ti ha tenuto lontano, vero?

Trovo che il TDD sia molto eccitante e mi renda entusiasta anche nello scrivere codice. Qualcuno della mia età pensa già di sapere tutto sulla programmazione, si annoiano e non vedono l'ora di passare alla gestione, ai requisiti o all'analisi aziendale, ma ehi! Ho una nuova serie di tecniche che rendono di nuovo impegnativa la mia carriera di programmatore! Ed è un'abilità che posso applicare a molte tecnologie e linguaggi diversi, rendendomi uno sviluppatore complessivamente migliore! Non è qualcosa a cui vale la pena puntare?

7.2 Cominciamo!

In questo capitolo ho cercato di provocare e riconsiderare l'atteggiamento e la motivazione. Se si è ancora determinati a imparare il TDD leggendo questo libro, cosa che spero si faccia, allora mettiamoci al lavoro!

CAPITOLO 8

Gli strumenti essenziali

Avete mai visto Karate Kid, sia la vecchia versione che quella nuova? La cosa che hanno in comune è che quando il ragazzo "[kid]" inizia a imparare il karate (o il kung-fu) dal suo maestro, gli viene assegnato un compito basilare e ripetitivo (come togliersi una giacca e indossarla di nuovo), senza sapere ancora dove lo avrebbe portato. Oppure nel primo film di Rocky (sì, quello con Sylvester Stallone), in cui Rocky insegue un pollo per migliorare la sua agilità.

Quando ho provato a imparare a suonare la chitarra ho trovato due consigli sul web: il primo era quello di iniziare padroneggiando un singolo brano difficile. Il secondo era suonare con una sola corda, imparare a farla suonare in modi diversi e provare a suonare alcune melodie a orecchio solo con questa corda. Devo dire che il secondo consiglio ha funzionato meglio?

Onestamente, potrei tuffarmi direttamente nelle tecniche fondamentali del TDD, ma sento che sarebbe come salire su un ring contro un avversario molto forte: molto probabilmente si abbandonerebbe prima di acquisire le abilità necessarie. Quindi, invece di spiegare come vincere una gara, in questo capitolo daremo un'occhiata a quali automobili scintillanti guideremo.

In altre parole, offrirò un breve tour dei tre strumenti che utilizzeremo in questo libro.

In questo capitolo semplificherò eccessivamente alcune cose solo per iniziare a lavorare senza entrare ancora nella filosofia del TDD (si pensi alle lezioni di fisica alle elementari). Nessuna paura :-), mi rifarò nei prossimi capitoli!

8.1 Framework di test

Il primo tool che utilizzeremo è un framework. Un framework di test ci consente di specificare ed eseguire i test.

Supponiamo per questa introduzione di avere un'applicazione che accetta due numeri dalla riga di comando, li moltiplica e stampa il risultato sulla console. Il codice è piuttosto semplice:

```
public static void Main(string[] args)
{
  try
  {
    int firstNumber = Int32.Parse(args[0]);
    int secondNumber = Int32.Parse(args[1]);

    var result =
      new Multiplication(firstNumber, secondNumber).Perform();

    Console.WriteLine("Result is: " + result);
```

(continues on next page)

(continua dalla pagina precedente)

```
  }
  catch(Exception e)
  {
    Console.WriteLine("Multiplication failed because of: " + e);
  }
}
```

Supponiamo ora di voler verificare se questa applicazione produce risultati corretti. Il modo più ovvio sarebbe quello di invocarlo manualmente dalla riga di comando con alcuni argomenti esemplificativi, poi controllare l'output sulla console e confrontarlo con quello che ci aspettavamo di vedere. Tale sessione di test potrebbe assomigliare a questa:

```
C:\MultiplicationApp\MultiplicationApp.exe 3 7
21
C:\MultiplicationApp\
```

Come si vede, l'applicazione produce un risultato pari a 21 per la moltiplicazione di 3 per 7. Questo è corretto, quindi presumiamo che l'applicazione abbia superato il test.

E se l'applicazione eseguisse anche addizioni, sottrazioni, divisioni, calcoli, ecc.? Quante volte dovremmo richiamare manualmente l'applicazione per verificare che ogni operazione funzioni correttamente? Non sarebbe una perdita di tempo? Ma ..un momento.., siamo programmatori, giusto? Quindi possiamo scrivere programmi che eseguano i test per noi! Ad esempio, ecco il codice sorgente di un programma che utilizza la classe Multiplication, ma in un modo leggermente diverso rispetto all'applicazione originale:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  if(result != 21)
  {
    throw new Exception("Failed! Expected: 21 but was: " + result);
  }
}
```

Sembra semplice, vero? Ora, utilizziamo questo codice come base per costruire un framework di test molto primitivo, solo per mostrare le parti di cui sono costituiti tali framework. Come avanzamento in questa direzione, possiamo estrarre la verifica di `result` in un metodo riutilizzabile: dopo tutto, aggiungeremo la divisione in un secondo, giusto? Quindi ecco qui:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

//extracted code:
public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
{
  if(actual != expected)
  {
    throw new Exception(
```

(continues on next page)

(continua dalla pagina precedente)

```
      "Failed! Expected: "
        + expected + " but was: " + actual);
  }
}
```

Notare che ho iniziato il nome di questo metodo estratto con "Assert": torneremo presto sulla nomenclatura, per ora presupponiamo solo che questo sia un buon nome per un metodo che verifica che un risultato corrisponda alle nostre aspettative. Facciamo un ultimo giro ed estraiamo il test stesso in modo che il suo codice sia in un metodo separato. A questo metodo si può assegnare un nome che descriva lo scopo del test:

```
public static void Main(string[] args)
{
  Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers();
}

public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Assuming...
  var multiplication = new Multiplication(3,7);

  //when this happens:
  var result = multiplication.Perform();

  //then the result should be...
  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
{
  if(actual != expected)
  {
    throw new Exception(
    "Failed! Expected: " + expected + " but was: " + actual);
  }
}
```

E abbiamo finito. Ora, se abbiamo bisogno di un altro test, ad es. per la divisione, possiamo semplicemente aggiungere unaănuova chiamata al metodo `Main()` e implementarla. All'interno di questo nuovo test, possiamo riutilizzare il metodo `AssertTwoIntegersAreEqual()`, poiché il controllo della divisione riguarderebbe anche il confronto di due valori interi.

Come si vede, possiamo facilmente scrivere controlli automatizzati come questo, utilizzando i nostri metodi primitivi. Tuttavia, questo approccio presenta alcuni svantaggi:

1. Ogni volta che aggiungiamo un nuovo test, dobbiamo aggiornare il metodo `Main()` con unaăchiamata al nuovo test. Se dimentichiamo di aggiungere tale chiamata, il test non verrà mai eseguito. All'inizio non è un grosso problema, ma non appena avremo dozzine di test, un'omissione diventerà difficile da notare.
2. Si immagini che il sistema sia costituito da più di un'applicazione: si avrebbero dei problemi nel tentare di raccogliere risultati riassuntivi per tutte le applicazioni di cui è composto il sistema.
3. Presto si dovranno scrivere molti altri metodi simili ad `AssertTwoIntegersAreEqual()` -- quello che già abbiamo confronta due numeri interi per verificarne l'uguaglianza, ma cosa succederebbe se volessimo verificare una condizione diversa, ad es. che un numero intero sia maggiore di un altro? E se volessimo verificare l'uguaglianza non per gli interi, ma per i caratteri, le stringhe, i numeri in virgola mobile, ecc.? E se volessimo verificare alcune condizioni sulle collezioni, ad es. che una collezione sia ordinata o che tutti gli elementi della collezione siano unici?

4. Se un test fallisce, sarebbe difficile passare dall'output della riga di comando alla riga corrispondente del sorgente nell'IDE. Non sarebbe più semplice se si potesse cliccare sul messaggio di errore per poi andare immediatamente al codice in cui si è verificato l'errore?

Per questi e altri motivi sono stati creati framework di test automatizzati avanzati come CppUnit (per C++), JUnit (per Java) o NUnit (C#). Tali framework sono in linea di principio basate proprio sullidea che ho abbozzato sopra, inoltre compensano le carenze del nostro approccio primitivo. Derivano la loro struttura e funzionalità da SUnit di Smalltalk e sono collettivamente indicati come framework di test della **famiglia xUnit**.

Ad essere onesti, non vedo l'ora di mostrare come appare il test che abbiamo appena scritto quando viene utilizzato un framework di test. Ma prima, ricapitoliamo ciò che abbiamo nel nostro approccio diretto alla scrittura di test automatizzati e introduciamo una terminologia che ci aiuterà a capire come i framework di automazione dei test risolvono i nostri problemi:

1. Il metodo `Main()` funge da **Test List** -- un posto in cui si decide quali test eseguire.
2. Il metodo `Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()` è un **Test Method**.
3. Il metodo `AssertTwoIntegersAreEqual()` è una **Assertion** -- una condizione che, se non soddisfatta, termina un test con un fallimento.

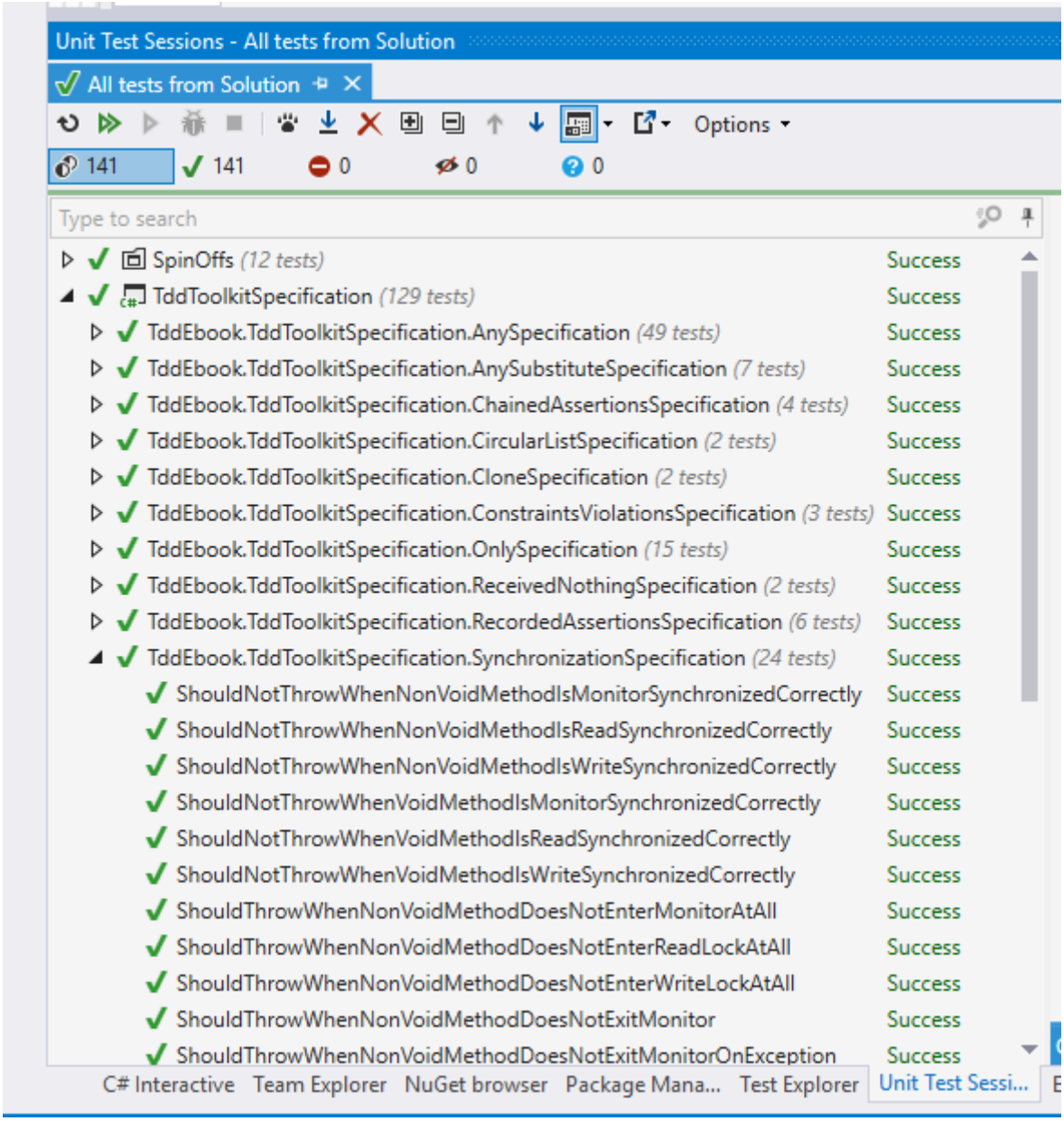
Con nostra gioia, questi tre elementi sono presenti anche quando utilizziamo un framework di test. Inoltre, sono molto più avanzati di quelli che abbiamo. Per illustrare ciò, ecco (finalmente!) lo stesso test che abbiamo scritto sopra, utilizzando ora il framework di test xUnit.Net:

```
[Fact] public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Assuming...
  var multiplication = new Multiplication(3,7);

  //when this happens:
  var result = multiplication.Perform();

  //then the result should be...
  Assert.Equal(21, result);
}
```

Guardando l'esempio, possiamo vedere che il metodo di test stesso è l'unica cosa rimasta: i due metodi (l'elenco di test e l'asserzione) che avevamo in precedenza ora non ci sono più. Beh, a dire il vero, non sono letteralmente spariti: è solo che il framework di test offre sostituti di gran lunga migliori, quindi li abbiamo utilizzati al loro posto. Ribadiamo i tre elementi della versione precedente del test che avevo promesso sarebbero stati presenti dopo la transizione al framework di test:

1. La **Test List** viene ora creato automaticamente dal framework da tutti i metodi contrassegnati con un attributo `[Fact]`. Non è più necessario mantenere uno o più elenchi centrali, quindi il metodo `Main()` non esiste più.
2. Il **Test Method** è presente e sembra quasi lo stesso di prima.
3. La **Assertion** assume la forma di unaăchiamata al metodo statico `Assert.Equal()` -- il framework xUnit.NET è fornito in bundle con un'ampia gamma di metodi di asserzione, quindi ne ho usato uno. Naturalmente, nessuno impedisce di scrivere la propria asserzione se i metodi di asserzione nativi non offrono ciò che serve.

Uff, spero di aver reso la transizione abbastanza indolore. Ora l'ultima cosa da aggiungere: poiché nell'ultimo esempio non esiste più il metodo `Main()`, ci si chiederà, sicuramente, come eseguiamo questi test, giusto? Ok, l'ultimo grande segreto svelato: utilizziamo un'applicazione esterna per questo (ci riferiremo ad essa utilizzando il termine **Test Runner**) -- gli diciamo quali assembly eseguire e poi li carica, li esegue, riporta i risultati, ecc. Un Test Runner può assumere varie forme, ad es. può essere un'applicazione console, un'applicazione GUI o un plug-in per un IDE. Ecco un esempio di un test runner fornito da un plug-in per l'IDE di Visual Studio chiamato Resharper:



8.2 Framework per il Mocking

{{keyExclamation}} Questa introduzione è scritta per coloro che non sono esperti nell'uso dei mock. Anche se accetto il fatto che il concetto potrebbe essere troppo difficile da comprendere. Se, leggendo questa sezione, ci si sente persi, la si salti. Non ci occuperemo degli oggetti mock fino alla parte 2, dove offro una descrizione più ricca e accurata del concetto.

Quando vogliamo testare una classe che dipende da altre classi, potremmo pensare che sia una buona idea includere anche quelle classi nel test. Questo però non ci permette di testare un singolo oggetto o un piccolo gruppo di oggetti, dove potremmo verificare che solo una piccola parte dell'applicazione funzioni correttamente. Per fortuna, se facciamo in modo che le nostre classi dipendano da interfacce anziché da altre classi, possiamo facilmente implementare tali interfacce con speciali classi "fake" che possono essere realizzate in modo da semplificare i nostri test. Ad esempio, gli oggetti di tali classi possono contenere valori di ritorno preprogrammati per alcuni metodi. Possono anche registrare i metodi che vengono invocati e consentire al test di verificare se la comunicazione tra il nostro oggetto in test e le sue dipendenze sia corretta.

Al giorno d'oggi, possiamo fare affidamento su tool per generare un'implementazione "fake" di una determinata interfaccia e consentirci di utilizzare tale implementazione al posto di un oggetto reale nei test. Ciò avviene in modo diverso, a seconda della linguaggio. A volte, le implementazioni dell'interfaccia possono essere generate a runtime (come in Java o in C#), a volte dobbiamo fare più affidamento sulla generazione in fase di compilazione (ad esempio in C++).

Restringendo il campo al C#, un framework di mocking è proprio questo, un meccanismo che ci consente di creare oggetti (chiamati "oggetti mock" o semplicemente "mock"), che aderiscono a una determinata interfaccia, in fase di esecuzione. Funziona così: il tipo di interfaccia che vogliamo implementare viene solitamente passato a un metodo speciale che restituisce un oggetto mock basato su quell'interfaccia (vedremo un esempio tra qualche secondo). A parte la creazione di oggetti mock, tale framework fornisce un'API per configurare i mock su come comportarsi quando vengono chiamati determinati metodi e ci consente di controllare quali chiamate hanno ricevuto. Questa è una funzionalità molto potente, perché possiamo simulare o verificare condizioni che sarebbero difficili da ottenere o osservare utilizzando solo il codice di produzione. I framework di mocking non sono vecchi quanto quelli di test, quindi non sono stati utilizzati in TDD sin dall'inizio.

Darò un rapido esempio di un framework di mocking in azione ora e rimanderò ulteriori spiegazioni del loro scopo ai capitoli successivi, poiché la descrizione completa dei mock e il loro posto in TDD non è così facile da trasmettere.

Supponiamo di avere una classe che consenta di effettuare ordini e poi di inserire questi ordini in un database (utilizzando un'implementazione di un'interfaccia chiamata `OrderDatabase`). Inoltre, gestisce qualsiasi eccezione che possa verificarsi, scrivendola in un log. La classe in sé non fa nulla di importante, ma proviamo a immaginare che si tratti di una logica di un dominio serio. Ecco il codice per questa classe:

```
public class OrderProcessing
{
  OrderDatabase _orderDatabase; //OrderDatabase is an interface
  Log _log;

  //we get the database object  from outside the class:
  public OrderProcessing(
    OrderDatabase database,
    Log log)
  {
    _orderDatabase = database;
    _log = log;
  }

  //other code...

  public void Place(Order order)
  {
    try
    {
      _orderDatabase.Insert(order);
    }
    catch(Exception e)
    {
      _log.Write("Could not insert an order. Reason: " + e);
    }
  }

  //other code...
}
```

Ora, immaginiamo di doverlo testare: come lo facciamo? Vedo già scuotere la testa e dire: "Creiamo semplicemente una connessione al database, invochiamo il metodo `Place()` e vediamo se il record viene aggiunto correttamente nel database". Se lo facessimo, il primo test sarebbe simile a questo:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
{
  //GIVEN
  var orderDatabase = new MySqlOrderDatabase(); //uses real database
  orderDatabase.Connect();
```

(continues on next page)

(continua dalla pagina precedente)

```
  orderDatabase.Clean(); //clean up after potential previous tests
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  var allOrders = orderDatabase.SelectAllOrders();
  Assert.Contains(order, allOrders);
}
```

All'inizio del test, apriamo una connessione al database e puliamo tutti gli ordini esistenti al suo interno (ne parleremo a breve), quindi creiamo un oggetto ordine, lo inseriamo nel database ed interroghiamo il database per tutti gli ordini in esso contenuti. Alla fine affermiamo che l'ordine che abbiamo provato a inserire è tra tutti gli ordini presenti nel database.

Perché puliamo il database all'inizio del test? C'è da ricordarsi che un database fornisce spazio di archiviazione permanente. Se non lo ripuliamo prima di eseguire la logica di questo test, il database potrebbe già contenere l'elemento che stiamo tentando di aggiungere, ad es. dalle precedenti esecuzioni di questo test. Il database potrebbe non consentirci di aggiungere nuovamente lo stesso elemento e il test fallirebbe. Ahia! Fa così male, perché volevamo che i nostri test dimostrassero che qualcosa funziona, ma sembra che possa fallire anche quando la logica è codificata correttamente. A cosa servirebbe un test del genere se non potesse dirci in modo affidabile se la logica implementata è corretta o meno? Pertanto, per garantire che lo stato dell'archiviazione persistente sia lo stesso ogni volta che eseguiamo questo test, puliamo il database prima di ogni esecuzione.

Ora che il test è pronto, abbiamo ottenuto ciò che volevamo? Sarei titubante nel rispondere "sì". Ci sono diverse ragioni per questo:

1. Molto probabilmente il test sarà lento perché l'accesso al database è relativamente lento. Non è raro avere più di mille test in una suite e non voglio aspettare mezz'ora per i risultati ogni volta che li eseguo. Cosa fare?
2. Tutti coloro che desiderano eseguire questo test dovranno impostare un ambiente speciale, ad es. unădatabase locale sul proprio computer. E se la loro configurazione fosse leggermente diversa dalla nostra? Cosa succede se lo schema diventa obsoleto: tutti riusciranno a notarlo e ad aggiornare di conseguenza lo schema dei propri database locali? Dovremmo eseguire nuovamente il nostro script di creazione del database solo per assicurarci di avere lo schema più recente disponibile su cui eseguire i test?
3. Potrebbe non esserci alcuna implementazione del motore di database per il sistema operativo in esecuzione sulla nostra macchina di sviluppo se il nostro target è una piattaforma esotica o mobile.
4. Notare che il test che abbiamo scritto è solo uno di due. Dobbiamo ancora scriverne un altro per lo scenario in cui l'inserimento di un ordine termina con un'eccezione. Come configuriamo il database in uno stato in cui genera un'eccezione? È possibile, ma richiede uno sforzo significativo (ad esempio eliminare una tabella e ricrearla dopo il test, per utilizzarla da altri test che potrebbero aver bisogno che venga eseguita correttamente), il che potrebbe portare alcuni alla conclusione che non vale affatto la pena scrivere tali test.

Ora proviamo ad affrontare questo problema in modo diverso. Supponiamo che `MySqlOrderDatabase` che esegue una query su un database reale sia già testato (questo perché non voglio ancora entrare in una discussione sul test delle query di database: ci arriveremo nei prossimi capitoli) e che l'unica cosa che dobbiamo testare è la classe `OrderProcessing` (ricordate, qui stiamo cercando di immaginare che ci sia davvero una logica di dominio seria codificata). In questa situazione possiamo lasciare `MySqlOrderDatabase` fuori dal test e creare invece un'altra falsa implementazione di `OrderDatabase` che si comporta come se fosse una connessione a un database ma non scrive su un database reale: memorizza solo i record inseriti in un elenco in memoria. Il codice per una connessione falsa potrebbe assomigliare a questo:

```
public class FakeOrderDatabase : OrderDatabase
{
  public Order _receivedArgument;

  public void Insert(Order order)
  {
    _receivedArgument = order;
  }

  public List<Order> SelectAllOrders()
  {
    return new List<Order>() { _receivedOrder };
  }
}
```

Notare che il database fake degli ordini è un'istanza di una classe personalizzata che implementa la stessa interfaccia di `MySqlOrderDatabase`. Pertanto, se ci proviamo, possiamo fare in modo che il codice testato utilizzi il nostro fake senza saperlo.

Sostituiamo la reale implementazione del database degli ordini con l'istanza fake nel test:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
{
  //GIVEN
  var orderDatabase = new FakeOrderDatabase();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  var allOrders = orderDatabase.SelectAllOrders();
  Assert.Contains(order, allOrders);
}
```

Notare che non puliamo l'oggetto del database fake come abbiamo fatto con il database reale poiché creiamo un nuovo oggetto ogni volta che viene eseguito il test e i risultati vengono archiviati in una posizione di memoria diversa per ogni istanza. Inoltre il test sarà molto più veloce ora perché non accederemo più al database. Inoltre, ora possiamo scrivere facilmente un test per il caso di errore. Come? Basta creare un'altra classe fake, implementata in questo modo:

```
public class ExplodingOrderDatabase : OrderDatabase
{
  public void Insert(Order order)
  {
    throw new Exception();
  }

  public List<Order> SelectAllOrders()
  {
  }
}
```

Ok, fin qui tutto bene, ma ora abbiamo due classi di oggetti fake da mantenere (e probabilmente ne avremo bisogno ancora di più). Qualsiasi metodo aggiunto all'interfaccia `OrderDatabase` deve essere aggiunto anche a ciascuna di queste classi fake. Possiamo risparmiare parte della codifica rendendo i nostri mock un po' più generici in modo che il loro comportamento possa essere configurato utilizzando le espressioni lambda:

```
public class ConfigurableOrderDatabase : OrderDatabase
{
  public Action<Order> doWhenInsertCalled;
  public Func<List<Order>> doWhenSelectAllOrdersCalled;

  public void Insert(Order order)
  {
    doWhenInsertCalled(order);
  }

  public List<Order> SelectAllOrders()
  {
    return doWhenSelectAllOrdersCalled();
  }
}
```

Ora non dobbiamo creare classi aggiuntive per nuovi scenari, ma la nostra sintassi diventa scomoda. Ecco come configuriamo il database fake degli ordini per scrivere e restituire l'ordine inserito:

```
var db = new ConfigurableOrderDatabase();
Order gotOrder = null;
db.doWhenInsertCalled = o => {gotOrder = o;};
db.doWhenSelectAllOrdersCalled = () => new List<Order>() { gotOrder };
```

E se vogliamo che sollevi un'eccezione quando viene inserito qualcosa:

```
var db = new ConfigurableOrderDatabase();
db.doWhenInsertCalled = o => {throw new Exception();};
```

Per fortuna, alcuni programmatori intelligenti hanno creato librerie che forniscono ulteriore automazione in tali scenari. Una di queste librerie è **NSubstitute**. Fornisce un'API sotto forma di metodi di estensione di C#, motivo per cui all'inizio potrebbe sembrare un po' magico, soprattutto se non si ha familiarità col C#. Niente paura, ci si abituerà.

Utilizzando NSubstitute, il nostro primo test può essere riscritto come:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderisPlaced()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

Notare che non abbiamo più bisogno del metodo `SelectAllOrders()` sull'interfaccia di connessione al database. Era lì solo per rendere più semplice la scrittura del test: nessun codice di produzione lo utilizzava. Possiamo eliminare il metodo ed eliminare altri problemi di manutenzione. Invece della chiamata a `SelectAllOrders()`, i mock creati da NSubstitute registrano tutte le chiamate ricevute e ci permettono di usare su di esse un metodo speciale chiamato `Received()` (l'ultima riga di questo test), che in realtà è un'asserzione camuffata che controlla se il metodo `Insert()` è stato chiamato con l'oggetto order come parametro.

Questa spiegazione degli oggetti mock è molto superficiale e il suo scopo è solo quello di iniziare a lavorare. Torneremo ai mock più tardi poiché qui abbiamo solo scalfito la superficie.

8.3 Generatore di valori anonimi

Osservando i dati del test nella sezione precedente vediamo che molti valori sono specificati letteralmente, ad es. nel seguente codice:

```
var order = new Order(
  name: "Grzesiek",
  surname: "Galezowski",
  product: "Agile Acceptance Testing",
  date: DateTime.Now,
  quantity: 1);
```

name, surname, product, date e quantity sono molto specifici. Ciò potrebbe suggerire che i valori esatti siano importanti dal punto di vista del comportamento che stiamo testando. D'altra parte, quando guardiamo di nuovo il codice testato:

```
public void Place(Order order)
{
  try
  {
    this.orderDatabase.Insert(order);
  }
  catch(Exception e)
  {
    this.log.Write("Could not insert an order. Reason: " + e);
  }
}
```

possiamo notare che questi valori non vengono utilizzati da nessuna parte: la classe testata non li utilizza né li controlla in alcun modo. Questi valori sono importanti dal punto di vista del database, ma abbiamo già escluso il database vero e proprio. Non disturba il fatto che riempiamo l'oggetto ordine con così tanti valori che sono irrilevanti per la logica del test stesso e che ingombrano la struttura del test con dettagli inutili? Per rimuovere questa confusione, introduciamo un metodo con un nome descrittivo per creare l'ordine e nascondere i dettagli che non ci servono al lettore del test:

```
[Fact] public void
ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = AnonymousOrder();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

(continues on next page)

(continua dalla pagina precedente)

```
public Order AnonymousOrder()
{
  return new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);
}
```

Adesso va meglio. Non solo abbiamo abbreviato il test, ma abbiamo anche fornito al lettore un suggerimento sul fatto che i valori effettivi utilizzati per creare un ordine non contano dal punto di vista della logica testata dell'elaborazione degli ordini. Da qui il nome `AnonymousOrder()`.

A proposito, non sarebbe bello se non dovessimo fornire noi stessi gli oggetti anonimi, ma potessimo fare affidamento su un'altra libreria per generarli per noi? Sorpresa, sorpresa, ce n'è una! Si chiama **Autofixture**. È un esempio del cosiddetto generatore di valori anonimi (anche se al suo creatore piace dire che è anche un'implementazione del pattern Test Data Builder, ma evitiamo questa discussione qui).

Dopo aver modificato il nostro test per utilizzare AutoFixture, arriviamo a quanto segue:

```
private Fixture any = new Fixture();

[Fact] public void
ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = any.Create<Order>();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

In questo test, utilizziamo un'istanza di una classe `Fixture` (che fa parte di AutoFixture) per crearci dei valori anonimi tramite un metodo chiamato `Create()`. Questo ci consente di rimuovere il metodo `AnonymousOrder()`, rendendo così più breve la configurazione del nostro test.

Bello, eh? AutoFixture ha molte funzionalità avanzate, ma per semplificare le cose mi piace nasconderne l'utilizzo dietro una classe statica chiamata `Any`. L'implementazione più semplice di tale classe sarebbe simile a questa:

```
public static class Any
{
  private static any = new Fixture();

  public static T Instance<T>()
  {
    return any.Create<T>();
  }
}
```

Nei prossimi capitoli vedremo molti metodi diversi dal tipo `Any`, oltre alla spiegazione completa della filosofia che sta dietro ad esso. Più usi questa classe, più cresce con altri metodi per creare oggetti personalizzati.

8.4 Riepilogo

Questo capitolo ha introdotto i tre tool che utilizzeremo in questo libro che, una volta assimilati, renderanno più fluido il flusso di sviluppo basato sui test. Se questo capitolo non giustifica abbastanza il loro utilizzo, non c'è da preoccuparsi: approfondiremo la filosofia che sta dietro ad essi nei prossimi capitoli. Per ora, voglio solo che si acquisisca familiarità con gli strumenti stessi e la loro sintassi. Proseguiamo col download di questi tool, avviandoli, provando a scriverci qualcosa di semplice. Non è necessario che si capisca ancora il loro scopo completo, basta fare pratica :-).

CAPITOLO 9

Non è (solo) unătest

Il ruolo di un test è solo quello di "verificare" o "controllare" se un pezzo di software funziona? Sicuramente questa è una parte significativa del suo valore a runtime, ovvero il valore che otteniamo quando eseguiamo il test. Tuttavia, quando limitiamo la nostra visione sui test solo a questo, potremmo portarci alla conclusione che lunica cosa preziosa nellavere un test è poterlo eseguire e visualizzare il risultato. Atti come la progettazione di un test o la sua implementazione avrebbero solo il valore di produrre qualcosa che possiamo eseguire. La lettura di un test avrebbe valore solo durante il debug. È proprio così?

In questo capitolo sostengo che gli atti di progettazione, implementazione, compilazione e lettura di un test sono tutte attività molto preziose. E ci permettono di considerare i test come qualcosa di più di semplici "controlli automatici".

9.1 Quando un test diventa qualcosa di più

Ho studiato a ód, una grande città nel centro della Polonia. Come probabilmente tutti gli altri studenti in tutti gli altri paesi, abbiamo avuto lezioni, esercitazioni ed esami. Gli esami erano piuttosto difficili. Dato che il mio gruppo di informatica frequentava la facoltà di ingegneria elettronica ed elettrica, dovevamo seguire molte lezioni che non avevano nulla a che fare con la programmazione. Ad esempio: elettrotecnica, fisica dello stato solido o metrologia elettronica ed elettrica.

Sapendo che gli esami erano difficili e che era difficile imparare tutto durante il semestre, i docenti a volte ci davano esami esempi dagli anni precedenti. Le domande erano diverse dagli esami effettivi che dovevamo sostenere, ma la struttura e il tipo di domande poste (pratica vs teoria, ecc.) erano simili. Di solito ricevevamo queste domande di esempio prima di iniziare a imparare davvero (che di solito avveniva alla fine di un semestre). Indovina cosa è successo allora? Come potresti sospettare, non abbiamo utilizzato i test ricevuti solo per "verificare" o "controllare" le nostre conoscenze dopo aver finito di apprendere. Al contrario, esaminare questi test è stato il primo passo della nostra preparazione. Perché è stato così? A cosa servivano i test quando sapevamo che non avremmo saputo la maggior parte delle risposte?

Immagino che i miei docenti non sarebbero d'accordo con me, ma trovo piuttosto divertente che ciò che stavamo realmente facendo allora fosse simile allo "sviluppo di software lean" [*snello*]. Lean è una filosofia in cui, tra le altre cose, c'è una rigorosa enfasi sull'eliminazione degli sprechi. Ogni caratteristica o prodotto che viene fatto ma non è necessario a nessuno è considerato uno spreco. Questo perché se qualcosa non è necessario, non c'è motivo di presumere che lo sarà mai. In tal caso, lintera funzionalità o prodotto non aggiunge alcun valore. Anche se mai *sarà* necessario, molto probabilmente richiederà una rielaborazione per soddisfare le esigenze del cliente in quel momento. In tal caso, il lavoro svolto sulle parti della soluzione originale che dovevano essere rielaborate è uno spreco: ha avuto un costo, ma non ha portato alcun beneficio (non sto parlando di cose come demo dei clienti, ma di cose finite, funzionalità o prodotti nuovi).

Quindi, per eliminare gli sprechi, di solito proviamo a "estrarre funzionalità dalla domanda" invece di "inserirle" in un prodotto, sperando che un giorno possano diventare utili. In altre parole, ogni funzionalità è lì per soddisfare un bisogno concreto. In caso contrario, lo sforzo è considerato sprecato e il denaro "annegato".

Tornando all'esempio degli esami, perché l'approccio di visione preliminare degli esempi di test può essere considerato lean [*snello*]? Questo perché, quando consideriamo il superamento di un esame il nostro obiettivo, allora tutto ciò che non ci avvicina a questo obiettivo è considerato uno spreco. Supponiamo che l'esame riguardi solo la teoria: perché allora far pratica con gli esercizi? Probabilmente varrebbe molto di più studiare il lato teorico degli argomenti. Tale conoscenza si potrebbe acquisire da questi test di esempio. Quindi, i test erano una sorta di specifica di ciò che era necessario per superare l'esame. Ci ha permesso di trarre valore (ovvero la nostra conoscenza) dalla domanda (informazioni ottenute da test realistici) piuttosto che spingerlo dall'implementazione (ovvero imparare tutto in un libro di testo, capitolo dopo capitolo).

Quindi i test sono diventati qualcosa di più. Si sono rivelati molto preziosi prima della "implementazione" (cioè dell'apprendimento per l'esame) perché:

1. ci hanno aiutato a concentrarci su ciò che era necessario per raggiungere il nostro obiettivo
2. hanno distolto la nostra attenzione da ciò che **non** era necessario per raggiungere il nostro obiettivo

Questo era il valore di un test prima di imparare. Notare che i test che riceveremo di solito non erano esattamente quelli che avremmo incontrato al momento dell'esame, quindi dovevamo ancora indovinare. Tuttavia, il ruolo di un**ătest come specifica di unăbisogno** era già visibile.

9.2 Torniamo nella terra dello sviluppo del software

Ho scelto questa lunga metafora per mostrare che scrivere un "test" è in realtà un altro modo per specificare un requisito o un bisogno e che non è contro-intuitivo pensare in questo modo: accade nella nostra vita quotidiana. Questo vale anche per lo sviluppo del software. Facciamo il seguente "test" e vediamo che tipo di esigenze vengono specificate:

```
var reporting = new ReportingFeature();
var anyPowerUser = Any.Of(Users.Admin, Users.Auditor);
Assert.True(reporting.CanBePerformedBy(anyPowerUser));
```

(In questo esempio, abbiamo utilizzato il metodo `Any.Of()` che restituisce qualsiasi valore di enumerazione dalla lista specificata. Qui diciamo "dammi un valore che sia `Users.Admin` o `Users.Auditor`").

Diamo un'occhiata a quelle (sole!) tre righe di codice e immaginiamo che il codice di produzione che fa passare questo "test" non esista ancora. Cosa possiamo imparare da queste tre righe su ciò che questo codice di produzione deve fornire? Contate con me:

1. Abbiamo bisogno di una funzionalità di reporting.
2. Dobbiamo supportare la nozione di utenti e privilegi.
3. Dobbiamo supportare il concetto di utente "power", che sia un amministratore o un revisore.
4. Gli utenti "power" devono essere autorizzati a utilizzare la funzionalità di reporting (notare che non specifica quali altri utenti dovrebbero o non dovrebbero essere in grado di utilizzare questa funzionalità: per questo avremmo bisogno di un "test" separato).

Inoltre, siamo già alla fase di progettazione di un'API (perché il test la sta già utilizzando) che soddisferà l'esigenza. Non credete che queste siano già alcune informazioni sulla funzionalità dell'applicazione da sole tre righe di codice?

9.3 Una Specifica invece di una "test suite"

Spero che ora si possa capire che quello che abbiamo chiamato "test" può essere visto anche come una sorta di specifica. Questa è anche la risposta alla domanda sollevata allinizio di questo capitolo.

In realtà il ruolo di un test, se scritto prima del codice di produzione, può essere ulteriormente scomposto:

- progettare uno scenario -- è quando specifichiamo le nostre esigenze fornendo esempi concreti di comportamenti che ci aspettiamo
- scrivere il codice di test -- è quando specifichiamo un'API attraverso la quale vogliamo utilizzare il codice che stiamo testando

- compilazione -- è quando riceviamo feedback sul fatto che il codice di produzione abbia le classi e i metodi richiesti dalle specifiche che abbiamo scritto. In caso contrario, la compilazione fallirà.
- esecuzione -- è dove riceviamo feedback sul fatto che il codice di produzione presenti i comportamenti descritti dalle specifiche
- lettura -- è dove utilizziamo le specifiche già scritte per ottenere informazioni sul codice di produzione.

Pertanto, il nome "test" sembra restringere troppo il campo di ciò che stiamo facendo qui. La mia sensazione è che forse un nome diverso sarebbe migliore, da qui il termine *specifica*.

La scoperta del ruolo dei test come specifica è piuttosto recente e non esiste ancora una terminologia uniforme ad essa collegata. Ad alcuni piace chiamare il processo di utilizzo dei test come *Specifica Per Esempi* per dire che i test sono esempi che aiutano a specificare e chiarire la funzionalità in fase di sviluppo. Alcuni usano il termine BDD (*Behavior-Driven Development*) per sottolineare che scrivere test riguarda in realtà l'analisi e la descrizione dei comportamenti. Inoltre, si potrebbero incontrare nomi diversi per alcuni elementi particolari di questo approccio, ad esempio, ună"test" può essere definito "specifica", un "esempio" o unaă"descrizione del comportamento" o unaă"dichiarazione di specifiche" " o "un fatto relativo al sistema" (come già visto nel capitolo sugli strumenti, il framework xUnit.NET contrassegna ogni "test" con un attributo `[Fact]`, suggerendo che scrivendolo stiamo enunciando un singolo fatto sul codice sviluppato. A proposito, xUnit.NET ci permette anche di formulare theories' [*teorie*] sul nostro codice, ma lasciamo questo argomento per un'altra volta).

Data questa varietà terminologica, vorrei fare unăaccordo: per essere coerente in tutto questo libro, stabilirò una convenzione sulla nomenclatura, ma lascerò la libertà di seguire la propria se lo si desidera. Il motivo di questa convenzione sui nomi è pedagogico: non sto cercando di creare un movimento per cambiare i termini stabiliti o per inventare una nuova metodologia o altro. Spero che, utilizzando questa terminologia in tutto il libro, si vedranno alcune cose diversamente[1]. Quindi, concordiamo che per il bene di questo libro:

Specification Statement (o semplicemente **Statement**, con la 'S' maiuscola)

: verrà utilizzato al posto delle parole "test" e "metodo di test"

Specifica (o semplicemente **Spec**, anch'essa con la 'S' maiuscola)

: verrà utilizzato al posto delle parole "test suite" e "test list"

Statement "False"

: verrà utilizzato al posto di "test fallito"

Statement "True"

: verrà utilizzato al posto di "test superato"

Di tanto in tanto farò riferimento alla terminologia "tradizionale", perché è meglio consolidata e perché si potrebbe aver già sentito altri termini consolidati e chiedersi come dovrebbero essere intesi nel contesto in cui consideriamo i test come una specifica.

9.4 Le differenze tra specifiche eseguibili e "tradizionali".

Si potrebbe avere familiarità con le specifiche dei requisiti o le specifiche di progettazione scritte in inglese semplice o in un'altra lingua parlata. Tuttavia, le nostre Specifiche differiscono da esse in diversi modi. In particolare, il tipo di Specifica che creiamo scrivendo i test:

1. Non è *completamente* scritto in anticipo come sono state scritte molte di queste specifiche "tradizionali" (il che non significa che sia scritto dopo che il completamento del codice - ne parleremo più approfonditamente nei prossimi capitoli).
2. È eseguibile: la si può eseguire per vedere se il codice aderisce o meno alle specifiche. Ciò riduce il rischio di imprecisioni nelle Specifiche e di non sincronizzazione con il codice di produzione.
3. È scritto nel codice sorgente piuttosto che nel linguaggio parlato -- il che è sia positivo, poiché la struttura e la formalità del codice lasciano meno spazio a malintesi, sia impegnativo, poiché è necessario prestare molta attenzione per mantenere tali specifiche leggibili.

[1] inoltre, questo libro è open source, quindi se non piace la terminologia, si è liberi di creare un fork e modificarlo a piacimento!

CAPITOLO 10

La programmazione Statement-first

10.1 Che senso ha scrivere una specifica a posteriori?

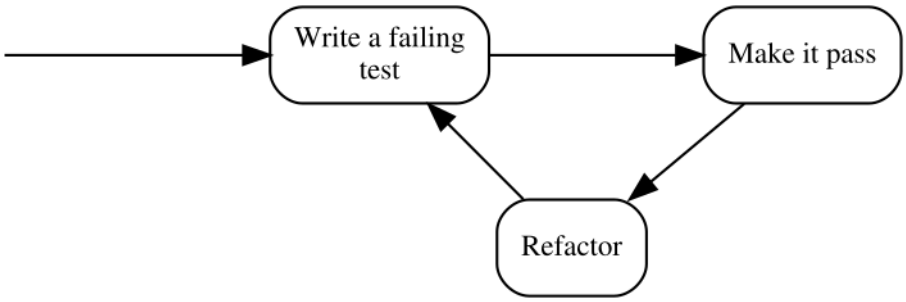
Uno degli aspetti più noti del TDD è che prima che questo comportamento venga implementato viene scritto un test (che fallisce) per un comportamento di un pezzo di codice. Questo concetto è spesso chiamato "sviluppo test-first" e sembra controverso a molti.

Nel capitolo precedente, ho detto che in TDD un "test" assume un ruolo aggiuntivo -- quello di un'affermazione che fa parte di una specifica. Se la mettiamo in questo modo, l'intero controverso concetto di "scrivere un test prima del codice" non costituisce affatto un problema. Al contrario, sembra naturale specificare cosa ci aspettiamo da un pezzo di codice prima di tentare di scriverlo. Ha senso anche il contrario? Una specifica scritta dopo aver completato l'implementazione non è altro che un tentativo di documentare la soluzione esistente. Certo, tali tentativi possono fornire un certo valore se eseguiti come una sorta di reverse engineering (ovvero scrivendo le specifiche per qualcosa che è stato implementato molto tempo fa e per il quale scopriamo le regole o le politiche aziendali precedentemente implicite mentre documentiamo la soluzione esistente) -- contiene l'eccitazione della scoperta, ma farlo subito dopo aver preso tutte le decisioni da soli non mi sembra un modo produttivo di passare il tempo, per non parlare del fatto che lo trovo mortalmente noioso (si può controllare se si è come me su questo. Provare a implementare unaăsemplice app di una calcolatrice e poi scrivere le specifiche subito dopo averla implementata e verificato manualmente che funzioni). Ad ogni modo, difficilmente trovo creativo specificare come dovrebbe funzionare qualcosa dopo che ha funzionato. Forse è questo il motivo per cui, nel corso degli anni, ho osservato che le specifiche scritte dopo l'implementazione di una funzionalità erano molto meno complete di quelle scritte prima dell'implementazione.

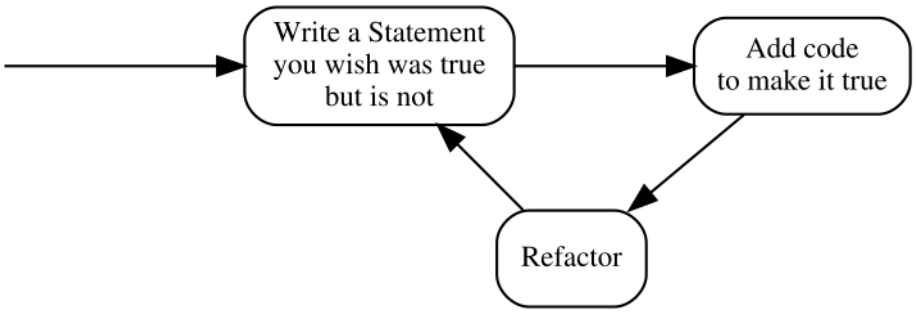
Oh, e ho detto che senza specifiche di alcun tipo non sappiamo se abbiamo finito di implementare le nostre modifiche o meno? Questo perché, per determinare se il cambiamento è completo, dobbiamo paragonare la funzionalità implementata a "qualcosa", anche se questo "qualcosa" è solo nella testa del cliente. In TDD, lo "confrontiamo" con le aspettative stabilite da una serie di test automatizzati.

Un'altra cosa che ho menzionato nel capitolo precedente è che ci avviciniamo alla scrittura di una Specifica di "Istruzioni eseguibili" in modo diverso dalla scrittura di un progetto testuale o di una specifica dei requisiti: anche se un comportamento viene implementato dopo che la sua Specifica è pronta, non scriviamo la Specifica interamente in anticipo. La sequenza abituale consiste nello specificare prima un pezzetto e poi codificarlo, ripetendolo un'istruzione alla volta. Quando eseguiamo il TDD, attraversiamo ripetutamente alcune fasi che compongono un ciclo. Ci piace che questi cicli siano brevi, in modo da ricevere feedback tempestivi e frequenti. Questo è fondamentale perché ci permette di andare avanti, certi che ciò che già abbiamo funzioni come avevamo previsto. Ci consente inoltre di rendere il ciclo successivo più efficiente grazie alla conoscenza acquisita nel ciclo precedente (se non credete che il feedback rapido sia importante, ci si ponga una domanda: "quante volte al giorno si compila il codice su cui si sta lavorando a?").

Leggendo così tanto sui cicli, probabilmente non sorprende che l'illustrazione tradizionale del processo TDD sia modellata visivamente come un flusso circolare:



Notare che il modulo sopra utilizza la terminologia tradizionale di TDD, quindi prima di spiegare i passaggi, ecco un'illustrazione simile che utilizza i nostri termini di Specifiche e [Statement]:



La seconda versione sembra più simile al buon senso della prima -- specificare come dovrebbe comportarsi qualcosa prima di mettere in atto quel comportamento è molto più intuitivo che testare qualcosa che non esiste ancora.

Ad ogni modo, questi tre passaggi meritano qualche spiegazione. Nei prossimi capitoli vi darò alcuni esempi di come funziona nella pratica questo processo e ne presenterò una versione ampliata, ma nel frattempo è sufficiente dire che:

Scrivere uno [Statement] che si vorrebbe fosse vero ma non lo è: significa che lo [Statement] risulta falso. Nell'elenco dei test, appare come non superato, che la maggior parte dei framework xUnit contrassegna con il colore rosso.

Aggiungere codice per renderlo vero: significa che scriviamo codice sufficiente per rendere vera la dichiarazione. Nell'elenco dei test appare come superato, che la maggior parte dei framework xUnit contrassegna con il colore verde. Più avanti nel corso del libro si vedrà quanto poco può essere appena sufficiente.

Refactor : è un passaggio che finora ho ignorato tacitamente e continuerò a farlo per molti altri capitoli. Non c'è da preoccuparsi, prima o poi ci torneremo. Per ora, è importante essere consapevoli che la Specifica eseguibile può fungere da rete di sicurezza mentre miglioriamo la qualità del codice senza modificarne il comportamento visibile esternamente: eseguendo spesso la Specifica, scopriamo rapidamente eventuali errori commessi nel processo.

A proposito, questo processo viene talvolta definito "Red-Green-Refactor", a causa dei colori visualizzati dagli strumenti xUnit per il test fallito e superato. Lo menziono qui solo per la cronaca -- non utilizzerò questo termine ulteriormente nel libro.

10.2 "Test-First" significa vedere un fallimento

Spiegando l'illustrazione con il processo TDD sopra, ho sottolineato che dovremmo scrivere una dichiarazione che vorremmo fosse vera **ma non lo è**. Ciò significa che non solo dobbiamo scrivere uno [Statement] prima di fornire un'implementazione che la renda vera, ma dobbiamo anche valutarla (ovvero eseguirla) e guardarla fallire nelle sue asserzioni prima di fornire l'implementazione.

Perché è così importante? Non è sufficiente scrivere prima lo [Statement]? Perché eseguirlo e vederlo fallire? Ci sono diversi motivi e cercherò di delinearne brevemente alcuni.

Il motivo principale per cui scrivo uno [Statement] e lo vedo fallire è che altrimenti non avrei alcuna prova che lo [Statement] possa mai fallire.

Ogni [Statement] accurato fallisce quando non è soddisfatto e passa quando lo è. Questo è uno dei motivi principali per cui lo scriviamo: vedere questa transizione da *rosso* a *verde*, il che significa che ciò che prima non era stato implementato

(e ne avevamo una prova) ora funziona (e ne abbiamo una prova). Osservare la transizione dimostra che abbiamo fatto progressi.

Un'altra cosa da notare è che, dopo essere stato soddisfatto, lo [Statement] diventa parte della specifica eseguibile e inizia a fallire non appena il codice smette di soddisfarla, ad esempio a seguito di un errore commesso durante il refactoring del codice.

Vedere uno [Statement] provato essere falso fornisce un feedback prezioso. Se eseguiamo uno [Statement] solo *dopo* che il comportamento descritto è stato implementato e valutato come vero, come facciamo a sapere se descrive accuratamente un'esigenza? Non labbiamo mai visto fallire, quindi quali prove abbiamo che fallirà mai?

La prima volta che mi sono imbattuto in questo argomento è stato prima che iniziassi a pensare ai test come a una specifica eseguibile. "Sul serio?" -- ho pensato -- "So quello che sto scrivendo. Se rendo i miei test abbastanza piccoli, è evidente che sto descrivendo il comportamento corretto. Questo è paranoico". Tuttavia, la vita ha rapidamente verificato le mie affermazioni e sono stato costretto a ritirare le mie argomentazioni. Vorrei descrivere tre dei modi in cui ho sperimentato come scrivere uno [Statement] che sia sempre vero, indipendentemente dal fatto che il codice sia corretto o meno. Ci sono più modi, tuttavia, penso che darvene tre dovrebbe essere un esempio sufficiente.

Il "Test-first" mi ha permesso di evitare le seguenti situazioni in cui le dichiarazioni mi hanno ingannato facendomi credere che fossero soddisfatte anche quando non avrebbero dovuto esserlo:

10.2.1 1. Accidental omission of including aăStatement in a Specification

Di solito non è sufficiente scrivere semplicemente il codice di uno [Statement] -- dobbiamo anche far sapere al test runner che un metodo che abbiamo scritto è uno [Statement] (non, ad esempio, solo un metodo helper) e deve essere valutato, cioè eseguito dal runner.

La maggior parte dei framework xUnit dispone di un qualche tipo di meccanismo per contrassegnare i metodi come [Statement], sia utilizzando attributi (C#, ad esempio `[Fact]`) o annotazioni (Java, ad esempio `@Test`), sia utilizzando macro (C e C++), o utilizzando unaăconvenzione sulla nomenclatura. Dobbiamo utilizzare un meccanismo del genere per far sapere al runner che dovrebbe eseguire tali metodi.

Prendiamo xUnit.Net come esempio. Per trasformare un metodo in uno [Statement] in xUnit.Net, dobbiamo contrassegnarlo con l'attributo `[Fact]` in questo modo:

```
public class CalculatorSpecification
{
  [Fact]
  public void ShouldDisplayAdditionResultAsSumOfArguments()
  {
    //...
  }
}
```

È possibile che ci dimentichiamo di decorare un metodo con l'attributo `[Fact]` -- in tal caso, questo metodo non verrà mai eseguito dal test runner. Per quanto possa sembrare divertente, questo è esattamente quello che mi è successo più volte. Prendiamo come esempio lo [Statement] di cui sopra e immaginiamo di scriverlo post-factum come una "unit test" in un ambiente che ha, diciamo, più di trenta [Statement] già scritti e superati. Abbiamo scritto il codice e ora stiamo semplicemente creando test dopo test per garantire che il codice funzioni. Test -- passato, test -- passato, test -- passato. Quando eseguo i test, quasi sempre ne eseguo più di uno alla volta, poiché per me è più semplice che selezionare cosa valutare ogni volta. Inoltre, in questo modo ottengo più sicurezza di non commettere errori e di guastare qualcosa che già funziona. Immaginiamo di fare lo stesso qui. Quindi il flusso di lavoro in realtà è: Test -- tutto superato, test -- tutto superato, test -- tutto superato...

Nel corso del tempo, ho imparato a utilizzare il meccanismo degli snippet di codice del mio IDE per generare un corpo di un template per i miei [Statement]. Tuttavia, nei primi giorni, di tanto in tanto ho scritto qualcosa del genere:

```
public class CalculatorSpecification
{
  //... some Statements here
```

(continues on next page)

(continua dalla pagina precedente)

```
  //oops... forgot to insert the attribute!
  public void ShouldDisplayZeroWhenResetIsPerformed()
  {
    //...
  }
}
```

Come puoi vedere, manca l'attributo `[Fact]`, il che significa che questo [Statement] non verrà eseguito. Ciò è accaduto non solo perché non si utilizzavano generatori di codice -- a volte -- per creare un nuovo [Statement], ma aveva senso copiare e incollare uno [Statement] esistente, modificare il nome e alcune righe di codice[1]. Non sempre mi ricordavo di includere l'attributo `[Fact]` nel codice sorgente copiato. Anche il compilatore non si è lamentato.

Il motivo per cui non ho visto il mio errore è che stavo eseguendo più di un test alla volta -- quando ho ottenuto una barra verde (cioè tutte gli [Statement] si sono rivelati veri), ho pensato che anche lo [Statement] che avevo appena scritto funzionasse. Per me non era interessante cercare ogni nuovo [Statement] nell'elenco e assicurarmi che ci fosse. La ragione più importante, tuttavia, era che l'assenza dell'attributo `[Fact]` non disturbava il mio flusso di lavoro: test -- tutti superati, test -- tutti superati, test -- tutti superati... In altre parole, il mio processo non mi ha dato alcun feedback sul fatto che avessi commesso un errore. Quindi, in tal caso, ciò che ottengo è uno [Statement] che non solo non sarà mai dimostrato falso, ma **non verrà affatto valutato**.

In che modo trattare i test come [Statement] e valutarli prima di renderli veri aiuta in questo caso? La differenza fondamentale è che il flusso di lavoro del TDD è: test -- fallito -- superato, test -- fallito -- superato, test -- fallito -- superato... In altre parole, ci aspettiamo che ogni [Statement] venga dimostrato falso almeno una volta. Pertanto, ogni volta che perdiamo la fase del "fallimento", riceviamo un feedback dal nostro processo che indica che sta accadendo qualcosa di sospetto. Questo ci consente di indagare e risolvere il problema, se necessario.

10.2.2 2. Misplacing Statement setup

Ok, potrebbe sembrare ancora più divertente, ma è successo anche a me un paio di volte, quindi presumo che un giorno potrebbe succedere anche ad altri, soprattutto se si va di fretta.

Consideriamo il seguente esempio banale: vogliamo convalidare una semplice struttura dati che modella un frame di dati che può arrivare tramite rete. La struttura è simile a questa:

```
public class Frame
{
  public int timeSlot;
}
```

e dobbiamo scrivere una Specifica per una classe `Validation` che accetti un oggetto `Frame` come argomento e controlli se l'intervallo di tempo (qualunque esso sia) al suo interno sia corretto. La correttezza viene determinata confrontando il "time slot" [intervallo di tempo] con un valore massimo consentito specificato in una costante chiamata `TimeSlot.MaxAllowed` (quindi è una costante definita in una classe `TimeSlot`). Se l'intervallo di tempo del frame è superiore al massimo consentito, si presuppone che non sia corretto e la convalida dovrebbe restituire `false`. Altrimenti, dovrebbe essere restituito `true`.

Diamo un'occhiata al seguente [Statement] che specifica che l'impostazione di un valore superiore a quello consentito per un campo di un `frame` dovrebbe far fallire la validazione:

```
[Fact]
public void ShouldRecognizeTimeSlotAboveMaximumAllowedAsInvalid()
{
  var frame = new Frame();
  var validation = new Validation();
  var timeSlotAboveMaximumAllowed = TimeSlot.MaxAllowed + 1;
  var result = validation.PerformForTimeSlotIn(frame);
```

(continues on next page)

[1] So che il codice copia-incolla è considerato dannoso e non dovremmo farlo. Quando scrivo [Statement] a livello di unità, faccio alcune eccezioni a questa regola. Ciò verrà spiegato nelle parti successive.

(continua dalla pagina precedente)

```
  frame.timeSlot = timeSlotAboveMaximumAllowed;
  Assert.False(result);
}
```

Notare come il metodo `PerformForTimeSlotIn()`, che attiva il comportamento specificato, viene accidentalmente chiamato *prima* che il valore di `timeSlotAboveMaximumAllowed` venga impostato e quindi questo valore non viene preso in considerazione nel momento in cui viene eseguita la validazione. Se, ad esempio, commettiamo un errore nell'implementazione della classe `Validation` in modo che restituisca `false` per valori inferiori al massimo e non superiori, tale errore potrebbe passare inosservato, poiché lo [Statement] sarà sempre vero.

Ancora una volta, questo è un esempio: l'ho usato solo per illustrare qualcosa che può accadere quando si affrontano casi più complessi.

10.2.3 3. Using static data inside the production code

Di tanto in tanto, dobbiamo intervenire e aggiungere alcuni nuovi [Statement] a una Specification esistente e un po' di logica alla classe che descrive. Supponiamo che la classe e la sua Specifica siano state scritte da qualcun altro oltre a noi. Immaginiamo che il codice di cui stiamo parlando sia un wrapper attorno al file di configurazione XML del nostro prodotto. Decidiamo di scrivere i nostri [Statement] *dopo* aver applicato le modifiche ("beh", potremmo dire, "siamo tutti protetti dalle Specifiche già in vigore, quindi possiamo apportare le nostre modifiche senza il rischio di guastare accidentalmente le funzionalità esistenti, e poi basta testare le nostre modifiche e tutto va bene...").

Iniziamo a programmare... fatto. Ora iniziamo a scrivere questo nuovo [Statement] che descrive la funzionalità che abbiamo appena aggiunto. Dopo aver esaminato la classe Specification , possiamo vedere che ha un campo membro come questo:

```
public class XmlConfigurationSpecification
{
  XmlConfiguration config = new XmlConfiguration(xmlFixtureString);

  //...
```

Ciò che fa è impostare un oggetto utilizzato da ogni [Statement]. Pertanto, ogni [Statement] utilizza un oggettoă`config` inizializzato con lo stesso valore di stringa `xmlConfiguration`. Un altro rapido esame ci porta a scoprire il seguente contenuto della `xmlFixtureString`:

```
<config>
  <section name="General Settings">
    <subsection name="Network Related">
      <parameter name="IP">192.168.3.2</parameter>
      <parameter name="Port">9000</parameter>
      <parameter name="Protocol">AHJ-112</parameter>
    </subsection>
      <subsection name="User Related">
      <parameter name="login">Johnny</parameter>
      <parameter name="Role">Admin</parameter>
      <parameter name="Password Expiry (days)">30</parameter>
    </subsection>
    <!-- and so on and on and on...-->
  </section>
</config>
```

La stringa è già piuttosto grande e disordinata poiché contiene tutte le informazioni richieste dagli [Statement] esistenti. Supponiamo di dover scrivere dei test per unăpiccolo caso particolare che non abbia bisogno di tutta questa schifezza all'interno di questa stringa. Decidiamo quindi di ricominciare da capo e di creare un oggetto separato della classe `XmlConfiguration` con la stringa minima. Lo [Statement] comincia così:

```
string customFixture = CreateMyOwnFixtureForThisTestOnly();
var configuration = new XmlConfiguration(customFixture);
...
```

E continua con lo scenario. Quando lo eseguiamo, passa: bello... no. Ok, cosa c'è che non va? A prima vista va tutto bene, finché non leggiamo attentamente il codice sorgente della classe `XmlConfiguration`. All'interno possiamo vedere come viene memorizzata la stringa XML:

```
private static string xmlText; //note the static keyword!
```

È un campo statico, il che significa che il suo valore viene mantenuto tra le istanze. Che diavolo...? Bene, bene, ecco cosa è successo: l'autore di questa classe ha applicato una piccola ottimizzazione. Ha pensato: "In questa app la configurazione viene modificata solo dai membri del personale di supporto e per farlo devono spegnere il sistema, quindi non è necessario leggere il file XML ogni volta che viene creato un oggetto XmlConfiguration. Posso risparmiare alcuni cicli della CPU e operazioni di I/O leggendolo solo una volta quando viene creato il primo oggetto. Gli oggetti successivi utilizzeranno semplicemente lo stesso XML!". Buono per lui, non altrettanto buono per noi. Perché? Perché, a seconda dell'ordine in cui vengono valutati gli [Statement], per tutti gli [Statement] verrà utilizzata la stringa XML originale o quella personalizzata! Pertanto le dichiarazioni contenute in questa Specifica potrebbero passare o fallire per il motivo sbagliato -- poiché utilizzano accidentalmente l'XML sbagliato.

Iniziare lo sviluppo da uno Statement che prevediamo fallisca può essere d'aiuto quando tale Statement passa anche se il comportamento che descrive non è ancora stato implementato.

10.3 "Test-After" [*testare-dopo*] spesso finisce come "Test-Never" [*testare-mai*]

Si consideri ancora la domanda che ho già posto in questo capitolo: avete mai dovuto scrivere i requisiti o un documento di progettazione per qualcosa che già implementato? È stato divertente? È stato di valore? È stato creativo? Per quanto mi riguarda, la mia risposta a queste domande è *no*. Ho osservato che la stessa risposta si applicava alla formulazione della mia Specifica eseguibile. Osservando me stesso e gli altri sviluppatori, ho concluso che dopo aver scritto il codice, abbiamo poca motivazione per specificare ciò che abbiamo scritto -- alcuni pezzi di codice "che possiamo vedere adesso sono corretti", altri pezzi "che abbiamo già visto funzionare" quando abbiamo compilato e distribuito le nostre modifiche ed eseguito alcuni controlli manuali... Il progetto è pronto... La Specifica? Magari la prossima volta... Pertanto, la Specifica potrebbe non essere mai scritta e, se viene scritta, spesso trovo che copre la maggior parte del flusso principale del programma, ma mancano alcune dichiarazioni che dicono cosa dovrebbe accadere in caso di errori, ecc.

Un altro motivo per cui si finisce per non scrivere la Specifica potrebbe essere la pressione del tempo, soprattutto in team non ancora maturi o che non hanno unetica professionale molto forte. Molte volte ho visto persone reagire alla pressione abbandonando tutto oltre a scrivere il codice che implementa direttamente una funzionalità. Tra le cose che vengono abbandonate ci sono il design, i requisiti e i test. E anche imparare. Ho visto molte volte team che, quando sotto pressione, hanno smesso di sperimentare e apprendere e sono tornati a vecchi comportamenti "sicuri" con una mentalità di "salvare una nave che affonda" e "sperare per il meglio". In tali situazioni, ho visto aumentare la pressione man mano che il progetto si avvicinava alla scadenza o al traguardo, lasciare la Specifica fino alla fine significa che è molto probabile che venga abbandonata, soprattutto nel caso in cui le modifiche vengono (in una certa misura) testate manualmente in seguito comunque.

D'altra parte, quando si esegue TDD (come vedremo nei prossimi capitoli) la nostra Specifica cresce insieme al codice di produzione, quindi c'è molta meno tentazione di abbandonarla del tutto. Inoltre, in TDD, una Specifica scritta non è un'aggiunta al codice, ma piuttosto *un motivo* per scrivere il codice. La creazione di una Specifica eseguibile diventa una parte indispensabile dell'implementazione di una funzionalità.

10.4 Il "Test-After" spesso porta a una rielaborazione del progetto

Mi piace leggere e guardare Uncle Bob (Robert C. Martin) [https://it.wikipedia.org/wiki/Robert_Cecil_Martin]. Un giorno stavo ascoltando il suo keynote al Ruby Midwest 2011, intitolato Architecture The Lost Years. Alla fine, Robert ha fatto alcune digressioni, una delle quali sul TDD. Ha detto che scrivere test dopo il codice non è TDD e lo ha invece definito "una perdita di tempo".

Il mio pensiero iniziale era che il commento fosse forse un po' troppo esagerato e riguardasse solo il fatto di perdere tutti i vantaggi che mi porta iniziare con una dichiarazione falsa: la possibilità di vedere la dichiarazione fallire, la capacità di fare un'analisi pulita, ecc. Tuttavia, ora sento che c'è molto di più, grazie a qualcosa che ho imparato da Amir Kolsky e Scott Bain -- per poter scrivere una specifica gestibile per un pezzo di codice, il codice deve avere un alto livello di **testabilità **. Parleremo di questa qualità nella seconda parte di questo libro, ma per ora assumiamo la seguente definizione semplificata: maggiore è la testabilità di un pezzo di codice (ad esempio una classe), più facile sarà scrivere uno Statement per il suo comportamento.

Ora, dov'è lo spreco nello scrivere le specifiche dopo aver scritto il codice? Per scoprirlo, confrontiamo gli approcci Statement-first e code-first. Nel flusso di lavoro Statement-first per il nuovo codice (non legacy), il mio flusso di lavoro e il mio approccio alla testabilità di solito assomigliano a questo:

1. Scrivere uno Statement che sia falso inizialmente (durante questo passaggio, si rileva e si correggono i problemi di testabilità anche prima che il codice di produzione venga scritto).
2. Scrivere il codice per rendere vero lo Statement.

Ed ecco cosa vedo spesso fare ai programmatori quando scrivono prima il codice (passaggi aggiuntivi contrassegnati in **grassetto**):

1. Scrivere del codice di produzione senza considerare come verrà testato (dopo questo passaggio, la testabilità è spesso non ottimale poiché di solito non viene presa in considerazione a questo punto).
2. **Iniziare a scrivere una unit test** (questo potrebbe non sembrare un passaggio aggiuntivo, poiché è presente anche nell'approccio precedente, ma una volta raggiunto il passaggio 5, si capirà cosa intendo).
3. **Si nota che la unit test del codice che abbiamo scritto è complicato e insostenibile e i test diventano confusi mentre si cercano di aggirare i problemi di testabilità**.
4. **Sia decide di migliorare la testabilità ristrutturando il codice, ad es. per essere in grado di isolare gli oggetti e utilizzare tecniche come gli oggetti mock**.
5. Si scrivono unit test (questa volta dovrebbe essere più semplice poiché la testabilità del testato è migliore).

Qual è l'equivalente dei passaggi contrassegnati nell'approccio Statement-first? Non c'è n'è nessuno! Fare queste cose è una perdita di tempo! Purtroppo, questo è uno spreco che incontro spesso.

10.5 Riepilogo

In questo capitolo ho cercato di mostrare che la scelta di *quando* scriviamo la nostra Specifica spesso fa un'enorme differenza e che ci sono numerosi vantaggi nell'iniziare con uno [Statement]. Quando consideriamo la Specifica per quello che realmente è -- non solo come una serie di test che controllano la correttezza a runtime -- allora l'approccio Statement-first diventa meno scomodo e meno controintuitivo.

CAPITOLO 11

Mettere in pratica ciò che abbiamo già imparato

> E ora, un assaggio di quello che verrà!
>
> -- Shang Tsung, Mortal Kombat Il Film

La citazione precedente è avvenuta poco prima di unaăscena di combattimento in cui unăguerriero senza nome saltava su Sub-Zero solo per essere congelato e spezzato in più pezzi dopo aver colpito il muro. La scena non era spettacolare in termini di tecnica di combattimento o di durata. Inoltre, il ragazzo senza nome non si è nemmeno sforzato -- l'unica cosa che ha fatto è stata saltare solo per essere colpito da una palla congelata, che, tra l'altro, poteva effettivamente vedere arrivare. Sembrava che il combattimento fosse stato organizzato solo per mostrare l'abilità di congelamento di Sub-Zero. Indovinate un po'? In questo capitolo faremo più o meno la stessa cosa -- creeremo uno scenario facile e fasullo solo per mostrare alcuni degli elementi base del TDD!

Il capitolo precedente era pieno di molta teoria e filosofia, vero? Spero che non vi siate addormentati nel leggerlo. A dire il vero, dobbiamo comprendere molta più teoria prima di poter scrivere applicazioni realistiche utilizzando il TDD. Per compensare in qualche modo questo, propongo di fare una deviazione dal sentiero e provare ciò che abbiamo già imparato in un esempio semplice e veloce. Nell'esaminare l'esempio, ci si potrebbe chiedere come diavolo si potrebbero scrivere applicazioni reali nel modo in cui scriveremo il nostro semplice programma. Non c'è da preoccuparsi, non mostrerò ancora tutti i trucchi, quindi lo si consideri come un "assaggio delle cose a venire". In altre parole, l'esempio sarà tanto vicino ai problemi reali quanto il combattimento tra Sub-Zero e il ninja senza nome lo era al vero combattimento di arti marziali, ma mostrerà alcuni degli elementi del processo TDD.

11.1 Lasciate che racconti una storia

Incontriamo Johnny e Benjamin, due sviluppatori della Buthig Company. Johnny è abbastanza fluente nella programmazione e nello sviluppo basato sui test, mentre Benjamin è uno stagista sotto la guida di Johnny ed è ansioso di imparare il TDD. Stanno andando dalla loro cliente, Jane, che ha richiesto la loro presenza perché vuole che scrivano un piccolo programma per lei. Insieme a loro vedremo come interagiscono col cliente e come Benjamin cerca di comprendere le basi del TDD. Come altri, Benjamin è un principiante, quindi le sue domande potrebbero riflettere quelle di altri principianti. Tuttavia, se si trova qualcosa spiegato in modo non sufficientemente dettagliato, non c'è da preoccuparsi -- nei prossimi capitoli approfondiremo questo argomento.

11.2 Atto 1: L'Auto

Johnny: Come ti senti riguardo al tuo primo incarico?

Benjamin: Sono piuttosto emozionato! Spero di poter imparare alcune delle cose sul TDD che hai promesso di insegnarmi.

Johnny: Non solo TDD, ma utilizzeremo anche alcune delle pratiche associate a unăprocesso chiamato "Acceptance Test-Driven Development" [*sviluppo basato sui test di accettazione*], anche se in unaăforma semplificata.

Benjamin: Acceptance Test-Driven Development? Che cos'è?

Johnny: Mentre il TDD viene solitamente definito unaătecnica di sviluppo, l'Acceptance Test-Driven Development (ATDD) è qualcosa di più di unămetodo di collaborazione. Sia ATDD che TDD contengono un po' di analisi e funzionano molto bene insieme poiché entrambi utilizzano gli stessi principi di base, solo a livelli diversi. Avremo bisogno solo di unăpiccolo sottoinsieme di ciò che ATDD ha da offrire, quindi non esagerare.

Benjamin: Certo. Chi è il nostro cliente?

Johnny: Si chiama Jane. Gestisce un piccolo negozio nelle vicinanze e vuole che scriviamo un'applicazione per il suo nuovo cellulare. Avrai l'opportunità di incontrarla tra un minuto, dato che siamo quasi arrivati.

11.3 Atto 2: Presso il Cliente

Johnny: Ciao, Jane, come stai?

Jane: Bene, grazie e tu?

Johnny: Anch'io, grazie. Benjamin, lei è Jane, la nostra cliente. Jane, questo è Benjamin, lavoreremo insieme sul compito che hai per noi.

Benjamin: Ciao, piacere di conoscerti.

Jane: Ciao, piacere di conoscerti anch'io.

Johnny: Allora, puoi dirci qualcosa sul software che dobbiamo scriverti?

Jane: Certo. Di recente ho acquistato un nuovo smartphone in sostituzione di quello vecchio. Il fatto è che sono davvero abituato all'applicazione calcolatrice che girava sul mio telefono precedente e non riesco a trovare una controparte per il mio nuovo dispositivo.

Benjamin: Non potresti semplicemente usare un'altra app calcolatrice? Probabilmente ce ne sono molte disponibili per il download dal web.

Jane: Questo è vero. Le ho controllate tutte e nessuna ha lo stesso comportamento di quella che ho utilizzato per i miei calcoli fiscali. Vedi, questa app è stata come una mano destra per me e aveva alcune belle scorciatoie che mi hanno reso la vita più semplice.

Johnny: Quindi vuoi che riproduciamo l'applicazione per eseguirla sul tuo nuovo dispositivo?

Jane: Esattamente.

Johnny: Sei consapevole che, a parte le fantasiose funzionalità che stavi utilizzando, dovremo impegnarci per implementare le funzionalità di base di cui dispongono tutte le calcolatrici?

Jane: Certo,ămi va bene. Mi sono abituata così tanto all'applicazione della calcolatrice che, se uso qualcos'altro per più di qualche mese, dovrò pagare uno psicoterapeuta al posto vostro. A parte questo, scrivere un'app per la calcolatrice mi sembra un compito facile, quindi il costo non sarà eccessivo, giusto?

Johnny: Penso di aver capito. Allora diamoci da fare. Implementeremo la funzionalità in modo incrementale, iniziando con le funzionalità più essenziali. Quale caratteristica della calcolatrice considereresti più essenziale?

Jane: Si tratterebbe della somma di numeri,ăimmagino.

Johnny: Ok, questo sarà il nostro obiettivo per la prima iterazione. Dopo l'iterazione, forniremo questa parte della funzionalità affinché tu possa provarla e darci un feedback. Tuttavia, prima ancora di poter fornire la funzione aggiuntiva, dovremo implementare la visualizzazione delle cifre sullo schermo mentre le inserisci. È corretto?

Jane: Sì, ho bisogno che anche gli elementi di visualizzazione funzionino: è un prerequisito per altre funzionalità, quindi...

Johnny: Ok, questa è una funzionalità semplice, quindi permettimi di suggerire alcune "user story" quando ho capito quello che hai già detto e mi correggerai dove sbaglio. Eccoci qui:

1. **Per** sapere che la calcolatrice è accesa, **Come** contribuente **voglio** vedere "0" sullo schermo non appena la accendo.

2. *Per** vedere su quali numeri sto operando attualmente, **Come** contribuente, **voglio** che la calcolatrice visualizzi i valori che inserisco
3. **Per** calcolare la somma dei miei diversi redditi, **Come** contribuente **voglio** che la calcolatrice consenta l'addizione di più numeri

Cosa ne pensi?

Jane: Le "story" riflettono più o meno ciò che voglio per la prima iterazione. Non credo di avere correzioni da apportare.

Johnny: Ora prenderemo ogni "story" e raccoglieremo alcuni esempi di come dovrebbe funzionare.

Benjamin: Johnny, non pensi che sia abbastanza ovvio procedere subito con l'implementazione?

Johnny: Credimi, Benjamin, se c'è una parola che temo di più nella comunicazione, è "ovvio". I problemi di comunicazione si verificano più spesso riguardo a cose che le persone considerano ovvie, semplicemente perché altre persone non lo fanno.

Jane: Ok, ci sto. Cosa devo fare?

Johnny: Esaminiamo le "story" una per una e vediamo se riusciamo a trovare alcuni esempi chiave di come dovrebbero funzionare le funzionalità. La prima story è...

11.3.1 Per sapere che la calcolatrice è accesa, Come contribuente voglio vedere "0" sullo schermo non appena la accendo.

Jane: Non credo che ci sia molto di cui parlare. Se visualizzi "0", sarò felice. È tutto.

Johnny: Scriviamo questo esempio utilizzando una tabella:

sequenza di tasti	Output visualizzato	Note
N/A	0	Valore visualizzato iniziale

Benjamin: Questo mi fa pensare... cosa dovrebbe succedere se premo di nuovo "0" in questa fase?

Johnny: Bella questione, ecco a cosa servono questi esempi -- rendono concreto il nostro pensiero. Come dice Ken Pugh[1]: "Spesso la completa comprensione di un concetto non avviene finché qualcuno non tenta di utilizzare il concetto". Normalmente, inseriremmo l'esempio "premere zero più volte" in un elenco di TODO e lo lasceremmo per dopo, perché fa parte di una "story" diversa. Tuttavia, sembra che abbiamo finito con la "story" attuale, quindi andiamo avanti. La "story" successiva riguarda la visualizzazione delle cifre immesse. Che ne dici, Jane?

Jane: Sono d'accordo.

Johnny: Benjamin?

Benjamin: Sì, vai avanti.

11.3.2 *Per** vedere su quali numeri sto operando attualmente, Come contribuente, voglio che la calcolatrice visualizzi i valori che inserisco

Johnny: Cominciamo con il caso sollevato da Benjamin. Cosa dovrebbe succedere se inserisco "0" più volte dopo che sul display è visualizzato solo "0"?

Jane: Dovrebbe essere visualizzato un singolo "0", indipendentemente da quante volte premo "0".

Johnny: Vuoi dire questo?

sequenza di tasti	Output visualizzato	Note
0,0,0	0	Lo zero è un caso speciale – viene visualizzato solo una volta

Jane: Questo è vero. Oltre a questo, le cifre dovrebbero semplicemente essere visualizzate sullo schermo, in questo modo:

[1] K. Pugh, Prefactoring, O'Reilly Media, 2005

sequenza di tasti	Output visualizzato	Note
1,2,3	123	Vengono visualizzate le cifre immesse

Benjamin: Che ne dici di questo:

sequenza di tasti	Output visualizzato	Note
1,2,3,4,5,6,7,1,2,3,4,5,6	1234567123456?	Le cifre immesse vengono visualizzate?

Jane: A dire il vero no. La mia vecchia app calcolatrice ha un limite di sei cifre che posso inserire, quindi dovrebbe essere:

sequenza di tasti	Output visualizzato	Note
1,2,3,4,5,6,7,1,2,3,4,5,6	123456	Display limitato a sei cifre

Johnny: Un'altra bella questione, Benjamin!

Benjamin: Penso di cominciare a capire perché ti piace lavorare con gli esempi!

Johnny: Bene. C'è qualcos'altro, Jane?

Jane: No, praticamente è tutto. Cominciamo a lavorare su un'altra "story".

11.3.3 Per calcolare la somma dei miei diversi redditi, Come contribuente voglio che la calcolatrice consenta l'addizione di più numeri

Johnny: Il seguente scenario è l'unico che dobbiamo supportare?

sequenza di tasti	Output visualizzato	Note
2,+,3,+,4,=	9	Semplice addizione di numeri

Jane: Questo scenario è corretto, tuttavia, c'è anche un caso in cui inizio con "+" senza aver prima inserito alcun numero. Questo dovrebbe essere considerato come un'aggiunta a zero:

sequenza di tasti	Output visualizzato	Note
+,1,=	1	Scorciatoia per l'addizione – trattata come 0+1

Benjamin: Che ne dici di quando l'output è un numero più lungo del limite di sei cifre? È corretto troncarlo in questo modo?

sequenza di tasti	Output visualizzato	Note
9,9,9,9,9,9,+,9,9,9,9,9,9,=	199999	Il nostro display è limitato a sole sei cifre

Jane: Certo, non mi dispiace. Comunque non aggiungo numeri così grandi.

Johnny: C'è ancora una domanda che ci è sfuggita. Diciamo che inserisco un numero, quindi premo "+" e poi un altro numero senza chiedere il risultato con "=". Cosa dovrei vedere?

Jane: Ogni volta che premi "+", la calcolatrice considera terminato l'inserimento del numero corrente e lo sovrascrive non appena premi qualsiasi altra cifra:

sequenza di tasti	Output visualizzato	Note
2,+,3	3	Le cifre immesse dopo l'operatore + vengono trattate come cifre di un nuovo numero, quello precedente viene memorizzato

Jane: Oh, e chiedere il risultato subito dopo aver acceso la calcolatrice dovrebbe risultare "0".

sequenza di tasti	Output visualizzato	Note
=	0	Il tasto risultato di per sé non fa nulla

Johnny: Riassumiamo le nostre scoperte:

sequenza di tasti	Output visualizzato	Note
N/A	0	Valore visualizzato iniziale
1,2,3	123	Vengono visualizzate le cifre immesse
0,0,0	0	Lo zero è un caso speciale – viene visualizzato solo una volta
1,2,3,4,5,6,7	123456	Il nostro display è limitato a sole sei cifre
2,+,3	3	Le cifre immesse dopo l'operatore + vengono trattate come cifre di un nuovo numero, quello precedente viene memorizzato
=	0	Il tasto risultato di per sé non fa nulla
+,1,=	1	Scorciatoia per l'addizione – trattata come 0+1
2,+,3,+,4,=	9	Semplice addizione di numeri
9,9,9,9,9,9,+,9,9,9	199999	Il nostro display è limitato a sole sei cifre

Johnny: La limitazione delle cifre visualizzate sembra una funzionalità completamente nuova, quindi suggerisco di aggiungerla al backlog e di farlo in un altro sprint. In questo sprint non gestiremo affatto una situazione del genere. Che ne dici, Jane?

Jane: Per me va bene. Sembra che ci sia molto lavoro. È bello che l'abbiamo scoperto in anticipo. Per me, la capacità limitante sembrava così ovvia che non pensavo nemmeno che valesse la pena menzionarla.

Johnny: Vedi? Ecco perché non mi piace la parola "ovvio". Jane, ti ricontatteremo se dovessero sorgere altre domande. Per ora, penso che ne sappiamo abbastanza per implementare queste tre "story" per te.

Jane: buona fortuna!

11.4 Atto 3: Test-Driven Development

Benjamin: Wow, è stato fantastico. Si trattava di uno "Acceptance Test-Driven Development"?

Johnny: In una versione molto semplificata, sì. Il motivo per cui ti ho portato con me è stato per mostrarti le somiglianze tra lavorare con il cliente come abbiamo fatto noi e lavorare con il codice utilizzando il processo TDD. Entrambi applicano gli stessi principi, solo a livelli diversi.

Benjamin: Muoio dalla voglia di vederlo con i miei occhi. Cominciamo?

Johnny: Certo. Se seguissimo il processo ATDD, inizieremmo a scrivere quelle che chiamiamo specifiche del livello di accettazione (acceptance-level specification). Nel nostro caso, tuttavia, sarà sufficiente una specifica a livello di unità (unit-level). Prendiamo il primo esempio:

11.4.1 Statement 1: La calcolatrice dovrebbe visualizzare 0 al momento della creazione

sequenza di tasti	Output visualizzato	Note
N/A	0	Valore visualizzato iniziale

Johnny: Benjamin, prova a scrivere il primo Statement.

Benjamin: Oh cavolo, non so come iniziare.

Johnny: Inizia scrivendo la dichiarazione in un inglese semplice. Cosa dovrebbe fare la calcolatrice?

Benjamin: Dovrebbe essere visualizzato "0" quando accendo l'applicazione.

Johnny: Nel nostro caso, "accendere" significa creare unaăcalcolatrice. Scriviamolo come nome del metodo:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{

}

}
```

Benjamin: Perché il nome della classe è `CalculatorSpecification` e il nome del metodo `ShouldDisplay0WhenCreated`?

Johnny: È una convenzione di denominazione. Ce ne sono molti altri, ma questo è quello che mi piace. In questa convenzione, la regola è che quando prendi il nome della classe senza la parte `Specification` seguita dal nome del metodo, dovrebbe formare una frase valida. Ad esempio, se lo applico a ciò che abbiamo scritto, creerebbe una frase: "Calculator should display 0 when created" [La calcolatrice dovrebbe visualizzare 0 quando viene creata].

Benjamin: Ah, ora capisco. Quindi è una dichiarazione di comportamento, giusto?

Johnny: Esatto. Ora, il secondo trucco che posso dirti è che se non sai con quale codice iniziare il tuo Statement, inizia con il risultato atteso. Nel nostro caso, ci aspettiamo che il comportamento finisca per visualizzare "0", giusto? Quindi scriviamolo semplicemente sotto forma di affermazione.

Benjamin: Intendi qualcosa del genere?

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Esattamente.

Benjamin: Ma questo non è nemmeno compilabile. A cosa serve?

Johnny: Il codice che non viene compilato è il feedback di cui avevi bisogno per procedere. Mentre prima non sapevi da dove cominciare, ora hai un obiettivo chiaro -- compilare questo codice. Innanzitutto, da dove prendi il valore visualizzato?

Benjamin: Dal display della calcolatrice, ovviamente!

Johnny: Quindi scrivi come ottieni il valore dal display.

Benjamin: Tipo come?

Johnny: In questo modo:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Benjamin: Capisco. Ora la calcolatrice non viene creata da nessuna parte. Devo crearla da qualche parte adesso altrimenti non verrà compilato: così capisco che è il prossimo passo. È così che funziona?

Johnny: Sì, stai capendo velocemente.

Benjamin: Ok allora, ecco qui:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Bravo!

Benjamin: Il codice non viene ancora compilato, perché non ho definito la classe `Calculator`...

Johnny: Sembra un buon motivo per crearla.

Benjamin: OK.

```
public class Calculator
{
}
```

Benjamin: Sembra che manchi anche il metodo `Display()`. Lo aggiungerò.

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Ehi ehi, non così in fretta!

Benjamin: Cosa?

Johnny: Hai già fornito un'implementazione di `Display()` che renderà vero il nostro Statement attuale. Ricordi il suo nome? `ShouldDisplay0WhenCreated` -- ed è esattamente ciò che fa il codice che hai scritto. Prima di arrivare a questo punto, assicuriamoci che questa affermazione possa mai essere valutata come falsa. Non otterrai questo risultato fornendo un'implementazione corretta e pronta all'uso. Quindi per ora, cambiamolo in questo:

```
public class Calculator
{
  public string Display()
  {
    return "Once upon a time in Africa";
  }
}
```

Johnny: Guarda, ora possiamo eseguire la Specifica e guardare lo Statement valutato come falso, perché si aspetta "0", ma ottiene "Once upon aătime in Africa" [C'era una volta in Africa].

Benjamin: In esecuzione... Ok, è falso. A proposito, usi sempre valori così stupidi per rendere falsi i tuoi Statement?

Johnny: Hahaha, no, l'ho fatto solo per enfatizzare il punto. Normalmente, scriverei `return "";` o qualcosa di simile. Ora possiamo valutare lo Statement e vederlo diventare falso. Siamo quindi sicuri di non aver ancora implementato quanto richiesto affinché lo Statement sia vero.

Benjamin: Penso di aver capito. Per ora, lo Statement mostra che non abbiamo qualcosa di cui abbiamo bisogno e ci dà un motivo per aggiungere questa "cosa". Quando lo faremo, questo Statement dimostrerà che abbiamo ciò di cui abbiamo bisogno. Quindi, cosa facciamo ora?

Johnny: Scrivi la cosa più semplice che rende vero questo Statement.

Benjamin: così?

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Sì.

Benjamin: Ma questa non è unaăimplementazione reale. A cosa serve inserire una stringa hardcoded? L'implementazione finale non sarà sicuramente così!

Johnny: Hai ragione. Molto probabilmente limplementazione finale sarà diversa. Ciò che abbiamo fatto, tuttavia, è ancora prezioso perché:

1. Sei ad un passo dall'implementazione della soluzione finale
2. La sensazione che questa non sia l'implementazione finale ti spinge a scrivere più Statement. Quando ci sono abbastanza Statement per completare la tua implementazione, di solito significa che hai anche una Specifica completa dei comportamenti della classe.
3. Se consideri il rendere vera ogni Statement come un risultato, questa pratica ti consentirà di evolvere il tuo codice senza perdere ciò che hai già ottenuto. Se per sbaglio guasti uno qualsiasi dei comportamenti che hai già implementato, la Specifica te lo dirà perché uno degli Statement esistenti che prima erano veri diventerà falso. Puoi quindi risolverlo o annullare le modifiche utilizzando il controllo della versione e ricominciare dal punto in cui tutti gli Statement esistenti erano veri.

Benjamin: Ok, sembra che dopo tutto ci siano dei vantaggi. Dovrò comunque abituarmi a questo modo di lavorare.

Johnny: Non preoccuparti, questo approccio è una parte importante del TDD, quindi lo capirai in pochissimo tempo. Ora, prima di procedere col prossimo Statement, diamo un'occhiata a ciò che abbiamo già ottenuto. Per prima cosa abbiamo

scritto uno Statement che si è rivelato falso. Poi, abbiamo scritto il codice sufficiente per rendere vera lo Statement. È ora di eseguire un passaggio chiamato Refactoring. In questo passaggio, daremo un'occhiata allo Statement e al codice ed elimineremo le duplicazioni. Riesci a vedere cosa è duplicato tra lo Statement e il codice?

Benjamin: entrambi contengono lo "0" letterale. Lo Statement lo dice qui:

```
Assert.Equal("0", displayedResult);
```

e l'implementazione qui:

```
return "0";
```

Johnny: Bene, eliminiamo questa duplicazione introducendo una costante chiamata `InitialValue`. Lo Statement ora sarà simile a questo:

```
[Fact] public void
ShouldDisplayInitialValueWhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal(Calculator.InitialValue, displayedResult);
}
```

e l'implementazione:

```
public class Calculator
{
  public const string InitialValue = "0";
  public string Display()
  {
    return InitialValue;
  }
}
```

Benjamin: l codice sembra migliore e avere la costante "0" in un unico posto lo renderà più gestibile. Tuttavia, penso che lo Statement nella sua forma attuale sia più debole di prima. Voglio dire, possiamo cambiare `InitialValue` in qualsiasi cosa e la dichiarazione sarà comunque vera poiché non afferma che questa costante deve avere un valore di "0".

Johnny: Esatto. Dobbiamo aggiungerlo alla lista dei nostri TODO per gestire questo caso. Puoi scriverlo?

Benjamin: Certo. Lo scriverò come "TODO: 0 deve essere utilizzato come valore iniziale".

Johnny: Ok. Dovremmo occuparcene ora, soprattutto perché fa parte della "story" che stiamo attualmente implementando, ma lo lascerò per dopo solo per mostrare la potenza della lista TODO nel TDD -- qualunque cosa stia nella lista, possiamo dimenticarla e tornare indietro a quando non abbiamo niente di meglio da fare. Il prossimo elemento della lista è questo:

11.4.2 Statement 2: La calcolatrice dovrebbe visualizzare le cifre immesse

sequenza di tasti	Output visualizzato	Note
1,2,3	123	Vengono visualizzate le cifre immesse

Johnny: Benjamin, puoi presentare uno Statement per questo comportamento?

Benjamin: Ci proverò. Ecco qui:

```
[Fact] public void
ShouldDisplayEnteredDigits()
```

(continues on next page)

(continua dalla pagina precedente)

```
{
  var calculator = new Calculator();

  calculator.Enter(1);
  calculator.Enter(2);
  calculator.Enter(3);
  var displayedValue = calculator.Display();

  Assert.Equal("123", displayedValue);
}
```

Johnny: Vedo che stai imparando velocemente. Hai capito bene le parti relative alla denominazione e alla strutturazione di una Dichiarazione. C'è una cosa su cui dovremo lavorare qui però.

Benjamin: Che succede?

Johnny: Quando abbiamo parlato con Jane, abbiamo utilizzato esempi con valori reali. Questi valori reali sono stati estremamente utili per individuare i casi limite e scoprire gli scenari mancanti. Erano anche più facili da immaginare, quindi erano perfetti per la conversazione. Se automatizzassimo questi esempi a livello di accettazione, utilizzeremmo anche quei valori reali. Quando scriviamo dichiarazioni a livello di unità, però, utilizziamo una tecnica diversa per rendere questo tipo di specifica più astratta. Prima di tutto, lasciami elencare i punti deboli dell'approccio che hai appena utilizzato:

1. Fare in modo che un metodo `Enter()` accetti un valore intero suggerisce che è possibile inserire più di una cifra contemporaneamente, ad es. `calculator.Enter(123)`, che non è ciò che vogliamo. Potremmo rilevare tali casi e generare eccezioni se il valore non è compreso nell'intervallo 0-9, ma ci sono modi migliori quando sappiamo che supporteremo solo dieci cifre (0,1,2,3,4,5,6,7,8,9).
2. Lo Statement non mostra chiaramente la relazione tra input e output. Naturalmente, in questo semplice caso, è abbastanza evidente che la somma è unaăconcatenazione di cifre immesse. In generale, tuttavia, non vogliamo che chiunque legga le nostre Specifiche in futuro debba indovinare queste cose.
3. Il nome dello Statement suggerisce che ciò che hai scritto è vero per qualsiasi valore, mentre in realtà è vero solo per cifre diverse da "0" poiché il comportamento per "0" è diverso (non importa quante volte inseriamo "0", il risultato è solo "0"). Ci sono alcuni buoni modi per comunicarlo.

Pertanto, propongo quanto segue:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
   (int)anyDigit1,
   (int)anyDigit2
  ),
  calculator.Display()
```

(continues on next page)

(continua dalla pagina precedente)

```
 );
}
```

Benjamin: Johnny, mi sono perso! Puoi spiegare cosa sta succedendo qui?

Johnny: Certo, cosa vuoi sapere?

Benjamin: Ad esempio, cosa ci fa qui questo tipo `DigitKeys`?

Johnny: Dovrebbe essere un'enumerazione (nota che non esiste ancora, supponiamo semplicemente di averla) per contenere tutte le possibili cifre che un utente può inserire, comprese nell'intervallo 0-9. Questo per garantire che l'utente non scriva `calculator.Enter(123)`. Invece di consentire ai nostri utenti di inserire qualsiasi numero e quindi di rilevare gli errori, diamo loro la possibilità di scegliere solo tra i valori validi.

Benjamin: Adesso ho capito. Che ne dici di `Any.OtherThan()` e `Any.Of()`? Cosa fanno?

Johnny: Sono metodi di unaăpiccola libreria di utilità che utilizzo quando scrivo Specifiche a livello di unità (unit-level). `Any.OtherThan()` restituisce qualsiasi valore dall'enumerazione oltre a quello passato come argomento. Pertanto, la chiamata `Any.OtherThan(DigitKeys.Zero)` significa "qualsiasi valore contenuto nell'enumerazione DigitKeys, ma non DigitKeys.Zero".

`Any.Of()` è più semplice -- restituisce semplicemente qualsiasi valore in un'enumerazione.

Notare che dicendo:

```
var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
var anyDigit1 = Any.Of<DigitKeys>();
var anyDigit2 = Any.Of<DigitKeys>();
```

Specifico esplicitamente che il primo valore inserito deve essere diverso da "0" e che questo vincolo non vale per la seconda cifra, la terza e così via..

A proposito, questa tecnica di utilizzo dei valori generati anziché dei valori letterali ha i suoi principi e vincoli che devi conoscere per utilizzarla in modo efficace. Lasciamo questo argomento per ora e prometto che ti terrò una lezione dettagliata più tardi. D'accordo?

Benjamin: È meglio che tu lo faccia, perché per ora mi sento un po' a disagio nel generare i valori -- sembra che la dichiarazione che stiamo scrivendo stia diventando meno deterministica in questo modo. L'ultima domanda -- che dire di quegli strani commenti che hai inserito nel codice? `GIVEN`? `WHEN`? `THEN`?

Johnny: Sì, questa è una convenzione che utilizzo non solo nello scrivere, ma anche nel pensare. Mi piace pensare a ogni comportamento in termini di tre elementi: presupposti (given-dato), trigger (when-quando) e risultato atteso (then-allora). Usando le parole, possiamo riassumere lo Statement che stiamo scrivendo nel modo seguente : "**Given-Data** unaăcalcolatrice, **when-quando** inserisco alcune cifre, la prima delle quali è diversa da zero, **then-allora** esse dovrebbero essere visualizzate tutte nell'ordine in cui sono stati immesse". Anche questo è qualcosa di cui ti parlerò più avanti.

Benjamin: Certo, per ora mi servono solo i dettagli sufficienti per poter andare avanti -- dei principi, dei pro e dei contro potremo parlare più tardi. A proposito, la seguente sequenza di cast sembra bruttina:

```
string.Format("{0}{1}{2}",
 (int)nonZeroDigit,
 (int)anyDigit1,
 (int)anyDigit2
)
```

Johnny: Ci torneremo su e lo renderemo "più intelligente" tra un secondo dopo aver reso vera questo statement. Per ora, abbiamo bisogno di qualcosa di ovvio. Qualcosa che sappiamo che funziona. Valutiamo questo Statement. Qual è il risultato?

Benjamin: Fallito: atteso "351", ma era "0".

Johnny: Bene, ora scriviamo del codice per rendere vero questo Statement. Per prima cosa introdurremo un'enumerazione di digit. Questa enumerazione conterrà la cifra che utilizziamo nello Statement (che è `DigitKeys.Zero`) e alcuni valori fasulli:

```
public enum DigitKeys
{
 Zero = 0,
 TODO1, //TODO - bogus value for now
 TODO2, //TODO - bogus value for now
 TODO3, //TODO - bogus value for now
 TODO4, //TODO - bogus value for now
}
```

Benjamin: Cosa significano tutti questi valori fasulli? Non dovremmo definire correttamente i valori per tutte le cifre che supportiamo?

Johnny: No, non ancora. Non abbiamo ancora uno Statement che indichi quali cifre sono supportate e che ci induca ad aggiungerle, giusto?

Benjamin: Dici che hai bisogno di uno Statement affinché un elemento sia in un enum?

Johnny: Questa è una specifica che stiamo scrivendo, ricordi? Dovrebbe dire da qualche parte quali cifre supportiamo, no?

Benjamin: È difficile essere d'accordo, voglio dire, posso vedere i valori nell'enumerazione, dovrei testare qualcosa quando non è coinvolta la complessità?

Johnny: Ancora una volta, non stiamo solo testando ma stiamo anche specificando. Cercherò di fornirti ulteriori argomentazioni in seguito. Per ora, abbi pazienza e nota che quando potremo specificare gli elementi enum, aggiungere tale Statement sarà quasi semplice.

Benjamin: OK.

Johnny: Ora passiamo all'implementazione. Giusto per ricordarti -- quello che abbiamo finora assomiglia a questo:

```
public class Calculator
{
 public const string InitialValue = "0";
 public string Display()
 {
  return InitialValue;
 }
}
```

Questo non supporta ancora la visualizzazione di più cifre (come abbiamo appena dimostrato, perché lo Statement che afferma che sono supportate si è rivelata falso). Quindi modifichiamo il codice per gestire questo caso:

```
public class Calculator
{
 public const string InitialValue = "0";
 private int _result = 0;

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
 }

 public string Display()
 {
  return _result.ToString();
 }
}
```

Johnny: Ora lo Statement è vero, quindi possiamo tornarci ed abbellirlo. Diamo una seconda occhiata:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
   (int)anyDigit1,
   (int)anyDigit2
  ),
  calculator.Display()
 );
}
```

Johnny: Ricordi che hai detto che non ti piace la parte in cui viene utilizzato`string.Format()`?

Benjamin: Sì, sembra poco leggibile.

Johnny: Estraiamo questa parte in unămetodo di utilità e rendiamola più generale: avremo bisogno di unămodo per costruire l'output visualizzato previsto in molti dei nostri futuri Statement. Ecco il mio metodo helper:

```
string StringConsistingOf(params DigitKeys[] digits)
{
 var result = string.Empty;

 foreach(var digit in digits)
 {
  result += (int)digit;
 }
 return result;
}
```

Notare che questo è più generale poiché supporta un numero qualsiasi di parametri. E lo Statement dopo questa estrazione assomiglia a questo:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
```

(continues on next page)

(continua dalla pagina precedente)

```
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  StringConsistingOf(nonZeroDigit, anyDigit1, anyDigit2),
  calculator.Display()
 );
}
```

Benjamin: Mi sembra migliore. Lo Statement viene comunque valutato come vero, il che significa che abbiamo capito bene, no?

Johnny: Non esattamente. Con mosse come questa, mi piace prestare particolare attenzione e ricontrollare se lo Statement descrive ancora il comportamento in modo accurato. Per assicurarci che sia ancora così, commentiamo il corpo del metodo `Enter()` e vediamo se questo Statement risulta comunque falso:

```
public void Enter(DigitKeys digit)
{
 //_result *= 10;
 //_result += (int)digit;
}
```

Benjamin: In esecuzione... Ok, adesso è falso. Era previsto"243", ottenuto "0".

Johnny: Bene, ora siamo abbastanza sicuri che funzioni bene. Decommentiamo le righe che abbiamo appena commentato e andiamo avanti.

Benjamin: Ma aspetta, c'è una cosa che mi preoccupa.

Johnny: Penso di saperlo -- mi chiedevo se l'avresti capito. Va avanti.

Benjamin: Ciò che mi preoccupa sono queste due righe:

```
public const string InitialValue = "0";
private int _result = 0;
```

Non è questa una duplicazione? Voglio dire, non è esattamente una duplicazione del codice, ma in entrambe le righe il valore "0" ha lo stesso intento. Non dovremmo rimuovere questa duplicazione in qualche modo?

Johnny: Sì, facciamolo. Preferirei cambiare `InitialValue` in `int` invece di string e usarlo. Ma non posso farlo in un unico passaggio poiché i due Statement dipendono dal fatto che `InitialValue` sia una stringa. se cambiassi semplicemente il tipo in `int`, guasterei quei test e l'implementazione e vorrei sempre risolvere una cosa alla volta.

Benjamin: Allora cosa facciamo?

Johnny: Bene, il mio primo passo sarebbe andare agli Statement che utilizzano `InitialValue` e utilizzare un metodo `ToString()`. Ad esempio, nello Statement `ShouldDisplayInitialValueWhenCreated()`, ho un'asserzione [affermazione]:

```
Assert.Equal(Calculator.InitialValue, displayedResult);
```

che posso modificare in:

```
Assert.Equal(Calculator.InitialValue.ToString(), displayedResult);
```

Benjamin: Ma chiamare `ToString()` su una `string` restituisce semplicemente lo stesso valore, qual è il punto?

Johnny: Il punto è rendere irrilevante il tipo di ciò che si trova sul lato sinistro di `.ToString()`. Quindi potrò modificare quel tipo senza guastare lo Statement. La nuova implementazione della classe `Calculator` sarà simile a questa:

```
public class Calculator
{
 public const int InitialValue = 0;
 private int _result = InitialValue;

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
 }

 public string Display()
 {
  return _result.ToString();
 }
}
```

Benjamin: Oh, capisco. E gli Statement vengono comunque valutati come veri.

Johnny: Sì. Facciamo un altro Statement?

11.4.3 Statement 3: La calcolatrice dovrebbe visualizzare solo una cifra zero se è l'unica cifra immessa, anche se viene immessa più volte

Johnny: Benjamin, dovrebbe essere facile per te, quindi vai avanti e provalo. Si tratta in realtà di una variante dello Statement precedente.

Benjamin: Fammi provare... ok, eccolo qui:

```
[Fact] public void
ShouldDisplayOnlyOneZeroDigitWhenItIsTheOnlyEnteredDigitEvenIfItIsEnteredMultipleTimes()
{
 //GIVEN
 var calculator = new Calculator();

 //WHEN
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);

 //THEN
 Assert.Equal(
  StringConsistingOf(DigitKeys.Zero),
  calculator.Display()
 );
}
```

Johnny: Bene, stai imparando in fretta! Valutiamo questo Statement.

Benjamin: Sembra che il nostro codice attuale soddisfi già lo Statement. Dovrei provare a commentare del codice per assicurarmi che questo Statement possa fallire proprio come hai fatto tu nello Statement precedente?

Johnny: Sarebbe una cosa saggia da fare. Quando uno ăStatement risulta vero senza richiedere la modifica del codice di produzione, è sempre sospetto. Proprio come hai detto tu, dobbiamo modificare il codice di produzione per un secondo per forzare questo Statement a diventare falso, poi annullare questa modifica per renderlo nuovamente vero. Questo non è così ovvio come in precedenza, quindi lasciamelo fare. Contrassegnerò tutte le righe aggiunte con il commento //+ in modo che tu possa vederle facilmente:

```
public class Calculator
{
 public const int InitialValue = 0;
 private int _result = InitialValue;
 string _fakeResult = "0"; //+

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
  if(digit == DigitKeys.Zero) //+
  {  //+
   _fakeResult += "0";  //+
  } //+
 }

 public string Display()
 {
  if(_result == 0)  //+
  {  //+
   return _fakeResult;  //+
  }  //+
  return _result.ToString();
 }
}
```

Benjamin: Wow, sembra che ci sia un sacco di codice solo per rendere falso lo Statement! Ne vale la pena? Annulleremo comunque l'intera modifica tra un secondo...

Johnny: Dipende da quanto vuoi sentirti sicuro. Direi che di solito ne vale la pena -- almeno sai di aver fatto tutto bene. Potrebbe sembrare molto lavoro, ma mi ci è voluto solo un minuto circa per aggiungere questo codice e immaginare di aver sbagliato e di aver dovuto eseguirne il debug in un ambiente di produzione. Ora, *quello* sarebbe una perdita di tempo.

Benjamin: Ok, penso di aver capito. Poiché abbiamo visto questo Statement diventare falso, annullerò questa modifica per renderlo nuovamente vero.

Johnny: Certo.

11.5 Epilogo

È ora di lasciare Johnny e Benjamin, almeno per ora. Avevo intenzione di rendere questo capitolo più lungo e di coprire tutte le altre operazioni, ma temo che questo lo renderebbe noioso. Si dovrebbe avere un'idea di come appare il ciclo TDD, soprattutto perché nel frattempo Johnny e Benjamin hanno avuto molte conversazioni su molti altri argomenti. Rivisiterò questi argomenti più avanti nel libro. Per ora, se ci si sente persi o non si è convinti su uno qualsiasi degli argomenti menzionati da Johnny, non c'è da preoccuparsi: non mi aspetto che siate esperti con nessuna delle tecniche mostrate in questo capitolo. Verrà il momento per quello.

CAPITOLO 12

Facciamo un po' d'ordine

Nell'ultimo capitolo c'è stata una conversazione vivace tra Johnny e Benjamin. Anche in una sessione così breve, Benjamin, in quanto principiante del TDD, aveva molte domande e molte cose da risolvere. Raccoglieremo tutte quelle domande a cui non è stata già data risposta e cercheremo di rispondere nei prossimi capitoli. Ecco le domande:

- Come dare un nome a uno Statement?
- Come iniziare a scrivere uno Statement?
- In che modo il TDD riguarda l'analisi e cosa significa questo "GIVEN-WHEN-THEN"?
- Qual è esattamente lo scopo di uno Statement? Una classe, un metodo o qualcos'altro?
- Qual è il ruolo della lista di TODO in TDD?
- Perché utilizzare valori generati anonimi anziché valori letterali come input di un comportamento specifico?
- Perché e come utilizzare la classe `Any`?
- Quale codice estrarre da uno Statement per i metodi di utilità condivisa?
- Perché un approccio così strano alla creazione di costanti enumerate?

Molte domande, vero? Sfortunatamente, il TDD ha questa barriera dingresso elevata, almeno per chi è abituato al modo tradizionale di scrivere codice. Ad ogni modo, questo è lo scopo di questo tutorial -- rispondere a queste domande e abbassare questa barriera. Cercheremo quindi di rispondere a queste domande una per una.

CAPITOLO 13

Come iniziare?

Ogni volta che mi sedevo con qualcuno che stava per scrivere codice in modalità Statement per la prima volta, la persona fissava lo schermo, poi me, quindi diceva: "e adesso?". È facile dire: "Sai scrivere codice, sai scrivere un test, solo che questa volta inizia con quest'ultimo anziché con il primo", ma per molte persone questo è qualcosa che li blocca completamente. Se siete uno di loro, non preoccupatevi -- non siete soli. Ho deciso di dedicare questo capitolo esclusivamente alle tecniche per avviare uno Statement in assenza di codice.

13.1 Si inizia con un buon nome

Ho già detto che uno Statement è una descrizione del comportamento espressa in codice. Un processo di pensiero che porta alla creazione di tale Statement eseguibile potrebbe assomigliare alla seguente sequenza di domande:

1. Qual è lo "scope" [l'ambito] del comportamento che sto cercando di specificare? Risposta di esempio: sto cercando di specificare un comportamento di una classe `Calculator`.
2. Qual è il comportamento di una classe `Calculator` che sto cercando di specificare? Risposta di esempio: dovrebbe visualizzare tutte le cifre immesse che non precedono gli zeri.
3. Come specificare questo comportamento tramite il codice? Risposta di esempio: `[Fact] public void ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes() ...` (ovvero un pezzo di codice).

Notare che prima di scrivere qualsiasi codice, ci sono almeno due domande a cui è possibile rispondere nel linguaggio umano. Spesso, rispondere a queste domande prima di iniziare a scrivere il codice dello Statement, facilita le cose. Anche se questo può ancora essere un processo impegnativo. Per applicare con successo questo consiglio, è necessaria una certa conoscenza su come denominare correttamente gli Statement. So che non tutti prestano attenzione a dare un nome ai propri Statement, soprattutto perché gli Statement sono spesso considerati cittadini di serie B -- finché funzionano e "dimostrano che il codice non contiene difetti", sono considerati sufficienti. Daremo un'occhiata ad alcuni esempi di nomi sbagliati e poi entreremo in alcune regole per una buona nomenclatura.

13.1.1 Conseguenze di una cattiva nomenclatura

Ho visto molte persone non preoccuparsi del nome dei loro Statement. Questo è un sintomo del fatto che si trattano le Specifiche come spazzatura o avanzi -- considero questo approccio pericoloso perché ho visto che porta a Specifiche difficili da mantenere e che assomigliano più a pezzi di codice messi insieme accidentalmente in fretta che a una sorta di "documentazione reale". Si immagini che la Specifica sia composta da Statement denominati in questo modo:

- `TrySendPacket()`
- `TrySendPacket2()`

- `testSendingManyPackets()`
- `testWrongPacketOrder1()`
- `testWrongPacketOrder2()`

e si vede quanto sia difficile rispondere alle seguenti domande:

1. Come si fa a sapere quale situazione descrive ciascuno Statement?
2. Come si fa a sapere se lo Statement descrive una singola situazione o più situazioni contemporaneamente?
3. Come si fa a sapere se le asserzioni contenute in quegli Statement sono quelle giuste presupponendo che ciascuno Statement sia stato scritto da qualcun altro o molto tempo fa?
4. Come si fa a sapere se lo Statement deve rimanere o essere rimosso dalle Specifiche quando si modifica la funzionalità descritta da questo Statement?
5. Se le modifiche apportate al codice di produzione rendono falso uno Statement, come si fa a sapere se lo Statement non è più corretta o il codice di produzione è sbagliato?
6. Come si fa a sapere se non si introdurrà uno Statement duplicato per uno scenario quando si aggiunge una Specifica creata da un altro membro del team?
7. Come si valuta, esaminando il report del tool runner, se la correzione di uno Statement non riuscito sarà facile o meno?
8. Cosa si risponde ai nuovi sviluppatori del team quando chiedono "a cosa serve questo Statement?"
9. Come si fa a sapere quando la Specifica è completa se non si riesce a distinguere dai nomi degli Statement quali comportamenti hai già coperto e quali no??

13.1.2 Cosa contiene un buon nome?

Per essere di qualche utilità, il nome di una Statement deve descriverne il comportamento previsto. Come minimo, dovrebbe descrivere cosa succede in quali circostanze. Diamo un'occhiata a uno dei nomi che Steve Freeman e Nat Pryce hanno inventato nel loro fantastico libro Growing Object-Oriented Software Guided By Tests:

```
notifiesListenersThatServerIsUnavailableWhenCannotConnectToItsMonitoringPort()
```

Notare alcune cose sul nome dello Statement:

1. Descrive un comportamento di un'istanza di una classe specifica. Da notare che non contiene il nome del metodo che attiva il comportamento, perché ciò che viene specificato non è un singolo metodo, ma il comportamento stesso (questo verrà trattato in maggior dettaglio nei prossimi capitoli). Il nome dello Statement indica semplicemente cosa fa un'istanza ("notifica agli ascoltatori che il server non è disponibile") in determinate circostanze ("quando non è possibile connettersi alla sua porta di monitoraggio"). È importante per me perché posso ricavare una descrizione del genere pensando alle responsabilità di una classe senza la necessità di conoscere le firme dei suoi metodi o il codice all'interno della classe. Quindi, questo è qualcosa che posso inventare prima di implementarlo -- devo solo sapere perché ho creato questa classe e basarmi su questa conoscenza.
2. Il nome è relativamente lungo. Veramente, veramente, **veramente** non c'è da preoccuparsi. Finché si descrive un singolo comportamento, direi che va bene. Ho visto persone esitare a dare nomi lunghi agli Statement perché cercavano di applicare a quei nomi le stesse regole dei nomi dei metodi nel codice di produzione. Nel codice di produzione, un nome di metodo lungo può essere un segno che il metodo ha troppe responsabilità o che viene utilizzato un livello di astrazione insufficiente per descrivere la funzionalità e che il nome potrebbe rivelare inutilmente dettagli di implementazione. La mia opinione è che queste due ragioni non si applicano tanto agli Statement. Nel caso degli Statement, i metodi non vengono invocati da nessuno oltre al runner automatico dei test, quindi non offuscheranno alcun codice che avrebbe bisogno di chiamarli con i loro nomi lunghi. Inoltre, i nomi degli statement non devono essere astratti come i nomi dei metodi del codice di produzione -- possono rivelare di più.

   In alternativa, potremmo inserire tutte le informazioni in un commento invece del nome dello Statement e lasciare il nome breve, in questo modo:

```
[Fact]
//Notifies listeners that server
//is unavailable when cannot connect
//to its monitoring port
public void Statement_002()
{
  //...
}
```

tuttavia, ci sono due aspetti negativi in questo. Innanzitutto, ora dobbiamo aggiungere un'ulteriore informazione (`Statement_002`) solo per soddisfare il compilatore, perché ogni metodo deve comunque avere un nome -- e di solito non esiste alcun valore che un essere umano possa derivare da un nome come `Statement_002`. Il secondo svantaggio è che quando l'istruzione diventa falsa, il test runner mostra la seguente riga: `Statement_002: FAILED` -- notare che tutte le informazioni incluse nel commento mancano dal rapporto dell'errore. Considero molto più prezioso ricevere un report del tipo:

`notifiesListenersThatServerIsUnavailableWhenCannotConnectToItsMonitoringPort: FAILED`

perché in tal caso, molte informazioni sulla dichiarazione fallita sono disponibili nel rapporto del test runner.

3. Usare un nome che descrive un singolo comportamento mi permette di scoprire velocemente perché lo Statement è risultato falso. Diciamo che uno Statement è vero quando inizio il refactoring, ma a un certo punto diventa falsa e il report nel runner assomiglia a questo: `TrySendingHttpRequest: FAILED` -- mi dice solo che è stato effettuato un tentativo di inviare una richiesta HTTP, ma, ad esempio, non mi dice se l'oggetto che ho specificato in quello Statement è una sorta di mittente che dovrebbe provare a inviare questa richiesta in alcune circostanze, o se è un destinatario che dovrebbe gestire tale richiesta correttamente. Per sapere cosa è andato storto, devo aprire il codice sorgente dello Statement. D'altra parte, quando ho uno Statement denominato `ShouldRespondWithAnAckWheneverItReceivesAnHttpRequest`, se diventa falso, so cosa non funziona -- l'oggetto non risponde più con un ACK a una richiesta HTTP. Questo potrebbe essere sufficiente per identificare quale parte del codice è difettosa e quale delle mie modifiche ha reso falso lo Statement.

13.1.3 La mia convenzione preferita

Esistono molte convenzioni per denominare gli Statement in modo appropriato. Il mio preferito è quello sviluppato da Dan North, in cui il nome di ogni istruzione inizia con la parola `Should` [Dovrebbe]. Quindi, ad esempio, chiamerei uno Statement:

`ShouldReportAllErrorsSortedAlphabeticallyWhenErrorsOccurDuringSearch()`

Il nome della Specifica (ovvero il nome della classe) risponde alla domanda "chi dovrebbe farlo?", ovvero quando ho una classe denominata `SortingOperation` e voglio dire che "dovrebbe ordinare tutti gli elementi in ordine crescente quando eseguita", lo dico in questo modo:

```
public class SortingOperationSpecification
{
 [Fact] public void
 ShouldSortAllItemsInAscendingOrderWhenPerformed()
 {
 }
}
```

Scrivendo quanto sopra, dico che "L'operazione di ordinamento *(deriva dal nome della classe della Specifica)* dovrebbe ordinare tutti gli elementi in ordine crescente quando eseguita *(questo deriva dal nome dello Statement)*".

La parola "should" [dovrebbe] è stata introdotta da Dan per indebolire l'affermazione che la segue e permettere così di mettere in discussione ciò che si sta affermando e porsi la domanda: "davvero dovrebbe?". Se questo causa incertezza, allora è giunto il momento di parlare con un esperto del settore e assicurarsi di capire bene cosa si deve realizzare. Se non si è madrelingua inglese, il prefisso "should" probabilmente avrà un'influenza minore -- questo è uno dei motivi per cui non insisto sul suo uso. Mi piace però[1].

[1] Ci sono anche alcuni argomenti contro l'uso della parola "should" [dovrebbe], ad es. di Kevlin Henney (vedere https://www.infoq.com/presentations/testing-communication).

Quando si inventa un nome, è importante concentrarsi principalmente sul risultato o sull'azione che ci si aspetta da un oggetto, non ad es. da uno dei suoi metodi. Se non lo si fa, potrebbe diventare rapidamente problematico. Ad esempio, uno dei miei colleghi ha specificato una classe `UserId` (che consisteva in un nome utente e alcune altre informazioni) e ha scritto il seguente nome per lo Statement relativo al confronto di due identificatori:

`EqualOperationShouldFailForTwoInstancesWithTheSameUserName()` [L'operazione uguale dovrebbe fallire per due istanze con lo stesso nome utente].

Si noti che questo nome non è scritto dalla prospettiva di un singolo oggetto, ma piuttosto dalla prospettiva di un'operazione che viene eseguita su di esso. Abbiamo smesso di pensare in termini di responsabilità degli oggetti e abbiamo iniziato a pensare in termini di correttezza operativa. Per riflettere la prospettiva dell'oggetto, questo nome dovrebbe essere qualcosa di più simile a:

`ShouldNotBeEqualToAnotherIdThatHasDifferentUserName()` [Non deve essere uguale a un altro ID con un username diverso].

Quando mi ritrovo ad avere problemi con la nomenclatura in questo modo, sospetto che potrebbe essere uno di questi casi:

1. Non sto specificando il comportamento di una classe, ma piuttosto il risultato di un metodo.
2. Sto specificando più di un comportamento.
3. Il comportamento è troppo complicato e quindi devo modificare il mio design (ne parleremo più avanti).
4. Sto nominando il comportamento di un'astrazione di livello troppo basso, inserendo troppi dettagli nel nome. Di solito arrivo a questa conclusione solo quando tutti i punti precedenti sono falsi.

13.1.4 Il nome non può davvero diventare troppo lungo?

Qualche paragrafo fa ho detto che non ci si dovrebbe preoccupare della lunghezza dei nomi degli Statement, ma devo ammettere che il nome può diventare occasionalmente troppo lungo. Una regola che cerco di seguire è che il nome di uno Statement dovrebbe essere più facile da leggere del suo contenuto. Pertanto, se mi occorre meno tempo per comprendere il senso di uno Statement leggendone il corpo che leggendone il nome, allora considero il nome troppo lungo. Se questo è il caso, provo ad applicare l'euristica sopra descritta per trovare e risolvere la causa principale del problema.

13.2 Iniziare riempiendo la struttura GIVEN-WHEN-THEN con l'ovvio

Questa tecnica può essere utilizzata come estensione della precedente (cioè iniziare con un buon nome), inserendo un'altra domanda nella sequenza di domande che abbiamo seguito l'ultima volta:

1. Qual è lo "scope" [l'ambito] del comportamento che sto cercando di specificare? Risposta di esempio: sto cercando di specificare il comportamento di una classe `Calculator`.
2. Qual è il comportamento di una classe `Calculator` che sto cercando di specificare? Risposta di esempio: dovrebbe visualizzare tutte le cifre immesse che non precedono gli zeri.
3. **Qual è il contesto ("GIVEN") del comportamento, l'azione ("WHEN") che lo innesca e la reazione attesa ("THEN") dell'oggetto specificato? Risposta di esempio: Se accendo la calcolatrice, quando inserisco qualsiasi cifra diversa da 0 seguita da qualsiasi cifra, queste dovrebbero essere visibili sul display**.
4. Come specificare questo comportamento tramite il codice? Risposta di esempio: `[Fact] public void ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes() ...` (ovvero un pezzo di codice).

In alternativa, può essere utilizzato senza la fase di denominazione, quando è più difficile trovare un nome che con una struttura GIVEN-WHEN-THEN. In altre parole, una struttura GIVEN-WHEN-THEN può essere facilmente derivata da un buon nome e viceversa.

Questa tecnica consiste nel prendere le parti GIVEN, WHEN e THEN e tradurle in codice in modo quasi letterale e con forza bruta (senza prestare attenzione alle classi, ai metodi o alle variabili mancanti), quindi aggiungere tutti i pezzi mancanti necessari per il codice da compilare ed eseguire.

13.2.1 Esempio

Proviamolo su un semplice problema di confronto tra due utenti in termini di uguaglianza. Assumiamo che due utenti debbano essere uguali tra loro se hanno lo stesso nome:

```
Given a user with any name
When I compare it to another user with the same name
Then it should appear equal to this other user
```

Cominciamo con la parte della traduzione. Ancora una volta, ricordarsi che stiamo cercando di rendere la traduzione il più letterale possibile senza prestare attenzione a tutti i pezzi mancanti per ora.

La prima riga:

```
Given a user with any name
```

può essere tradotta letteralmente nel seguente pezzo di codice:

```
var user = new User(anyName);
```

Notare che non abbiamo ancora la classe `User` e per ora non ci preoccupiamo di cosa sia realmente `anyName`. Va bene.

Poi la seconda riga:

```
When I compare it to another user with the same name
```

può essere scritta come:

```
user.Equals(anotherUserWithTheSameName);
```

Grande! Ancora una volta, non ci interessa cosa sia `anotherUserWithTheSameName`. Lo trattiamo come un segnaposto. Ora l'ultima riga:

```
Then it should appear equal to this other user
```

e la sua traduzione nel codice:

```
Assert.True(usersAreEqual);
```

Ok, ora che la traduzione letterale è completa, mettiamo insieme tutte le parti e vediamo cosa manca per compilare questo codice:

```
[Fact] public void
ShouldAppearEqualToAnotherUserWithTheSameName()
{
  //GIVEN
  var user = new User(anyName);

  //WHEN
  user.Equals(anotherUserWithTheSameName);

  //THEN
  Assert.True(usersAreEqual);
}
```

Come ci aspettavamo, questo non viene compilato. In particolare, il nostro compilatore potrebbe indicarci le seguenti lacune:

1. La variabile `anyName` non è dichiarata.
2. L'oggetto `anotherUserWithTheSameName` non è dichiarato.
3. La variabile `usersAreEqual` non è dichiarata e non contiene il risultato del confronto.

4. Se questa è la nostra prima istruzione, potremmo non avere nemmeno la classe `User` definita.

Il compilatore ha creato per noi una sorta di piccolo elenco di TODO, il che è carino. Notare che sebbene non disponiamo di un pezzo di codice da compilare, colmare le lacune per farlo compilare si riduce a fare alcune banali dichiarazioni e assegnazioni:

1. `anyName` può essere definito come:

   ```
   var anyName = Any.String();
   ```

2. `anotherUserWithTheSameName` può essere definito come:

   ```
   var anotherUserWithTheSameName = new User(anyName);
   ```

3. `usersAreEqual` può essere definito come una variabile a cui assegniamo il risultato del confronto:

   ```
   var usersAreEqual = user.Equals(anotherUserWithTheSameName);
   ```

4. Se la classe `User` non esiste ancora, possiamo aggiungerla semplicemente affermando:

   ```
   public class User
   {
     public User(string name) {}
   }
   ```

Rimettere il tutto insieme, dopo aver colmato le lacune, ci dà:

```
[Fact] public void
ShouldAppearEqualToAnotherUserWithTheSameName()
{
  //GIVEN
  var anyName = Any.String();
  var user = new User(anyName);
  var anotherUserWithTheSameName = new User(anyName);

  //WHEN
  var usersAreEqual = user.Equals(anotherUserWithTheSameName);

  //THEN
  Assert.True(usersAreEqual);
}
```

E questo è tutto -- lo Statement stesso è completo!

13.3 Iniziare dalla fine

Questa è una tecnica che suggerisco alle persone che sembrano non avere idea di come iniziare. L'ho preso dal libro di Kent Beck "Test-Driven Development by Example". A prima vista sembra buffo, ma a volte l'ho trovato piuttosto potente. Il trucco sta nello scrivere lo Statement "all'indietro", cioè iniziando da ciò che verifica il risultato (in termini di struttura *GIVEN-WHEN-THEN*, diremmo che iniziamo con la nostra parte *THEN*).

Funziona bene quando siamo abbastanza sicuri di quale dovrebbe essere il risultato nel nostro scenario, ma non così sicuri di come arrivarci.

13.3.1 Esempio

Immaginiamo di scrivere una classe contenente le regole per concedere o negare l'accesso a una funzionalità di reporting. Questa funzionalità di reporting si basa sui ruoli. Non abbiamo idea di come dovrebbe essere l'API e di come scrivere il nostro Statement, ma sappiamo una cosa: nel nostro dominio l'accesso può essere concesso o negato. Prendiamo il primo caso che ci viene in mente -- il caso di "accesso concesso" -- e, mentre torniamo indietro dalla fine, iniziamo con la seguente asserzione:

```
//THEN
Assert.True(accessGranted);
```

Ok, quella parte è stata facile, ma abbiamo fatto qualche progresso? Ovviamente l'abbiamo fatto -- ora abbiamo codice che non viene compilato, con l'errore causato dalla variabile `accessGranted`. Ora, a differenza dell'approccio precedente in cui traducevamo una struttura GIVEN-WHEN-THEN in uno Statement, il nostro obiettivo non è quello di completare questa compilazione il prima possibile. Dobbiamo invece rispondere alla domanda: come faccio a sapere se l'accesso è concesso o meno? La risposta: è il risultato dell'autorizzazione del ruolo consentito. Ok, quindi scriviamolo semplicemente nel codice, ignorando tutto ciò che ci ostacola:

```
//WHEN
var accessGranted
 = access.ToReportingIsGrantedTo(roleAllowedToUseReporting);
```

Per ora, provare a resistere all'impulso di definire una classe o una variabile per rendere felice il compilatore, poiché ciò potrebbe portare fuori strada e distogliere l'attenzione da ciò che è importante. La chiave per eseguire con successo il TDD è imparare a utilizzare qualcosa che non esiste ancora come se esistesse e non preoccuparsi finché non sia necessario.

Notare che non sappiamo cosa sia `roleAllowedToUseReporting`, né sappiamo cosa significhi l'oggetto `access`, ma ciò non ci ha impedito di scrivere questa riga. Inoltre, il metodo `ToReportingIsGrantedTo()` ci è stato semplicemente tolto dalla testa. Non è definito da nessuna parte, semplicemente aveva senso scriverlo così, perché è la traduzione più diretta di ciò che avevamo in mente.

Ad ogni modo, questa nuova riga risponde alla domanda su da dove prendiamo il valore `accessGranted`, ma ci fa anche porre ulteriori domande:

1. Da dove viene la variabile `access`?
2. Da dove viene la variabile `roleAllowedToUseReporting`?

Per quanto riguarda `access`, non abbiamo nulla di specifico da dire a parte il fatto che si tratta di un oggetto di una classe non ancora definita. Ciò che dobbiamo fare ora è fingere di avere una classe del genere (ma non definiamola ancora). Come lo chiamiamo? Il nome dell'istanza è `access`, quindi è abbastanza semplice chiamare la classe `Access` e istanziarla nel modo più semplice a cui possiamo pensare:

```
//GIVEN
var access = new Access();
```

Ora passiamo a `roleAllowedToUseReporting`. La prima domanda che viene in mente quando si esamina la questione è: quali ruoli sono autorizzati a utilizzare il reporting? Supponiamo che nel nostro dominio si tratti di un Administrator [amministratore] o di un Auditor [revisore]. Let's assume that in our domain, this is either an Administrator or an Auditor. Quindi, sappiamo quale sarà il valore di questa variabile. Per quanto riguarda il tipo, ci sono vari modi in cui possiamo modellare un ruolo, ma quello più ovvio per un tipo che ha pochi valori possibili è un enum[2]. Quindi:

```
//GIVEN
var roleAllowedToUseReporting = Any.Of(Roles.Admin, Roles.Auditor);
```

E così, procedendo all'indietro, siamo arrivati alla soluzione finale (nel codice seguente, ho già dato un nome allo Statement -- questo è l'ultimo passaggio):

```
[Fact] public void
ShouldAllowAccessToReportingWhenAskedForEitherAdministratorOrAuditor()
{
 //GIVEN
 var roleAllowedToUseReporting = Any.Of(Roles.Admin, Roles.Auditor);
 var access = new Access();

 //WHEN
```

(continues on next page)

[2] Questo approccio di scegliere un singolo valore tra diversi utilizzando `Any.From()` non sempre funziona bene con le enumerazioni. A volte risulta migliore un test parametrizzato (una "theory" A volte è migliore un test parametrizzato). Questo argomento verrà trattato in uno dei prossimi capitoli.

(continua dalla pagina precedente)

```
 var accessGranted
     = access.ToReportingIsGrantedTo(roleAllowedToUseReporting);

 //THEN
 Assert.True(accessGranted);
}
```

Utilizzando ciò che abbiamo imparato formulando lo Statement, è stato facile dargli un nome.

13.4 Iniziare invocando un metodo se c'è

Se vengono soddisfatte le precondizioni per questo approccio, è il più semplice e lo uso molto[3].

Molte volte dobbiamo aggiungere una nuova classe che implementa un'interfaccia già esistente. L'interfaccia impone quali metodi deve supportare la nuova classe. Se le firme dei metodi sono già decise, possiamo iniziare il nostro Statement con una chiamata a uno dei metodi e poi capire il resto del contesto di cui abbiamo bisogno per farlo funzionare correttamente.

13.4.1 Esempio

Immaginiamo di avere un'applicazione che, tra le altre cose, gestisce l'importazione di un database esistente esportato da un'altra istanza dell'applicazione. Dato che il database è di grandi dimensioni e la sua importazione può essere un processo lungo, viene visualizzata una "message box" ogni volta che un utente esegue l'import. Supponendo che il nome dell'utente sia Johnny, la "message box" visualizza il messaggio "Johnny, please sit down and enjoy your coffee for a few minutes as we take time to import your database". La classe che lo implementa è simile a:

```
public class FriendlyMessages
{
  public string
  HoldOnASecondWhileWeImportYourDatabase(string userName)
  {
    return string.Format("{0}, "
      + "please sit down and enjoy your coffee "
      + "for a few minutes as we take time "
      + "to import your database",
      userName);
  }
}
```

Immaginiamo ora di voler fornire una versione di prova dell'applicazione con alcune funzionalità disabilitate, una delle quali è l'importazione del database. Una delle cose che dobbiamo fare è visualizzare un messaggio che informa che si tratta di una versione di prova e che la funzione di importazione è bloccata. Possiamo farlo estraendo un'interfaccia dalla classe `FriendlyMessages` e implementando questa interfaccia in una nuova classe utilizzata quando l'applicazione viene eseguita come versione di prova. L'interfaccia estratta è simile alla seguente:

```
public interface Messages
{
  string HoldOnASecondWhileWeImportYourDatabase(string userName);
}
```

Quindi la nostra nuova implementazione è costretta a supportare il metodo `HoldOnASecondWhileWeImportYourDatabase()` [*Aspetta un attimo mentre importiamo il tuo database*]. Chiamiamo questa nuova classe `TrialVersionMessages` (ma non la creiamo ancora!) e possiamo scrivere uno Statement per il suo comportamento. Supponendo di non sapere da dove cominciare, iniziamo semplicemente creando un oggetto della classe (conosciamo già il nome) e invocando il metodo che già sappiamo di dover implementare:

[3] Cercare i dettagli nel capitolo 2.

```
[Fact]
public void TODO()
{
 //GIVEN
 var trialMessages = new TrialVersionMessages();

 //WHEN
 trialMessages.HoldOnASecondWhileWeImportYourDatabase();

 //THEN
 Assert.True(false); //so we don't forget this later
}
```

Come potete vedere, abbiamo aggiunto un'asserzione che alla fine non riesce mai a ricordarci che lo Statement non è ancora finito. Poiché non disponiamo ancora di asserzioni rilevanti, lo Statement sarà considerato vero non appena verrà compilata ed eseguita e potremmo non notare che è incompleto. Allo stato attuale, lo Statement non viene comunque compilato, perché non esiste ancora una classe `TrialVersionMessages`. Creiamone una con la minima implementazione possibile:

```
public class TrialVersionMessages : Messages
{
 public string HoldOnASecondWhileWeImportYourDatabase(string userName)
 {
   throw new NotImplementedException();
 }
}
```

Notare che in questa classe è presente solo la quantità di implementazione richiesta per compilare questo codice. Tuttavia, lo Statement non sarà ancora compilato. Questo perché il metodo `HoldOnASecondWhileWeImportYourDatabase()` accetta un argomento di tipo stringa e non ne abbiamo passato nessuna nello Statement. Questo ci porta a chiederci quale sia questo argomento e quale sia il suo ruolo nel comportamento attivato dal metodo `HoldOnASecondWhileWeImportYourDatabase()`. Sembra che sia un nome utente. Pertanto, possiamo aggiungerlo allo Statement in questo modo:

```
[Fact]
public void TODO()
{
 //GIVEN
 var trialMessages = new TrialVersionMessages();
 var userName = Any.String();

 //WHEN
 trialMessages.
  HoldOnASecondWhileWeImportYourDatabase(userName);

 //THEN
 Assert.True(false); //to remember about it
}
```

Ora, questo viene compilato ma è considerato falso a causa dell'asserzione di guardia che abbiamo messo alla fine. Il nostro obiettivo è sostituirla con un'asserzione adeguata per il risultato atteso. Il valore restituito della chiamata a `HoldOnASecondWhileWeImportYourDatabase` è un messaggio di tipo stringa, quindi tutto ciò che dobbiamo fare è visualizzare il messaggio che ci aspettiamo in caso di versione di prova:

```
[Fact]
public void TODO()
{
 //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
 var trialMessages = new TrialVersionMessages();
 var userName = Any.String();
 var expectedMessage =
  string.Format(
    "{0}, better get some pocket money and buy a full version!",
    userName);

 //WHEN
 var message = trialMessages.
  HoldOnASecondWhileWeImportYourDatabase(userName);

 //THEN
 Assert.Equal(expectedMessage, message);
}
```

Tutto ciò che resta da fare è trovare un buon nome per lo Statement. Questo non è un problema poiché abbiamo già specificato il comportamento desiderato nel codice, quindi possiamo semplicemente riassumerlo in qualcosa come `ShouldCreateAPromptForFullVersionPurchaseWhenAskedForImportDatabaseMessage()`.

13.5 Riepilogo

Quando sono bloccato e non so come iniziare a scrivere un nuovo Statement che fallisce, le tecniche di questo capitolo mi aiutano a spingere le cose nella giusta direzione. Notare che gli esempi forniti sono semplicistici e basati sul presupposto che esista un solo oggetto che accetta un qualche tipo di parametro di input e restituisce un risultato ben definito. Tuttavia, questo non è il modo in cui è costruito la maggior parte del mondo orientato agli oggetti. In quel mondo, abbiamo spesso oggetti che comunicano con altri oggetti, inviano messaggi, invocano metodi l'uno dall'altro e questi metodi spesso non hanno alcun valore di ritorno ma sono invece dichiarati come `void`. Anche se tutte le tecniche descritte in questo capitolo funzioneranno comunque in questo caso e le rivisiteremo non appena impareremo come eseguire TDD nel più ampio mondo orientato agli oggetti (dopo l'introduzione del concetto di oggetti mock nella Parte 2). Qui ho cercato di mantenerlo semplice.

CAPITOLO 14

In che modo il TDD riguarda l'analisi e cosa significa "GIVEN-WHEN-THEN"?

Durante il lavoro sul codice della calcolatrice, Johnny ha menzionato che il TDD riguarda, tra le altre cose, l'analisi. Questo capitolo esplora ulteriormente questo concetto. Iniziamo rispondendo alla seguente domanda:

14.1 Esiste qualcosa in comune tra analisi e il TDD?

Da Wikipedia:

> L'analisi è il processo di suddivisione di un argomento o di una sostanza complessa in parti più piccole per ottenerne una migliore comprensione.

Pertanto, affinché il TDD riguardi lanalisi, dovrebbe soddisfare due condizioni:

1. Dovrebbe trattarsi di un processo di suddivisione di un argomento complesso in parti più piccole
2. Dovrebbe consentire di acquisire una migliore comprensione di parti così piccole

Nella storia di Johnny, Benjamin e Jane, ho incluso una parte in cui analizzano i requisiti utilizzando esempi concreti. Johnny ha spiegato che questa è una parte di un processo chiamato "Acceptance Test-Driven Development". Questo processo, seguito dai tre personaggi, soddisfaceva entrambe le condizioni menzionate affinché potesse essere considerato analitico. Ma che dire del TDD in sé?

Anche se nella storia ho utilizzato parti del processo ATDD per rendere la parte di analisi più ovvia, cose simili accadono a livelli puramente tecnici. Ad esempio, quando si avvia lo sviluppo con una dichiarazione non valida a livello di applicazione (vale a dire che copre il comportamento di un'applicazione nel suo insieme. Parleremo più avanti dei livelli di granularità degli Statement. Per ora, l'unica cosa che devi sapere è che il cosiddetto "unit tests level" non è l'unico livello di granularità su cui scriviamo gli Statement), potremmo riscontrare una situazione in cui dobbiamo chiamare un metodo web e creare un'asserzione sul suo risultato. Questo ci fa pensare: come dovrebbe essere chiamato questo metodo? Quali sono gli scenari che supporta? Cosa mi aspetto che ne esca? Come dovrei, come utente, essere avvisato degli errori? Molte volte, questo ci porta o a unaăconversazione (se c'è un altro "attore" che deve essere coinvolto nella decisione) o a ripensare le nostre ipotesi. Lo stesso vale nello "unit level" - se una classe implementa una regola di dominio, potrebbero esserci alcune buone domande relative al dominio derivanti dal tentativo di scrivere uno Statement per esso. Se una classe implementa una regola tecnica, potrebbero esserci alcune domande tecniche da discutere con altri sviluppatori, ecc. In questo modo otteniamo una migliore comprensione dell'argomento che stiamo analizzando, il che fa sì che il TDD soddisfi il secondo dei due requisiti per essere un metodo di analisi.

Ma per quanto riguarda il primo requisito? Che ne dite di suddividere una logica complessa in parti più piccole?

Se si torna alla storia di Johnny e Benjamin, si noterà che quando parlavano con un cliente e scrivevano il codice, usavano una lista di TODO [cose da fare]. Questo elenco è stato inizialmente compilato con gli scenari escogitati, ma in seguito avrebbero aggiunto unità di lavoro più piccole. Quando faccio TDD, faccio lo stesso, essenzialmente scomponendo argomenti complessi in elementi più piccoli e inserendoli nella lista di TODO (questa è una delle pratiche che servono alla scomposizione. L'altro è il mocking, ma per ora lasciamo perdere). Grazie a questo, posso concentrarmi su una cosa alla volta, cancellando un elemento dopo l'altro dalla lista una volta terminato. Se imparo qualcosa di nuovo o incontro un nuovo problema che richiede la nostra attenzione, posso aggiungerlo alla lista dei TODO e tornarci più tardi, per ora continuo il mio lavoro sull'elemento su cui mi sto concentrando.

Un esempio di elenco di cose da fare (TODO) nel bel mezzo di un'attività di implementazione potrebbe assomigliare a questo (non è da leggere, l'ho inserito qui solo per dare un'idea: non dovreste nemmeno capire di cosa trattano gli elementi della lista):

1. ~~Creare un "entry point" [*punto di ingresso*] al modulo (astrazione di livello superiore)~~
2. ~~Implementare il flusso di lavoro principale del modulo~~
3. ~~Implementare l'interfaccia `Message`~~
4. ~~Implementare l'interfaccia `MessageFactory`~~
5. Implementare l'interfaccia `ValidationRules`
6. ~~Implementare il comportamento richiesto dal metodo Wrap nella classe `LocationMessageFactory`~~
7. Implementare il comportamento richiesto dal metodo ValidateWith nella classe `LocationMessage` per il campo Speed
8. Implementare il comportamento richiesto dal metodo ValidateWith nella classe `LocationMessage` per il campo Age
9. Implementare il comportamento richiesto dal metodo ValidateWith nella classe `LocationMessage` per il campo Sender

Notare che alcuni elementi sono già cancellati come completati, mentre altri rimangono in sospeso e in attesa di essere risolti. Tutti questi elementi sono ciò che l'articolo su Wikipedia chiama "parti più piccole" - il risultato della scomposizione di un argomento più ampio.

Per me, gli argomenti forniti sono sufficienti per pensare che il TDD riguardi l'analisi. La domanda successiva è: esistono strumenti che possiamo utilizzare per aiutare e informare questa parte di analisi del TDD? La risposta è sì e sono stati già visti entrambi in questo libro, quindi ora li esamineremo più da vicino.

14.2 Gherkin

Affamati? Peccato, perché il Gherkin [*cetriolino*] di cui vi parlerò non è commestibile. È unaănotazione e unămodo di pensare ai comportamenti della parte di codice specificata. Può essere applicato a diversi livelli di granularità -- qualsiasi comportamento, sia di un intero sistema che di una singola classe, può essere descritto utilizzando Gherkin.

Abbiamo già usato questa notazione, semplicemente non l'abbiamo chiamata così. Gherkin è la struttura GIVEN-WHEN-THEN che si puoò vedere ovunque, anche come commenti negli esempi di codice. Questa volta gli stampiamo un nome e lo analizziamo ulteriormente.

In Gherkin, la descrizione di un comportamento è costituita principalmente da tre parti:

1. Given -- un contesto
2. When -- una causa
3. Then -- un effetto

In altre parole, l'enfasi è sulla causalità in un dato contesto. C'è anche una quarta parola chiave: `And`[1] -- possiamo usarla per aggiungere più contesto, più cause o più effetti. Tra poco si avrà la possibilità di vedere un esempio.

Come ho detto, ci sono diversi livelli a cui è applicabile. Ecco un esempio di tale descrizione del comportamento dal punto di vista dell'utente finale (detto acceptance-level Statement):

[1] Alcuni sostengono che esistano altre parole chiave, come `But` e `Or`. Tuttavia, non avremo bisogno di ricorrere ad essi, quindi ho deciso di ignorarli in questa descrizione.

```
Given a bag of tea costs $20
And there is a discount saying "pay half for a second bag"
When I buy two bags
Then I should be charged $30
```

Ed eccone uno a livello di unità (notare nuovamente la riga che inizia con "And" che aggiunge al contesto):

```
Given a list with 2 items
When I add another item
And check items count
Then the count should be 3
```

A livello di accettazione inseriamo tali descrizioni di comportamento insieme al codice come un unico insieme (se questo non dice niente, esaminare tool come SpecFlow o Cucumber o FIT per avere alcuni esempi), a livello di unità la descrizione di solito non è scritta in modo letterale, ma piuttosto è tradotto e scritto solo sotto forma di codice sorgente. Tuttavia, la struttura di GIVEN-WHEN-THEN è utile quando si pensa ai comportamenti richiesti da uno o più oggetti, come abbiamo visto quando abbiamo parlato di iniziare dallo Statement anziché dal codice. Mi piace inserire la struttura in modo esplicito nei miei Statement -- trovo che aiuti a renderle più leggibili[2]. Quindi la maggior parte dei miei Statement a livello di unità seguono questo template:

```
[Fact]
public void Should__BEHAVIOR__()
{
  //GIVEN
  ...context...

  //WHEN
  ...trigger...

  //THEN
  ...assertions etc....
}
```

A volte le sezioni WHEN e THEN non sono così facilmente separabili -- allora le unisco, come nel caso del seguente Statement che specifica che un oggetto genera un'eccezione quando viene chiesto di memorizzare null:

```
[Fact]
public void ShouldThrowExceptionWhenAskedToStoreNull()
{
  //GIVEN
  var safeList = new SafeList();

  //WHEN - THEN
  Assert.Throws<Exception>(
    () => safeList.Store(null)
  );
}
```

Pensando in termini di queste tre parti del comportamento, possiamo arrivare a circostanze (GIVEN) diverse in cui il comportamento ha luogo, o ad altre che sono necessarie. Lo stesso vale per i trigger (WHEN) e gli effetti (THEN). Se ci viene in mente qualcosa di simile, lo aggiungiamo alla lista dei TODO per rivisitarlo in seguito.

[2] Seb Rose ha scritto un post sul blog in cui si pronuncia contro i commenti `//GIVEN //WHEN //THEN` e afferma di utilizzare solo righe vuote per separare le tre sezioni, vedere http://claysnow.co.uk/unit-tests-are-your-specification/

14.3 La lista TODO... di nuovo!

Come ho scritto prima, una lista TODO è unărepository per il nostro lavoro differito. Ciò include tutto ciò che ci viene in mente quando scriviamo o pensiamo a uno Statement, ma non fa parte dello Statement attuale che stiamo scrivendo. Da un lato, non vogliamo dimenticarlo, dall'altro non vogliamo che ci perseguiti e ci distragga dal nostro compito attuale, quindi lo scriviamo il prima possibile e continuiamo con il nostro lavoro. Quando abbiamo finito, prendiamo un altro elemento dalla lista dei TODO e iniziamo a lavorarci sopra.

Immaginare di scrivere un codice logico che consenta l'accesso agli utenti quando sono dipendenti di uno zoo, ma neghi l'accesso se sono semplicemente visitatori. Poi, dopo aver iniziato a scrivere uno Statement, ci rendiamo conto che anche i dipendenti possono essere visitatori -- ad esempio, potrebbero scegliere di visitare lo zoo con le loro famiglie durante le vacanze. Tuttavia, valgono le due regole precedenti, quindi per evitare di essere distratti da questo terzo scenario, possiamo aggiungerlo rapidamente come elemento alla lista dei TODO (come "TODO: cosa succede se qualcuno è un dipendente, ma viene allo zoo come visitatore?") e terminare lo Statement corrente. Una volta terminato, si ouò sempre tornare all'elenco degli elementi rinviati e scegliere l'elemento successivo su cui lavorare.

Ci sono due domande importanti relative lle liste di TODO: "cosa dovremmo aggiungere esattamente come elemento della lista TODO?" e "Come gestire in modo efficiente la lista TODO?". Ci occuperemo ora di queste due domande.

14.3.1 Cosa inserire in una lista TODO?

Tutto ciò di cui abbiamo bisogno venga affrontato, ma non rientra nell'ambito dello Statement corrente. Questi elementi potrebbero essere correlati all'implementazione di metodi non implementati, all'aggiunta di intere funzionalità (tali elementi vengono solitamente ulteriormente suddivisi in sottoattività più dettagliate non appena iniziamo a implementarle), potrebbero essere promemoria per dare un'occhiata migliore a qualcosa (ad esempio "investigare su qual è la politica di questo componente per la registrazione degli errori") o domande sul dominio da cui è necessario ottenere una risposta. Se tendiamo a lasciarci trasportare troppo dalla programmazione e a perdere il pranzo, possiamo anche aggiungere unăpromemoria ("TODO: andare a pranzo!"). Non ho mai riscontrato un caso in cui avessi bisogno di condividere questo elenco di cose da fare con qualcun altro, quindi tendo a trattarlo come il mio taccuino. Consiglio di fare lo stesso - la lista è personale!

14.3.2 Come scegliere gli elementi da una lista TODO?

Quale elemento scegliere da una lista TODO quando ne abbiamo diversi? Non ho una regola chiara, anche se tendo a prendere in considerazione i seguenti fattori:

1. Rischio -- se ciò che imparo implementando o discutendo un particolare elemento della lista può avere un grande impatto sulla progettazione o sul comportamento del sistema, tendo a scegliere per primi tali elementi. Un esempio di tale situazione è quando inizio a implementare la validazione di unaărichiesta che raggiunge la mia applicazione e desidero restituire errori diversi a seconda di quale parte della richiesta sia sbagliata. Poi, durante lo sviluppo, potrei scoprire che più di parti della richiesta possono essere sbagliate contemporaneamente e devo rispondere a una domanda: quale codice di errore dovrebbe essere restituito in questo caso? O forse i codici restituiti dovrebbero essere accumulati per tutte le validazioni e poi restituiti sotto forma di lista?
2. Difficoltà -- a seconda della mia condizione mentale (quanto sono stanco, quanto rumore c'è attorno alla mia scrivania ecc.), tendo a scegliere gli elementi con difficoltà che meglio si adattano a questa condizione. Ad esempio, dopo aver finito un oggetto che richiede molta riflessione e comprensione, tendo ad affrontare alcuni oggetti piccoli e facili per sentire il vento che soffia nelle mie vele e riposarmi un po'.
3. Completezza -- in parole povere, quando finisco di testare un caso "if", di solito prendo il successivo "else". Ad esempio, dopo aver terminato di implementare uno Statement che dice che qualcosa dovrebbe restituire vero per valori inferiori a 50, l'elemento successivo da prendere in considerazione è il caso "maggiore o uguale a 50". Di solito, quando inizio a testare una classe, prendo gli elementi relativi a questa classe finché non li esaurisco, quindi passo ad altro.

Naturalmente, una lista TODO è solo una delle fonti di tali elementi TODO. In genere, durante la ricerca di elementi da svolgere, esamino le seguenti fonti di elementi nel seguente ordine:

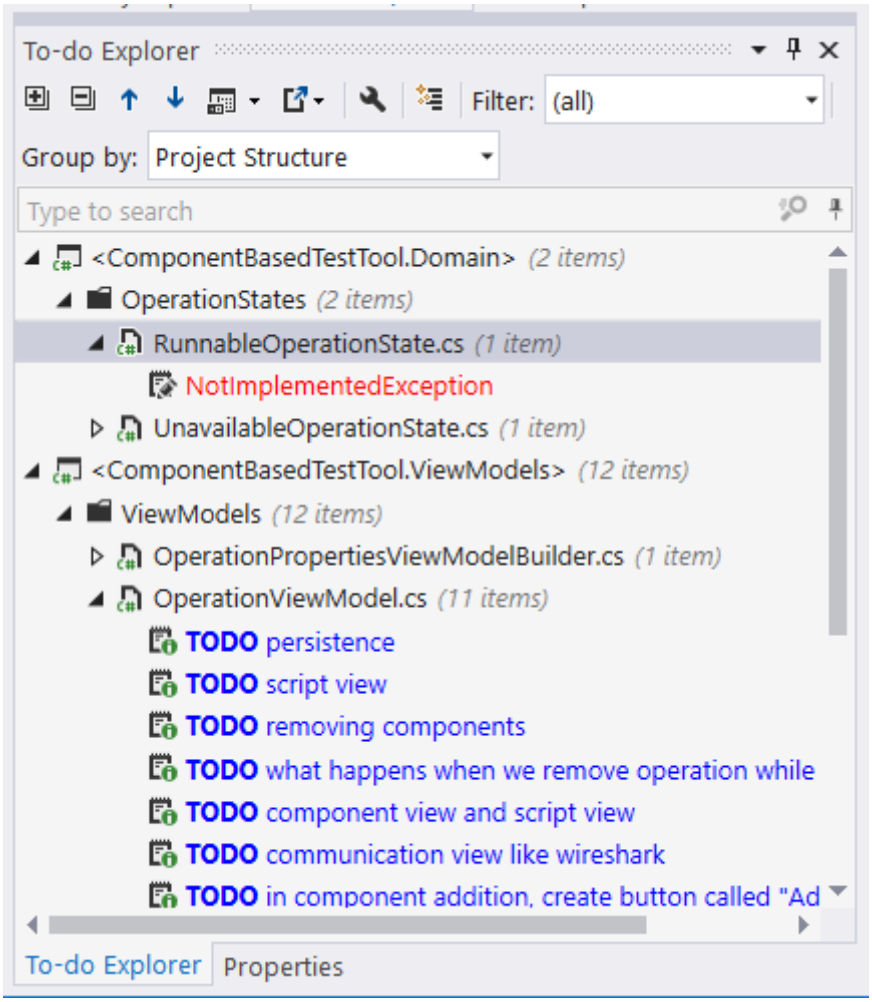
1. errori del compilatore,
2. Dichiarazioni false,
3. La mia lista TODO.

14.3.3 Dove mettere la lista TODO?

Ho riscontrato due modi per mantenere una lista di TODO. Il primo è su un foglio di carta. Lo svantaggio è che ogni volta che devo aggiungere qualcosa alla lista, devo togliere le mani dalla tastiera, prendere una penna o una matita e poi tornare a scrivere codice. Inoltre, l'unico modo in cui un elemento TODO scritto su un foglio di carta può dirmi a quale punto del mio codice è correlato è (ovviamente) tramite il suo testo. L'aspetto positivo della carta è che è di gran lunga uno dei migliori strumenti per disegnare, quindi quando è meglio archiviare la mia voce TODO come diagramma o disegno (cosa che non accade troppo spesso, ma a volte sì), utilizzo la carta e penna.

La seconda alternativa consiste nell'utilizzare la funzionalità della lista TODO integrata in un IDE. La maggior parte degli IDE, come Visual Studio (e il plug-in Resharper ha una versione migliorata), Xamarin Studio, IntelliJ o IDE basati su Eclipse hanno tale funzionalità. Le regole sono semplici -- inserisco commenti speciali (p. es. `//TODO fare qualcosa`) nel codice e unaăvista speciale nel mio IDE li aggrega, permettendomi di passare a ciascun elemento in un secondo momento. Questo è il mio modo principale di mantenere una lista di TODO, perché:

1. Non mi obbligano a togliere le mani dalla tastiera per aggiungere un elemento alla lista.
2. Posso inserire un elemento TODO in un determinato punto del codice in cui ha senso e poi tornarci successivamente con un clic del mouse. Questo, oltre ad altri vantaggi, permette di scrivere appunti più brevi che se dovessi farlo su carta. Per esempio, un elemento TODO che dice "TODO: cosa succede se si genera un'eccezione?" sembra fuori posto su un foglio di carta, ma se aggiunto come commento al mio codice nel posto giusto, è sufficiente.
3. Molti elenchi di cose da fare aggiungono automaticamente elementi per determinate cose che accadono nel codice. Per esempio in C#, quando devo ancora implementare un metodo generato automaticamente dall'IDE, il suo corpo è solitamente costituito da una riga che genera un'eccezione `NotImplementedException`. Indovinate un po' -- le occorrenze `NotImplementedException` vengono aggiunte automaticamente alla lista TODO, quindi non devo aggiungere manualmente elementi all'elenco TODO per implementare i metodi in cui si verificano.



L'elenco TODO mantenuto nel codice sorgente presenta un piccolo inconveniente: dobbiamo ricordarci di cancellare l'elenco quando finiamo di lavorarci o potremmo finire per inviare gli elementi TODO al repository del controllo sorgente insieme al resto del codice sorgente. Tali elementi TODO rimanenti potrebbero accumularsi nel codice, riducendo di fatto la capacità di navigare tra gli elementi aggiunti solo da uno specifico sviluppatore. Esistono diverse strategie per affrontare questo problema:

1. Per i progetti nuovi, ho trovato relativamente semplice impostare un controllo di analisi statica che viene eseguito quando il codice viene creato e non consente il passaggio della compilazione automatica a meno che tutti gli elementi TODO non vengano rimossi. Questo aiuta a garantire che ogni volta che una modifica viene inviata a un sistema di controllo della versione, venga rimossa dagli elementi TODO non risolti.
2. In alcuni altri casi, è possibile utilizzare una strategia che consiste nel rimuovere tutti gli elementi TODO da un progetto prima di iniziare a lavorarci. A volte può portare a conflitti tra le persone quando gli elementi TODO vengono utilizzati per qualcosa di diverso dall'elenco di attività di TDD e qualcuno per qualsiasi motivo desidera che rimangano nel codice più a lungo. Anche se sono dell'opinione che tali casi di abbandono degli elementi TODO per un periodo più lungo dovrebbero essere estremamente rari nella migliore delle ipotesi, altri potrebbero avere opinioni diverse.
3. La maggior parte degli IDE moderni offrono indicatori di supporto diversi da `//TODO` per posizionare elementi in un elenco TODO, ad esempio, `//BUG`. In tal caso, posso utilizzare il marcatore `//BUG` er contrassegnare solo i miei elementi e quindi posso filtrare altri elementi in base a quel marcatore. I marcatori di bug comunemente non sono destinati a essere lasciati nel codice, quindi è molto meno rischioso che si accumulino.
4. Come tecnica di ultima istanza, di solito posso definire marcatori personalizzati che vengono inseriti nell'elenco TODO e, ancora una volta, utilizzare i filtri per vedere solo gli elementi che sono stati definiti da me (più spesso dei `NotImplementedException`).

14.3.4 Il processo TDD è stato ampliato con una lista di TODO

In uno dei capitoli precedenti, ho presentato il processo TDD di base che conteneva tre passaggi: scrivere lo Statement falso che si desidera sia vera, modificare il codice di produzione in modo che lo Statement sia vero e poi effettuare il refactoring del codice. La lista di TODO aggiunge nuovi passaggi a questo processo che portano al seguente elenco espanso:

1. Esaminare l'elenco di TODO e sceglierne uno che ha più senso da implementare in seguito.
2. Scrivere uno Statement falso che si vorrebbe fosse vero.
3. Vederlo segnalato come falso per il giusto motivo.
4. Modificare il codice di produzione per rendere vero lo Statement e assicurarsi che tutti gli Statement già veri rimangano tali.
5. Calcellare l'elemento dalla lista di TODO.
6. Ripetere i passaggi 1-5 finché non rimane più alcun elemento nella lista di TODO.

Naturalmente, possiamo (e dovremmo) aggiungere nuovi elementi alla lista TODO man mano che facciamo progressi con quelli esistenti e all'inizio di ogni ciclo l'elenco dovrebbe essere rivalutato per scegliere l'elemento più importante da implementare successivamente, prendendo anche in considerazione quanto aggiunto nel ciclo precedente.

14.3.5 Potenziali problemi con le liste TODO

Ci sono anche alcuni problemi riscontrabili utilizzando le liste TODO. Ho già menzionato il più importante - ho visto spesso persone aggiungere elementi TODO per scopi diversi dal supporto del TDD, senza mai tornare a questi elementi. Alcune persone scherzano dicendo che un commento TODO lasciato nel codice significa "C'è stato un tempo in cui volevo fare...". Ad ogni modo, tali elementi potrebbero inquinare la nostra lista TODO relativa a TDD con così tanta confusione che sono gli elementi risultano difficili da trovare.

Un altro svantaggio è che quando si lavora con più "aree di lavoro"/soluzioni, l'IDE raccoglierà elementi TODO solo da una singola soluzione/"area di lavoro", quindi potrebbero esserci momenti in cui sarà necessario mantenere diverse liste TODO, una per "area di lavoro" o soluzione. Fortunatamente, questo di solito non è un grosso problema.

CAPITOLO 15

Qual è lo scopo di uno Statement a livello di unità in TDD?

Nei capitoli precedenti ho descritto come i test formino una sorta di Specifica eseguibile composta da molti Statement. Se è così, allora è necessario sollevare alcune domande fondamentali riguardanti questi Statement, ad esempio:

1. Cosa c'è in un singolo Statement?
2. Come faccio a sapere che devo scrivere un altro Statement invece di espandere quello esistente?
3. Quando vedo uno Statement, come faccio a sapere se è troppo grande, troppo piccolo o appena sufficiente?

Questo può essere riassunto in una domanda più generale: quale dovrebbe essere lo "scope" di un singolo Statement?

15.1 Scope e livello

Il software che scriviamo può essere visualizzato in termini di struttura e funzionalità. La funzionalità riguarda le caratteristiche -- cose che un software fa e non fa, date determinate circostanze. La struttura è il modo in cui questa funzionalità è organizzata e divisa tra molti sotto-elementi, ad es. sottosistemi, servizi, componenti, classi, metodi, ecc.

Un elemento strutturale può facilmente gestire diverse funzionalità (da solo o in collaborazione con altri elementi). Ad esempio, molte liste implementano il recupero degli elementi aggiunti nonché qualche tipo di ricerca o ordinamento. D'altra parte, una singola funzionalità può facilmente estendersi a diversi elementi strutturali (ad esempio, il pagamento di un prodotto in un negozio online probabilmente si estenderà ad almeno diverse classi e probabilmente toccherà un database).

Pertanto, quando decidiamo cosa dovrebbe contenere un singolo Statement, dobbiamo considerare sia la struttura che la funzionalità e prendere le seguenti decisioni:

- struttura -- specifichiamo cosa dovrebbe fare una classe, o cosa dovrebbe fare l'intero componente, o forse uno Statement dovrebbe riguardare l'intero sistema? Mi riferirò a tale decisione strutturale come "livello".
- funzionalità -- un singolo Statement dovrebbe specificare tutto ciò che fa l'elemento strutturale, o forse solo una parte di esso? Se solo una parte, quale parte e quanto dovrebbe essere grande? Mi riferirò a tale decisione funzionale come "*scope* funzionale".

Le nostre domande dallinizio del capitolo possono essere riformulate come:

1. A quale livello specifichiamo il nostro software?
2. Quale dovrebbe essere lo scope funzionale di un singolo Statement?

15.2 A quale livello specifichiamo il nostro software?

La risposta alla prima domanda è relativamente semplice -- specifichiamo su più livelli. Quanti livelli ci sono e quali ci interessano dipende molto dal tipo specifico di applicazione che scriviamo e dal paradigma di programmazione (ad esempio nella programmazione funzionale pura, non abbiamo classi).

In questo capitolo (e in quello successivo), mi concentrerò principalmente sul livello di classe (lo chiamerò livello di unità, poiché una classe è un'unità di comportamento), ovvero ogni Statement è scritto rispetto a un'API pubblica di una specifica classe[1].

Ciò significa che possiamo utilizzare solo una singola classe nel nostro Statement eseguibile? Diamo un'occhiata ad un esempio di uno Statement ben scritto e proviamo a rispondere a questa domanda:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN
  var exceptionThrown = Assert.Throws<CustomException>(
    () => validation.ApplyTo(string.Empty)
  );

  //THEN
  Assert.True(exceptionThrown.IsFatalError);
}
```

Ok, allora vediamo... quante classi reali partecipano a questo Statement? Tre: una stringa, un'eccezione e la validazione. Quindi, anche se si tratta di una dichiarazione scritta per l'API pubblica della classe `Validation`, l'API stessa richiede l'utilizzo di oggetti di classi aggiuntive.

15.3 Quale dovrebbe essere lo scope funzionale di un singolo Statement?

La risposta breve a questa domanda è *comportamento*. Mettendolo insieme alla sezione precedente, possiamo dire che ogni Statement a livello di unità specifica un singolo comportamento di una classe scritto rispetto all'API pubblica di quella classe. Mi piace come Liz Keogh affermi che uno Statement a livello di unità mostra un esempio di come una classe sia preziosa per i suoi utilizzatori. Inoltre, Amir Kolsky e Scott Bain affermano che ogni Statement dovrebbe "introdurre unaădistinzione comportamentale non esistente prima".

Cosè esattamente un comportamento? Se questo libro è stato letto dall'inizio, probabilmente saranno state viste molte affermazioni che specificano i comportamenti. Lasciate che ve ne mostri un altro, però.

Consideriamo un esempio di una classe che rappresenta una condizione per decidere se un tipo di coda è piena o meno. Un singolo comportamento che possiamo specificare è che la condizione è soddisfatta quando viene avvisato tre volte di qualcosa aggiunto su una coda (quindi da un punto di vista più ampio, è un osservatore (observer) della coda):

```
[Fact] public void
ShouldBeMetWhenNotifiedThreeTimesOfItemQueued()
{
  //GIVEN
  var condition = new FullQueueCondition();
  condition.NotifyItemQueued();
  condition.NotifyItemQueued();
  condition.NotifyItemQueued();
```

(continues on next page)

[1] Alcuni, tuttavia, non sono d'accordo con la scrittura di Statement a livello di classe - vedere https://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html o https://vimeo.com/68375232

(continua dalla pagina precedente)

```
  //WHEN
  var isMet = condition.IsMet();

  //THEN
  Assert.True(isMet);
}
```

La prima cosa da notare è che vengono chiamati due metodi sull'oggetto `condition`: `NotifyItemQueued()` (tre volte) e `IsMet()` (una volta). Considero questo esempio educativo perché ho visto persone fraintendere il livello unitario (unit level) come "specificare un singolo metodo". Certo, di solito c'è un singolo metodo che attiva il comportamento (in questo caso è `isMet()`, inserito nella sezione `//WHEN`), ma a volte sono necessarie più chiamate per impostare le precondizioni per un determinato comportamento (da qui le tre chiamate `Queued()` inserite nella sezione `//GIVEN`).

La seconda cosa da notare è che lo Statement dice solo cosa succede quando l'oggetto `condition` viene notificato tre volte -- questo è il comportamento specificato. Che dire dello scenario in cui la `condition` viene notificata solo due volte e, quando richiesto in seguito, dovrebbe dire che non è soddisfatta? Questo è un comportamento separato e dovrebbe essere descritto da uno Statement separato. Lideale a cui tendiamo è caratterizzato da tre regole formulate da Amir Kolsky e citate da Ken Pugh nel suo libro *Lean-Agile Acceptance Test-Driven Development*:

1. Uno Statement dovrebbe diventare falso per un motivo ben definito.
2. Nessun altro Statement dovrebbe risultare falso per lo stesso motivo.
3. Uno Statement non dovrebbe risultare falso per nessun altro motivo.

Sebbene sia impossibile ottenerlo in senso letterale (ad esempio, tutti gli Statement che specificano i comportamenti `FullQueueCondition` devono chiamare un costruttore, quindi quando inserisco un `throw new Exception()`, tutti gli Statement diventeranno falsi), however, tuttavia, vogliamo mantenerci il più vicino possibile a questo obiettivo. In questo modo, ogni Statement introdurrà quella "distinzione comportamentale" di cui ho parlato prima, ovvero mostrerà un nuovo modo in cui la classe può essere preziosa per i suoi utilizzatori.

La maggior parte delle volte specifico i comportamenti utilizzando la struttura mentale "GIVEN-WHEN-THEN". Un comportamento viene attivato (`WHEN`) in un certo tipo di contesto (`GIVEN`) e c'è sempre qualche tipo di risultato (`THEN`) di quel comportamento.

15.4 Non rispettare le tre regole

Le tre regole che ho citato derivano dallesperienza. Vediamo cosa succede se non ne seguiamo una di esse.

Il nostro prossimo esempio riguarda una sorta di regola sulla dimensione del buffer. A questa regola viene chiesto se il buffer può gestire una stringa della lunghezza specificata e risponde "yes" se questa stringa è lunga al massimo tre elementi. Chi ha scritto uno Statement per questa classe a deciso di violare le regole di cui abbiamo parlato e ha scritto qualcosa del genere:

```
[Fact] public void
ShouldReportItCanHandleStringWithLengthOf3ButNotOf4AndNotNullString()
{
  //GIVEN
  var bufferSizeRule = new BufferSizeRule();

  //WHEN
  var resultForLengthOf3
    = bufferSizeRule.CanHandle(Any.StringOfLength(3));
  //THEN
  Assert.True(resultForLengthOf3);

  //WHEN - again?
  var resultForLengthOf4
```

(continues on next page)

(continua dalla pagina precedente)

```
    = bufferSizeRule.CanHandle(Any.StringOfLength(4))
  //THEN - again?
  Assert.False(resultForLengthOf4);

  //WHEN - again??
  var resultForNull = bufferSizeRule.CanHandle(null);
  //THEN - again??
  Assert.False(resultForNull);
}
```

Notare che specifica tre comportamenti:

1. Accettazione di una stringa di dimensione consentita.
2. Rifiuto di gestire una stringa di dimensione superiore al limite consentito.
3. Un caso speciale di stringa nulla.

In quanto tale, lo Statement infrange le regole: 1 (Uno Statement dovrebbe diventare falso per un motivo ben definito) e 3 (Uno Statement non dovrebbe diventare falso per nessun altro motivo). In effetti, ci sono tre ragioni che possono rendere falso il nostro Statement.

Ci sono diversi motivi per evitare di scrivere Statement come questo. Alcune sono:

1. La maggior parte dei framework xUnit interrompe l'esecuzione di uno Statement al primo errore di asserzione. Se la prima asserzione fallisce nello Statement precedente, non sapremo se il resto dei comportamenti funziona bene finché non risolviamo il primo.
2. La leggibilità tende a peggiorare così come il valore della documentazione delle nostre Specifiche (i nomi di tali Statements tendono ad essere tutt'altro che utili).
3. L'isolamento del fallimento è peggiore -- quando uno Statement diventa falso, preferiremmo sapere esattamente quale comportamento è stato sbagliato. Gli Statement come quello sopra non ci danno questo vantaggio.
4. Nel corso di un singolo Statement, di solito lavoriamo con lo stesso oggetto. Quando attiviamo più comportamenti su di esso, non possiamo essere sicuri di come l'attivazione di un comportamento influenzi i comportamenti successivi. Se abbiamo ad es. quattro comportamenti in un singolo Statement, non possiamo essere sicuri di come i tre precedenti influenzino l'ultimo. Nell'esempio sopra, potremmo farla franca, poiché l'oggetto specificato ha restituito il suo risultato basandosi solo sull'input di un metodo specifico (cioè non conteneva alcuno stato mutabile). Immaginate, tuttavia, cosa potrebbe accadere se innescassimo più comportamenti su un unica lista. Saremmo sicuri che non contenga alcun elemento rimasto dopo aver aggiunto alcuni elementi, cancellato alcuni, ordinato l'elenco e cancellato ancora di più?

15.5 Di quante asserzioni ho bisogno?

Una singola asserzione per definizione controlla una singola condizione specificata. Se un singolo Statement riguarda un singolo comportamento, che dire delle asserzioni? "Comportamento singolo" significa che posso avere solo una singola asserzione per Statement? Questo è stato soprattutto il caso degli Statement già visti in questo libro, ma non di tutti.

A dire il vero, c'è una risposta semplice a questa domanda -- una regola che dice: "avere una sola asserzione per test". Ciò che è importante ricordare è che si applica alle "asserzioni logiche", come ha indicato Robert C. Martin[2].

Prima di andare oltre, vorrei introdurre una distinzione. Una "asserzione fisica" è una singola chiamata `AssertXXXXX()`. Una "asserzione logica" è una o più asserzioni fisiche che insieme specificano una condizione logica. Per illustrare ulteriormente questa distinzione, vorrei darvi due esempi di asserzioni logiche.

[2] Le serie di "Clean Code", episodio 19 (https://cleancoders.com/episode/clean-code-episode-19-p1/show), Robert C. Martin, 2013

15.5.1 Asserzione logica -- esempio #1

Un buon esempio potrebbe essere un'asserzione che specifica che tutti gli elementi in una lista sono unici (ovvero l'elenco non contiene duplicati). XUnit.net non ha tale asserzione per default, ma possiamo immaginare di aver scritto qualcosa del genere e di averlo chiamato `AssertHasUniqueItems()`. Ecco del codice che utilizza questa asserzione:

```
//some hypothetical code for getting the list:
var list = GetList();

//invoking the assertion:
AssertHasUniqueItems(list);
```

Notare che si tratta di una singola asserzione logica, che specifica una condizione ben definita. Se diamo un'occhiata all'implementazione, tuttavia, troveremo il seguente codice:

```
public static void AssertHasUniqueItems<T>(List<T> list)
{
  for(var i = 0 ; i < list.Count ; i++)
  {
    for(var j = 0 ; j < list.Count ; j++)
    {
      if(i != j)
      {
        Assert.NotEqual(list[i], list[j]);
      }
    }
  }
}
```

Che esegue già diverse asserzioni fisiche. Se conoscessimo il numero esatto di elementi nella collection, potremmo anche utilizzare tre asserzioni `Assert.NotEqual()` invece di `AssertHasUniqueItems()`:

```
//some hypothetical code for getting the collection:
var list = GetLastThreeElements();

//invoking the assertions:
Assert.NotEqual(list[0], list[1]);
Assert.NotEqual(list[0], list[2]);
Assert.NotEqual(list[1], list[2]);
```

È ancora una sola asserzione? Fisicamente no, ma logicamente -- sì. C'è ancora una cosa logica che queste asserzioni specificano e cioè l'unicità degli elementi nella lista.

15.5.2 Asserzione logica -- esempio #2

Un altro esempio di asserzione logica è quella che specifica le eccezioni: `Assert.Throws()`. Ne abbiamo già incontrato uno simile in questo capitolo. Ecco di nuovo il codice:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN
  var exceptionThrown = Assert.Throws<CustomException>(
    () => validation.ApplyTo(string.Empty)
  );
```

(continues on next page)

(continua dalla pagina precedente)

```
  //THEN
  Assert.True(exceptionThrown.IsFatalError);
}
```

Notare che in questo caso ci sono due asserzioni fisiche (`Assert.Throws()` e `Assert.True()`), ma un solo intento -- specificare l'eccezione che deve essere sollevata. Possiamo anche estrarre queste due asserzioni fisiche in una sola con un nome significativo:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN - THEN
  AssertFatalErrorIsThrownWhen(
    () => validation.ApplyTo(string.Empty)
  );
}
```

Quindi ogni volta che abbiamo diverse asserzioni fisiche che possono essere (o sono) estratte in un unico metodo di asserzione con un nome significativo, le considero un'unica asserzione logica. C'è sempre un'area grigia in quello che può essere considerato un "nome significativo" (ma concordiamo sul fatto che `AssertAllConditionsAreMet()` non è un nome significativo, ok?). La regola generale è che questo nome dovrebbe esprimere il nostro intento in modo migliore e più chiaro rispetto al mucchio di asserzioni che nasconde. Guardando nuovamente l'esempio di `AssertHasUniqueItems()` questa asserzione esprime meglio il nostro intento rispetto a un insieme di tre `Assert.NotEqual()`.

15.6 Riepilogo

In questo capitolo abbiamo cercato di scoprire quanto dovrebbe contenere un singolo Statement. Abbiamo esaminato le nozioni di livello e *scope* funzionale per arrivare alla conclusione che uno Statement a dovrebbe coprire un singolo comportamento. Abbiamo supportato questa affermazione con le tre regole di Amir Kolsky e abbiamo esaminato un esempio di cosa potrebbe accadere quando non ne seguiamo una. Infine, abbiamo discusso come la nozione di "singolo Statement per comportamento" sia correlata a "singola asserzione per Statement".

CAPITOLO 16

Sviluppare uno stile TDD e Non-Determinismo Vincolato

In *uno dei primi capitoli*, hoăpresentato l'idea di un generatore di valori anonimi. Ho mostrato la classe `Any` che utilizzo per generare tali valori. Nei capitoli successivi l'ho utilizzata ampiamente in molte degli Statement che ho scritto.

È giunto il momento di spiegare qualcosa in più sui principi alla base dell'utilizzo di valori anonimi negli Statement. Lungo il percorso esamineremo anche lo sviluppo di uno stile di TDD.

16.1 Uno stile?

Sì. Perché sto sprecando tempo a scrivere di stile invece di fornire i dettagli tecnici fondamentali? Ecco la mia risposta: prima di iniziare a scrivere questo tutorial, ho letto quattro o cinque libri esclusivamente sul TDD e forse altri due che contenevano capitoli sul TDD. Il tutto ammontava a circa due o tremila pagine cartacee, oltre a numerosi post su numerosi blog. E sapete cosa ho notato? Non esistono due autori che utilizzino lo stesso insieme di tecniche per testare il proprio codice! Voglio dire, a volte, quando si esaminano le tecniche suggerite, due autorità si contraddicono a vicenda. Dato che ogni autore ha i propri seguaci, non è raro osservare e prendere parte a discussioni su se questa o quella tecnica sia migliore di unaăconcorrente o quale tecnica sia "evanescente"[1] e porta a problemi a lungo termine.

Ne ho fatte molte anch'io. Hoăanche cercato di capire come mai le persone lodano le tecniche che ioă(pensavo di) SAPERE fossero sbagliate e portassero al disastro. Col tempo, ho capito che non si tratta di un dibattito "tecnica A contro tecnica B". Esistono alcuni insiemi di tecniche che lavorano insieme e si migliorano a vicenda in simbiosi. La scelta di una tecnica ci lascia con problemi che dobbiamo risolvere adottando altre tecniche. Ecco come si sviluppa uno stile.

Lo sviluppo di uno stile inizia con una serie di problemi da risolvere e una serie di principi sottostanti che consideriamo importanti. Questi principi ci portano ad adottare la nostra prima tecnica, che poi ce ne fa adottare un'altra e, alla fine, emerge uno stile coerente. Usando il "Constrained Non-Determinism" come esempio, cercherò di mostrare come parte di uno stile deriva da una tecnica derivata da un principio.

16.2 Principio: I Test Come Specifiche

Come ho già sottolineato, credo fermamente che i test dovrebbero costituire una specifica eseguibile. Pertanto, non dovrebbero solo passare valori di input a un oggetto e asserirli sull'output, ma dovrebbero anche trasmettere al lettore le regole in base alle quali lavorano oggetti e funzioni. Il seguente esempio banale mostra uno ăStatement in cui non viene spiegato esplicitamente quale sia la relazione tra input e output:

[1] Uno di questi articoli può essere trovato su https://martinfowler.com/articles/mocksArentStubs.html

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo("MY_HOSTNAME");

  //THEN
  Assert.Equal("backup_MY_HOSTNAME.zip", name);
}
```

Anche se in questo caso la relazione può essere dedotta abbastanza facilmente, non è ancora esplicitamente dichiarata, quindi in scenari più complessi potrebbe non essere così banale da individuare. Inoltre, vedere un codice del genere mi fa porre domande come:

- Il prefisso `"backup_"` viene sempre applicato? Cosa succede se passo il prefisso stesso invece di `"MY_HOSTNAME"`? Il nome sarà `"backup_backup_.zip"` o solo `"backup_.zip"`?
- Questo oggetto è responsabile della validazione della stringa passata? Se passo `"MY HOST NAME"` (con gli spazi) verrà generata un'eccezione o verrà semplicemente applicato il pattern di formattazione come al solito?
- Ultimo ma non meno importante, che dire delle lettere maiuscole? Perché `"MY_HOSTNAME"` è scritto in maiuscolo? Se passo `"my_HOSTNAME"`, verrà rifiutato o accettato? O forse verrà automaticamente convertito in maiuscolo?

Questo mi porta ad adottare la prima tecnica per fornire ai miei Statement un supporto migliore per il principio che seguo.

16.3 Prima tecnica: Input Anonimo

Posso racchiudere il valore effettivo `"MY_HOSTNAME"` con un metodo e assegnargli un nome che documenti meglio i vincoli imposti dalla funzionalità specificata. In questo caso, il metodo `BackupFileNamePattern()` dovrebbe accettare qualunque stringa gli venga fornita (l'oggetto non è responsabile della validazione dell'input), quindi chiamerò il metodo wrapper `AnyString()`:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var hostname = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname);

  //THEN
  Assert.Equal("backup_MY_HOSTNAME.zip", name);
}

public string AnyString()
{
  return "MY_HOSTNAME";
}
```

Utilizzando **input anonimo**, ho fornito una migliore documentazione del valore di input. Qui ho scritto `AnyString()`, ma ovviamente può esserci una situazione in cui utilizzo dati più vincolati, ad es.iInventerei un metodo chiamato `AnyAlphaNumericString()` se avessi bisogno di una stringa che non contenga caratteri diversi da lettere e cifre.

{{keyInfo}} **Input anonimo e classi di equivalenza**. Notare che questa tecnica è utile solo quando specifichiamo uno scenario che dovrebbe verificarsi per tutti i membri di qualche tipo di classe di equivalenza. Un esempio di classe di

equivalenza è "una stringa che inizia con un numero" o "un numero intero positivo" o "qualsiasi URI legale". Quando un comportamento dovrebbe verificarsi solo per un singolo valore di input specifico, non c'è spazio per renderlo anonimo. Prendendo come esempio l'autorizzazione, quando un determinato comportamento si verifica solo quando il valore di input è `Users.Admin`, non abbiamo una classe di equivalenza utile e dovremmo utilizzare semplicemente il valore letterale di `Users.Admin`. D'altra parte, per uno scenario che si verifica per tutti i valori diversi da `Users.Admin`, ha senso utilizzare un metodo come `AnyUserOtherThan(Users.Admin)` o anche `AnyNonAdminUser()`, perché si tratta di una classe di equivalenza utile.

Ora che lo Statement stesso è libero dalla conoscenza del valore concreto della variabile `hostname`, il valore concreto di `"backup_MY_HOSTNAME.zip"` nell'asserzione sembra un po' strano. Questo perché non c'è ancora un'indicazione chiara del tipo di relazione tra input e output e se esiste (come è attualmente, lo Statement suggerisce che il risultato di `ApplyTo()` sia lo stesso per qualsiasi valore di `hostname`). Questo ci porta ad un'altra tecnica.

16.4 Seconda tecnica: Valori Derivati

Per documentare meglio la relazione tra input e output, dobbiamo semplicemente derivare il valore atteso che asseriamo dal valore di input. Ecco lo stesso Statement con l'asserzione modificata::

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var hostname = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname);

  //THEN
  Assert.Equal($"backup_{hostname}.zip", name);
}
public string AnyString()
{
  return "MY_HOSTNAME";
}
```

Sembra più una parte di una specifica, perché stiamo documentando il formato del nome del file di backup e mostriamo quale parte del formato è variabile e quale parte è fissa. Questo è qualcosa che probabilmente si troverà documentato in una specifica cartacea per l'applicazione che si sta scrivendo -- probabilmente conterrà una frase che dice: "Il formato di un file di backup dovrebbe essere **backup_H.zip**, dove **H** è il nome dell'host locale corrente". Quello che abbiamo usato qui era un **valore derivato**.

I **Valori derivati** riguardano la definizione dell'output previsto in termini di input che è stato passato per fornire un'indicazione chiara sul tipo di "transformazione" che il codice di produzione deve eseguire sul suo input.

16.5 Terza tecnica: Valori Generati Distinti

Supponiamo che qualche tempo dopo la pubblicazione della nostra versione iniziale, ci venga chiesto di modificare la funzionalità di backup in modo che memorizzi i dati di backup separatamente per ciascun utente che richiama tale funzionalità. Poiché il cliente non desidera che si verifichino conflitti di nome tra file creati da utenti diversi, ci viene chiesto di aggiungere il nome dell'utente che esegue il backup al nome del file di backup. Pertanto, il nuovo formato è **backup_H_U.zip**, dove H è ancora il nome host e U è l'username. Anche il nostro Statement per il pattern deve cambiare per includere queste informazioni. Naturalmente, stiamo provando a utilizzare nuovamente l'input anonimo come una tecnica collaudata e alla fine otteniamo:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
```

(continues on next page)

(continua dalla pagina precedente)

```
{
  //GIVEN
  var hostname = AnyString();
  var userName = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal($"backup_{hostname}_{userName}.zip", name);
}

public string AnyString()
{
  return "MY_HOSTNAME";
}
```

Ora possiamo vedere che cè qualcosa di sbagliato in questo Statement. `AnyString()` viene utilizzato due volte e ogni volta restituisce lo stesso valore, il che significa che la valutazione dello Statement non ci dà alcuna garanzia, che vengano utilizzati entrambi gli argomenti del metodo `ApplyTo()` e che siano utilizzati nel posti corretti. Per esempio, lo Statement sarà considerato vero quando il valore del nome utente viene utilizzato al posto del nome host dal metodo `ApplyTo()`. Ciò significa che se vogliamo comunque utilizzare l'input anonimo in modo efficace senza incorrere in falsi positivi[2], dobbiamo rendere distinti i due valori, ad es. come questo:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
{
  //GIVEN
  var hostname = AnyString1();
  var userName = AnyString2(); //different value
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal($"backup_{hostname}_{userName}.zip", name);
}

public string AnyString1()
{
  return "MY_HOSTNAME";
}

public string AnyString2()
{
  return "MY_USER_NAME";
}
```

Abbiamo risolto il problema (per ora) introducendo un altro metodo helper. Tuttavia, come si può vedere, questa non è una soluzione molto scalabile. Pertanto, proviamo a ridurre a uno il numero di metodi helper per la generazione di stringhe e a fare in modo che restituiscano ogni volta un valore diverso:

[2] Un "falso positivo" è un test che dovrebbe fallire ma che passa.

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
{
  //GIVEN
  var hostname = AnyString();
  var userName = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal($"backup_{hostname}_{userName}.zip", name);
}

public string AnyString()
{
  return Guid.NewGuid().ToString();
}
```

Questa volta, il metodo `AnyString()` restituisce un GUID invece di un testo leggibile. La generazione di un nuovo GUID ogni volta offre una garanzia abbastanza forte che ogni valore sia distinto. La stringa non leggibile dall'uomo (contrariamente a qualcosa come `"MY_HOSTNAME"`) potrebbe far temere che forse stiamo perdendo qualcosa, ma ehi, non abbiamo detto **Any**String()?

Valori generati distinti significa che ogni volta che generiamo un valore anonimo, è diverso (se possibile) da quello precedente e ciascuno di questi valori viene generato automaticamente utilizzando un qualche tipo di euristica.

16.6 Quarta tecnica: Specifica Costante

Consideriamo un'altra modifica che ci viene richiesto di apportare: questa volta il nome del file di backup deve contenere anche il numero di versione della nostra applicazione. Ricordando che vogliamo utilizzare la tecnica dei valori derivati, non inseriremo il numero di versione nel nostro Statement. Utilizzeremo invece una costante già definita altrove nel codice di produzione dell'applicazione (in questo modo evitiamo anche la duplicazione di questo numero di versione nell'applicazione). Immaginiamo che questo numero di versione sia memorizzato come una costante chiamata `Number` nella classe `Version`. Lo Statement aggiornato per i nuovi requisiti si presenta così:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserNameAndVersion()
{
  //GIVEN
  var hostname = AnyString();
  var userName = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal(
    $"backup_{hostname}_{userName}_{Version.Number}.zip", name);
}

public string AnyString()
{
  return Guid.NewGuid().ToString();
}
```

Ancora una volta, invece di un valore letterale come `5.0`, ho utilizzato la costante `Version.Number` che contiene il valore. Ciò mi ha permesso di utilizzare un valore derivato nell'asserzione, ma mi ha lasciato un po' preoccupato sul fatto che `Version.Number` stesso sia corretto -- dopo tutto, ho utilizzato la costante del codice di produzione per la creazione del valore atteso. Se modificassi accidentalmente questa costante nel mio codice con un valore non valido, lo Statement verrebbe comunque considerato vero, anche se il comportamento stesso sarebbe sbagliato.

Per accontentare tutti, di solito risolvo questo dilemma scrivendo unăsingolo Statement solo per la costante per specificare quale dovrebbe essere il valore:

```
public class VersionSpecification
{
 [Fact] public void
 ShouldContainNumberEqualTo1_0()
 {
   Assert.Equal("1.0", Version.Number);
 }
}
```

In questo modo, mi sono assicurato che esista uno Statement che diventerà falso ogni volta che cambio accidentalmente il valore di `Version.Number`. In questo modo non devo preoccuparmene nel resto delle specifiche. Finché vale questo Statement, gli altri possono utilizzare la costante del codice di produzione senza preoccupazioni.

16.7 Riepilogo dell'esempio

Mostrando questo esempio, ho cercato di dimostrare come uno stile può evolversi dai principi in cui crediamo e dai vincoli che incontriamo quando applichiamo tali principi. L'ho fatto per due motivi:

1. Per presentare una serie di tecniche (anche se sarebbe più accurato usare la parola "pattern") che utilizzo e consiglio. Fare un esempio è stato il modo migliore che mi venisse in mente per descriverli in modo fluido e logico.
2. Per aiutare a comunicare meglio con persone che utilizzano stili diversi. Invece di limitarsi a dire loro "stai sbagliando", considerare la possibilità di comprendere i loro principi e il modo in cui le loro tecniche supportano tali principi.

Ora, facciamo un breve riepilogo di tutte le tecniche introdotte nell'esempio del nome del file di backup:

Valori Derivati : Definisco l'output atteso in termini di input per documentare la relazione tra input e output.

Input Anonimo : Quando voglio documentare il fatto che un particolare valore non è rilevante per la dichiarazione corrente, utilizzo un metodo speciale che mi produce il valore. Chiamo questo metodo in base alla classe di equivalenza a cui ho bisogno che appartenga (ad esempio `Any.AlphaNumericString()`) e in questo modo rendo il mio Statement agnostico rispetto al particolare valore utilizzato.

Valori Generati Distinti : Quando utilizzo l'input anonimo, genero ogni volta un valore distinto (in caso di tipi che hanno pochissimi valori, come booleano, provo almeno a non generare lo stesso valore due volte di seguito) per rendere lo Statement più affidabile.

Specifica Costante : Scrivo uno Statement separato per una costante per specificare quale dovrebbe essere il suo valore. In questo modo, posso utilizzare la costante invece del suo valore letterale in tutti gli altri Statement per creare un Valore Derivato senza il rischio che la modifica del valore della costante non venga rilevata dalla mia Specifica eseguibile.

16.8 Non.determinismo vincolato

Quando combiniamo input anonimi con valori generati distinti, otteniamo qualcosa che viene chiamato **Constrained Non-Determinism** [Non-determinismo vincolato]. Questo è un termine coniato da Mark Seemann e significa tre cose:

1. I valori sono anonimi, ovvero non conosciamo il valore effettivo che stiamo utilizzando.
2. I valori vengono generati nella sequenza più distinta possibile (il che significa che, quando possibile, non esistono due valori generati uno dopo l'altro che contengano lo stesso valore)

3. Il non determinismo nella generazione dei valori è vincolato, il che significa che gli algoritmi per la generazione dei valori sono scelti attentamente per fornire valori che appartengono a una specifica classe di equivalenza e che non sono "diabolici" (ad esempio quando generiamo "any integer", preferiremmo non generare lo 0' poiché di solito è un valore speciale che spesso merita uno Statement separato).

Esistono diversi modi per implementare il non-determinismo vincolato. Lo stesso Mark Seemann ha inventato la libreria AutoFixture per C# che è disponibile gratuitamente per il download. Ecco lo snippet più breve possibile per generare un numero intero anonimo utilizzando AutoFixture:

```
Fixture fixture = new Fixture();
var anonymousInteger = fixture.Create<int>();
```

Io, d'altra parte, seguo Amir Kolsky e Scott Bain, che consigliano di utilizzare la classe `Any` come visto nei capitoli precedenti di questo libro. `Any` adotta un approccio leggermente diverso rispetto ad AutoFixture (sebbene utilizzi AutoFixture internamente). La mia implementazione della classe `Any` è disponibile anche per il download.

16.9 Riepilogo

Spero che questo capitolo abbia dato una certa comprensione di come sono nati i diversi stili TDD e del motivo per cui utilizzo alcune delle tecniche che utilizzo (e di come queste tecniche non siano solo una serie di scelte casuali). Nei prossimi capitoli, cercherò di introdurre alcune altre tecniche per aiutarti a sviluppare un insieme di trucchi accurati -- uno stile coerente[3].

[3] Per la più grande raccolta di tali tecniche, o più precisamente, pattern, vedere "XUnit Test Patterns" di Gerard Meszaros.

CAPITOLO 17

Specificare i confini e le condizioni funzionali

{{keyInfo}} **A Disclaimer**. Prima di iniziare, devo dichiarare che questo capitolo si basa su una serie di post di Scott Bain e Amir Kolsky dal blog Sustainable Test-Driven Development e sul loro prossimo libro con lo stesso titolo. Mi piace così tanto il modo in cui adattano l'idea del boundary testing che ho imparato a seguire le loro linee guida. Questo capitolo sarà unaăriformulazione di queste linee guida. Invito a leggere i post originali del blog su questo argomento su http://www.sustainabletdd.com/ (e ad acquistare il prossimo libro di Scott, Amir e Max Guernsey).

17.1 A volte un valore anonimo non è sufficiente

Nell'ultimo capitolo ho descritto come i valori anonimi siano utili quando specifichiamo un comportamento che dovrebbe essere lo stesso indipendentemente dagli argomenti passati al costruttore o dai metodi invocati. Un esempio potrebbe essere quello di inserire un numero intero in uno stack e riprenderlo (pop) per vedere se è lo stesso elemento che abbiamo inserito -- il comportamento è coerente per qualunque numero inseriamo (push) e preleviamo (pop):

```
[Fact] public void
ShouldPopLastPushedItem()
{
  //GIVEN
  var lastPushedItem = Any.Integer();
  var stack = new Stack<int>();
  stack.Push(Any.Integer());
  stack.Push(Any.Integer());
  stack.Push(lastPushedItem);

  //WHEN
  var poppedItem = stack.Pop();

  //THEN
  Assert.Equal(lastPushedItem, poppedItem);
}
```

In questo caso, i valori dei primi due numeri interi inseriti nello stack non hanno importanza: la relazione descritta tra input e output è indipendente dai valori effettivi che utilizziamo. Come abbiamo visto nel capitolo precedente, questo è il caso tipico in cui applichiamo il "Constrained Non-Determinism".

A volte, tuttavia, gli oggetti specificati mostrano comportamenti diversi in base a ciò che viene passato ai rispettivi costruttori o metodi o a ciò che ottengono chiamando altri oggetti. Per esempio:

- nella nostra applicazione, potremmo avere una politica di licenza in base alla quale è consentito l'utilizzo di una funzionalità solo quando la licenza è valida e negata dopo la sua scadenza. In tal caso, il comportamento prima della data di scadenza è diverso da quello successivo -- la data di scadenza è il limite del comportamento funzionale.
- Alcuni negozi sono aperti dalle 10:00 alle 18:00, quindi se nella nostra applicazione avessimo una domanda se il negozio è attualmente aperto, ci aspetteremmo che ricevesse una risposta diversa in base all'ora corrente. Anche qui, le date di apertura e di chiusura rappresentano i limiti del comportamento funzionale.
- Un algoritmo che calcola il valore assoluto di un numero intero restituisce lo stesso numero per input maggiori o uguali a `0` ma nega l'input per numeri negativi. Pertanto, `0` segna il confine funzionale in questo caso.

In questi casi, dobbiamo scegliere attentamente i nostri valori di input per ottenere un livello di confidenza sufficiente evitando di specificare eccessivamente i comportamenti con troppi Statement (che di solito introducono penalità sia nel tempo di esecuzione delle Specifiche che nella manutenzione). Scott e Amir si basano sulle pratiche comprovate della comunità di testing[1] e ci danno alcuni consigli su come scegliere i valori. Descriverò queste linee guida (leggermente modificate in più punti) in tre parti:

1. specificando eccezioni alle regole -- dove il comportamento è lo stesso per ogni valore di input tranne uno o più valori specificati esplicitamente,
2. specificando i confini
3. specificando gli intervalli -- dove ci sono più confini.

17.2 Eccezioni alla regola

Ci sono momenti in cui uno ăStatement è vero per ogni valore tranne uno (o più) esplicitamente specificato. Il mio approccio varia leggermente a seconda dell'insieme di valori possibili e del numero di eccezioni. Fornirò tre esempi per aiutare a capire meglio queste variazioni.

17.2.1 Esempio 1: una singola eccezione da un ampio insieme di valori

In alcuni paesi alcune cifre vengono evitate, ad es. nei numeri dei piani di alcuni ospedali e alberghi a causa di alcune superstizioni locali o semplicemente perché hanno un suono simile a un'altra parola che ha un significato molto negativo. Un esempio di ciò è la /tetrafobia/[2], che porta a saltare la cifra `4`, poiché in alcune lingue suona simile alla parola "morte". In altre parole, qualsiasi numero contenente `4` viene evitato e quando si entra nell'edificio si potrebbe non trovare un quarto (o quattordicesimo) piano. Immaginiamo di avere diverse regole simili per i nostri hotel in diverse parti del mondo e vogliamo che il software ci dica se una determinata cifra è consentita dalle superstizioni locali. Una di queste regole deve essere implementata da una classe chiamata `Tetraphobia`:

```
public class Tetraphobia : LocalSuperstition
{
  public bool Allows(char number)
  {
    throw new NotImplementedException("not implemented yet");
  }
}
```

Implementa l'interfaccia `LocalSuperstition` che ha un metodo `Allows()`, quindi per motivi di correttezza della compilazione, abbiamo dovuto creare la classe e il metodo. Ora che lo abbiamo, vogliamo testare l'implementazione. Quali Statement scriviamo?

Ovviamente abbiamo bisogno di una dichiarazione che dica cosa succede quando passiamo una cifra non consentita:

```
[Fact] public void
ShouldReject4()
{
  //GIVEN
```

(continues on next page)

[1] vedere ad es. il capitolo 4.3 del "Foundation Level Syllabus" di ISQTB all'indirizzo https://istqb.ita-stqb.org/docs/ITASTQB-FLSY-V4.pdf
[2] https://it.wikipedia.org/wiki/Tetrafobia

(continua dalla pagina precedente)

```
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isFourAccepted = tetraphobia.Allows('4');

  //THEN
  Assert.False(isFourAccepted);
}
```

Notare che utilizziamo il valore specifico per il quale si verifica il comportamento eccezionale. Tuttavia, potrebbe essere un'ottima idea estrarre 4 in una costante. In uno dei capitoli precedenti, hoădescritto unaătecnica chiamata **Constant Specification** [*Specifica Costante*], in cui scriviamo una dichiarazione esplicita sul valore della costante denominata e utilizziamo la costante denominata stessa ovunque invece del suo valore letterale. Allora perché non ho usato questa tecnica questa volta? L'unico motivo è che questo potrebbe sembrare un po' sciocco con un esempio così banale. In realtà, avrei dovuto utilizzare il nome della costante. Facciamo adesso questo esercizio e vediamo cosa succede.

```
[Fact] public void
ShouldRejectSuperstitiousValue()
{
  //GIVEN
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isSuperstitiousValueAccepted =
    tetraphobia.Allows(Tetraphobia.SuperstitiousValue);

  //THEN
  Assert.False(isSuperstitiousValueAccepted);
}
```

Quando lo facciamo, dobbiamo documentare la costante denominata con la seguente istruzione:

```
[Fact] public void
ShouldReturn4AsSuperstitiousValue()
{
  Assert.Equal('4', Tetraphobia.SuperstitiousValue);
}
```

È ora di uno Statement che descriva il comportamento per tutti i valori non-eccezionali. Questa volta utilizzeremo un metodo della classe `Any` chiamata `Any.OtherThan()`, che genera qualsiasi valore diverso da quello specificato (e produce codice gradevole e leggibile come effetto collaterale):

```
[Fact] public void
ShouldAcceptNonSuperstitiousValue()
{
  //GIVEN
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isNonSuperstitiousValueAccepted =
    tetraphobia.Allows(Any.OtherThan(Tetraphobia.SuperstitiousValue);

  //THEN
  Assert.True(isNonSuperstitiousValueAccepted);
}
```

e questo è tutto -- di solito non scrivo più dichiarazioni in questi casi. I possibili valori di input sono così tanti che non

sarebbe razionale specificarli tutti. Prendendo spunto dal famoso commento di Kent Beck da Stack Overflow[3], penso che il nostro compito non sia scrivere quanti più Statement possiamo, ma il meno possibile documentare veramente il sistema e darci il livello di fiducia desiderato.

17.2.2 Esempio 2: una singola eccezione da un piccolo insieme di valori

La situazione è diversa, tuttavia, quando il valore eccezionale viene scelto da un insieme piccolo -- questo è spesso il caso in cui il tipo di valore di input è un'enumerazione. Torniamo a un esempio tratto da uno dei nostri capitoli precedenti, in cui abbiamo specificato che esiste una sorta di funzionalità di reporting a cui è possibile accedere sia da un ruolo di amministratore che da un ruolo di revisore (auditor). Modifichiamo questo esempio per ora e diciamo che solo gli amministratori possono accedere alla funzione di reporting:

```
[Fact] public void
ShouldGrantAdministratorsAccessToReporting()
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(Roles.Admin);

 //THEN
 Assert.True(accessGranted);
}
```

L'approccio a questa parte non è diverso da quello che ho fatto nel primo esempio -- ho scritto una dichiarazione per il singolo valore eccezionale. È ora di pensare all'altro Statement -- quello che specifica cosa dovrebbe accadere per il resto dei ruoli. Vorrei descrivere due modi in cui questo compito può essere affrontato.

Il primo modo è procedere come nell'esempio precedente -- scegliere un valore diverso da quello eccezionale. Questa volta utilizzeremo il metodo `Any.OtherThan()`, adatto a questo caso:

```
[Fact] public void
ShouldDenyAnyRoleOtherThanAdministratorAccessToReporting()
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(Any.OtherThan(Roles.Admin));

 //THEN
 Assert.False(accessGranted);
}
```

Questo approccio presenta due vantaggi:

1. Viene eseguito un solo Statement per il caso di "access denied", quindi non vi è alcuna penalità significativa in termini di tempo di esecuzione.
2. Nel caso in cui espandessimo la nostra enumerazione in futuro, non dovremo modificare questo Statement -- il membro dell'enumerazione aggiunto avrà la possibilità di essere selezionato.

Tuttavia, c'è anche uno svantaggio -- non possiamo essere sicuri che il membro enum appena aggiunto venga utilizzato in questo Statement. Nell'esempio precedente, non ci importava molto dei valori utilizzati, perché:

- L'intervallo `char` era piuttosto ampio, quindi specificare i comportamenti per tutti i valori potrebbe rivelarsi problematico e inefficiente dato il livello di confidenza desiderato,

[3] https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565

- `char` è un insieme fisso di valori -- non possiamo espandere `char` come espandiamo le enumerazioni, quindi non è necessario preoccuparsi del futuro.

E se ci fossero solo altri due ruoli tranne `Roles.Admin`, p. es. `Auditor` e `CasualUser`? In questi casi, a volte scrivo uno Statement che viene eseguito rispetto a tutti i valori non-eccezionali, utilizzando l'attributo `[Theory]` di xUnit.net che mi consente di eseguire lo stesso codice dello Statement con diversi set di argomenti. Un esempio qui sarebbe:

```
[Theory]
[InlineData(Roles.Auditor)]
[InlineData(Roles.CasualUser)]
public void
ShouldDenyAnyRoleOtherThanAdministratorAccessToReporting(Roles role)
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(role);

 //THEN
 Assert.False(accessGranted);
}
```

Lo Statement di cui sopra viene eseguito sia per `Roles.Auditor` che per `Roles.CasualUser`. Lo svantaggio di questo approccio è che ogni volta che espandiamo un'enumerazione, dobbiamo tornare qui e aggiornare lo Statement. Poiché tendo a dimenticare queste cose, cerco di mantenere al massimo uno Statement nel sistema a seconda dell'enumerazione -- se trovo più di un punto in cui vario il mio comportamento in base ai valori di una particolare enumerazione, cambio il design per sostituire enum col polimorfismo. Gli Statements nel TDD possono essere utilizzati come strumento per rilevare problemi di progettazione e fornirò una discussione più approfondita su questo argomento in un prossimo capitolo.

17.2.3 Esempio 3: Più di un'eccezione

I due esempi precedenti presuppongono che vi sia una sola eccezione alla regola. Tuttavia, questo concetto può essere esteso a più valori, purché si tratti di un insieme finito e discreto. Se più valori eccezionali producono lo stesso comportamento, di solito provo a coprirli tutti utilizzando la funzionalità `[Theory]` menzionata di xUnit.net. Lo dimostrerò prendendo l'esempio precedente di concessione dell'accesso e presupponendo che questa volta sia gli amministratori che i revisori siano autorizzati a utilizzare la funzionalità. Uno Statement per il comportamento assomiglierebbe a questo:

```
[Theory]
[InlineData(Roles.Admin)]
[InlineData(Roles.Auditor)]
public void
ShouldAllowAccessToReportingBy(Roles role)
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(role);

 //THEN
 Assert.False(accessGranted);
}
```

Nell'ultimo esempio, ho utilizzato questo approccio per i valori non-eccezionali, dicendo che questo è ciò che a volte faccio. Tuttavia, quando si specificano più eccezioni alla regola, questo è il mio approccio di default -- la natura dei valori eccezionali è che sono rigorosamente specificati, quindi voglio che siano tutti inclusi nella mia Specifica.

Questa volta non mostrerò lo Statement per valori non-eccezionali poiché segue l'approccio delineato nell'esempio precedente.

17.3 Regole valide entro i confini

A volte, unăcomportamento varia attorno a unăconfine. Un semplice esempio potrebbe essere una regola su come determinare se qualcuno è adulto o meno. Di solito si è considerati adulti dopo aver raggiunto una certa età, diciamo, 18 anni. Fingendo di operare sulla base degli anni (senza tenere conto dei mesi), la regola è:

1. Quando l'età in anni è inferiore a 18 anni non è considerato adulto.
2. Quando l'età in anni è almeno 18, si è considerati adulti.

Come si può vedere, esiste un confine tra i due comportamenti. Il "bordo destro" di questo confine è 18. Perché dico "bordo destro"? Questo perché il confine ha sempre due bordi, il che, tra l'altro, significa anche che ha una lunghezza. Se assumiamo che stiamo parlando della regola dell'età adulta menzionata e che il nostro dominio numerico è quello dei numeri interi, possiamo anche usare 17 invece di 18 come valore del bordo e dire che:

1. Quando l'età in anni è al massimo di 17 anni, non è considerato un adulto.
2. Quando l'età in anni è superiore a 17 anni, si è considerati adulti.

Quindi un confine non è un singolo numero -- ma ha sempre una lunghezza -- ovvero la lunghezza tra l'ultimo valore del comportamento precedente e il primo valore del comportamento successivo. Nel caso del nostro esempio, la lunghezza tra 17 (bordo sinistro -- ultima età non-adulta) e 18 (bordo destro -- primo valore adulto) è 1.

Ora, si immagini di non parlare più di valori interi, ma di trattare il tempo per quello che è -- un continuum. Quindi il valore del bordo destro sarebbe ancora di 18 anni. Ma per quanto riguarda il bordo sinistro? Non sarebbe possibile mantenerlo per 17 anni, poiché la regola si applicherebbe ad es. Anche 17 anni e 1 giorno. Quindi qual è il valore corretto del bordo destro e la lunghezza corretta del confine? Il bordo sinistro dovrebbe essere di 17 anni e 11 mesi? Oppure 17 anni, 11 mesi, 365/366 giorni (qui abbiamo la questione dell'anno bisestile)? O forse 17 anni, 11 mesi, 365/366 giorni, 23 ore, 59 minuti, ecc.? È più difficile rispondere a questa domanda e dipende fortemente dal contesto -- è necessario rispondere per ogni caso particolare in base al dominio e alle esigenze aziendali -- in questo modo sappiamo che tipo di precisione ci si aspetta da noi.

Nelle nostre Specifiche, dobbiamo documentare in qualche modo la lunghezza del confine. Ciò porta una domanda interessante: come descrivere la lunghezza del confine con gli Statement? Per illustrare ciò, voglio mostrare due Statement che descrivono il citato calcolo dell'età adulta espresso utilizzando il granulo degli anni (quindi tralasciamo mesi, giorni, ecc.).

Il primo Statement è per valori inferiori a 18 e vogliamo specificarlo per il valore del bordo sinistro (cioè 17), ma calcolato rispetto al valore del bordo destro (cioè invece di scrivere 17, scriviamo 18-1):

```
[Fact] public void
ShouldNotBeSuccessfulForAgeLessThan18()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var notAnAdult = 18 - 1; //more on this later

  //WHEN
  var isSuccessful = detection.PerformFor(notAnAdult);

  //THEN
  Assert.False(isSuccessful);
}
```

E il prossimo Statement per valori maggiori o uguali a 18 e vogliamo utilizzare il valore del bordo destro:

```
[Fact] public void
ShouldBeSuccessfulForAgeGreaterThanOrEqualTo18()
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  //GIVEN
  var detection = new AdultAgeDetection();
  var adult = 18;

  //WHEN
  var isSuccessful = detection.PerformFor(adult);

  //THEN
  Assert.True(isSuccessful);
}
```

Ci sono due cose da notare su questi esempi. Il primo è che non ho utilizzato alcun tipo di metodo `Any`. Utilizzo `Any` nei casi in cui non ho un limite o quando non considero alcun valore di una classe di equivalenza migliore di altri in un modo particolare. Quando specifico i limiti, tuttavia, invece di utilizzare metodi come `Any.IntegerGreaterOrEqualTo(18)`, utilizzo i valori dei bordi poiché trovo che definiscono più rigorosamente il confine e guidano la giusta implementazione. Inoltre, specificare esplicitamente i comportamenti per i valori dei bordi mi consente di documentare la lunghezza del confine.

La seconda cosa da notare è l'uso della costante letterale `18` nell'esempio sopra. In uno dei capitoli precedenti, ho descritto unaătecnica chiamata **Constant Specification** che consiste nello scrivere uno Statement esplicito sul valore della costante denominata e utilizzare il nome della costante ovunque invece del suo valore letterale. Allora perché non ho usato questa tecnica questa volta?

L'unico motivo è che questo potrebbe sembrare un po' sciocco con un esempio estremamente banale come il rilevamento dell'età adulta. In realtà, avrei dovuto utilizzare il nome della costante in entrambe le dichiarazioni e avrebbe mostrato la lunghezza del confine in modo ancora più chiaro. Eseguiamo ora questo esercizio e vediamo cosa succede.

Per prima cosa documentiamo il nome della costante con la seguente dichiarazione:

```
[Fact] public void
ShouldIncludeMinimumAdultAgeEqualTo18()
{
  Assert.Equal(18, Age.MinimumAdult);
}
```

Ora abbiamo tutto ciò di cui abbiamo bisogno per riscrivere i due Statement scritti in precedenza. Il primo sarebbe simile a questo:

```
[Fact] public void
ShouldNotBeSuccessfulForLessThanMinimumAdultAge()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var notAnAdultYet = Age.MinimumAdult - 1;

  //WHEN
  var isSuccessful = detection.PerformFor(notAnAdultYet);

  //THEN
  Assert.False(isSuccessful);
}
```

E il prossimo Statement per valori maggiori o uguali a 18 sarebbe simile a questo:

```
[Fact] public void
ShouldBeSuccessfulForAgeGreaterThanOrEqualToMinimumAdultAge()
{
  //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
  var detection = new AdultAgeDetection();
  var adultAge = Age.MinimumAdult;

  //WHEN
  var isSuccessful = detection.PerformFor(adultAge);

  //THEN
  Assert.True(isSuccessful);
}
```

Come si vede, il primo Statement contiene la seguente espressione:

```
Age.MinimumAdult - 1
```

dove 1 è la lunghezza esatta del confine. Come ho detto prima, lesempio è così banale che può sembrare sciocco e divertente, tuttavia, in scenari reali, questa è una tecnica che applico sempre e ovunque.

Potrebbe sembrare che i confini si applichino solo all'input numerico, ma si trovano in molti altri posti. Ci sono limiti associati a data/ora (ad esempio, il calcolo dell'età adulta sarebbe questo tipo di caso se non ci fermassimo a contare gli anni ma considerassimo invece il tempo come un continuum -- bisognerebbe decidere se abbiamo bisogno di precisione in secondi o magari in tick) o stringhe (es. validazione del nome utente dove deve essere di almeno 2 caratteri, oppure password che deve contenere almeno 2 caratteri speciali). Si applicano anche alle espressioni regolari. Ad esempio, per una semplice regex \d+, dovremmo sicuramente specificare almeno tre valori: una stringa vuota, una singola cifra e una singola non cifra.

17.4 Combinazione di confini -- intervalli

Gli esempi precedenti si concentravano su un unico confine. Che dire allora di unaăsituazione in cui ce ne sono più, ovvero un comportamento è valido entro unăintervallo?

17.4.1 Esempio -- patente di guida

Consideriamo il seguente esempio: viviamo in un paese in cui un cittadino può ottenere la patente di guida solo dopo i 18 anni, ma prima dei 65 (il governo ha deciso che le persone dopo i 65 anni potrebbero avere una vista peggiore e che è più sicuro non dare loro una nuova patente). Supponiamo di voler sviluppare una classe che risponda alla domanda se possiamo richiedere la patente di guida e i valori restituiti da questa query sono i seguenti:

1. Età < 18 -- restituisce il valore enum `QueryResults.TooYoung`
2. 18 <= età <= 65 -- restituisce il valore enum `QueryResults.AllowedToApply`
3. Età > 65 -- restituisce il valore enum `QueryResults.TooOld`

Ora, ci si ricorda che ho scritto che specifico i comportamenti con i limiti utilizzando i valori dei bordi? Questo approccio, se applicato alla situazione appena descritta, mi darebbe i seguenti Statement:

1. Età = 17, dovrebbe restituire il risultato `QueryResults.TooYoung`
2. Età = 18, dovrebbe restituire il risultato `QueryResults.AllowedToApply`
3. Età = 65, dovrebbe restituire il risultato `QueryResults.AllowedToApply`
4. Età = 66, dovrebbe restituire il risultato `QueryResults.TooOld`

pertanto, descriverei il comportamento in cui la query dovrebbe restituire due volte il valore `AllowedToApply`. Questo non è un grosso problema se mi aiuta a documentare i confini.

Il primo Statement dice cosa dovrebbe succedere fino ai 17 anni:

```
[Fact]
public void ShouldRespondThatAgeLessThan18IsTooYoung()
```

(continues on next page)

(continua dalla pagina precedente)

```
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(18-1);

  //THEN
  Assert.Equal(QueryResults.TooYoung, result);
}
```

Il secondo Statement ci dice che la fascia 18 -- 65 è quella in cui un cittadino può richiedere la patente di guida. Lo scrivo come una teoria (usando nuovamente l'attributo `[InlineData()]` di xUnit.net) perché questo intervallo ha due limiti attorno ai quali cambia il comportamento:

```
[Theory]
[InlineData(18, QueryResults.AllowedToApply)]
[InlineData(65, QueryResults.AllowedToApply)]
public void ShouldRespondThatDrivingLicenseCanBeAppliedForInRangeOf18To65(
  int age, QueryResults expectedResult
)
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(age);

  //THEN
  Assert.Equal(expectedResult, result);
}
```

L'ultimo Statement specifica quale dovrebbe essere la risposta quando qualcuno ha più di 65 anni:

```
[Fact]
public void ShouldRespondThatAgeMoreThan65IsTooOld()
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(65+1);

  //THEN
  Assert.Equal(QueryResults.TooOld, result);
}
```

Notare che ho usato 18-1 e 65+1 invece di 17 e 66 per mostrare che 18 e 65 sono i valori limite e che la lunghezza dei confini è, in entrambi i casi, 1. Naturalmente, avrei dovuto usare costanti al posto di 18 e 65 (forse qualcosa come `MinimumApplicantAge` e `MaximumApplicantAge`) -- lo lascerò come esercizio al lettore.

17.4.2 Esempio -- impostare un allarme

Nell'esempio precedente siamo stati abbastanza fortunati perché la logica specificata era puramente funzionale (cioè restituiva risultati diversi in base a input diversi). Grazie a ciò, durante la stesura della teoria per la fascia di età 18-65 anni, abbiamo potuto parametrizzare i valori di input insieme ai risultati attesi. Non è sempre così. Ad esempio, immaginiamo di avere una classe `Clock` che ci consente di programmare un allarme. La classe ci consente di impostare l'ora in modo sicuro tra 0 e 24, altrimenti genera un'eccezione.

Questa volta devo scrivere due Statement parametrizzati -- uno in cui viene restituito un valore (per i casi validi) e l'altro in cui viene generata l'eccezione (per i casi non validi). Il primo sarebbe simile a questo:

```
[Theory]
[InlineData(Hours.Min)]
[InlineData(Hours.Max)]
public void
ShouldBeAbleToSetHourBetweenMinAndMax(int inputHour)
{
  //GIVEN
  var clock = new Clock();
  clock.SetAlarmHour(inputHour);

  //WHEN
  var setHour = clock.GetAlarmHour();

  //THEN
  Assert.Equal(inputHour, setHour);
}
```

e il secondo:

```
[Theory]
[InlineData(Hours.Min-1)]
[InlineData(Hours.Max+1)]
public void
ShouldThrowOutOfRangeExceptionWhenTryingToSetAlarmHourOutsideValidRange(
  int inputHour)
{
  //GIVEN
  var clock = new Clock();

  //WHEN - THEN
  Assert.Throws<OutOfRangeException>(
    ()=> clock.SetAlarmHour(inputHour)
  );
}
```

A parte questo, ho usato lo stesso approccio dell'ultima volta.

17.5 Riepilogo

In questo capitolo ho descritto come specificare i limiti funzionali con una quantità minima di codice e Statement, in modo che la Specifica sia più mantenibile e venga eseguita più velocemente. Rimane ancora un altro tipo di situazione: quando abbiamo condizioni composte (ad esempio, una password deve contenere almeno 10 caratteri e contenere almeno 2 caratteri speciali) -- torneremo a quelle quando introdurremo gli oggetti mock.

CAPITOLO 18

Guidare l'implementazione dalle Specifiche

Come uno degli ultimi argomenti delle tecniche TDD fondamentali che non richiedono di approfondire il mondo della progettazione orientata agli oggetti, vorrei mostrare tre tecniche per trasformare in vero uno Statement falso. I nomi delle tecniche provengono da un libro di Kent Beck, Test-Driven Development: By Example e sono:

1. Digitare l'implementazione ovvia
2. Una versione fasulla (finché si può fare)
3. Triangolare

Non ci si preoccupi se questi nomi non dicono nulla, le tecniche non sono così difficili da comprendere e cercherò di fare un esempio di ognuna di esse.

18.1 Digitare l'implementazione ovvia

La prima tecnica dice semplicemente: quando si conosce l'implementazione corretta e finale per rendere vero uno Statement, la si scrive. Se l'implementazione è ovvia, questo approccio ha molto senso - dopo tutto, il numero di Statement necessari per specificare (e testare) una funzionalità dovrebbe riflettere il livello di sicurezza desiderato. Se questo livello è molto alto basterà digitare il codice corretto in risposta ad un singolo Statement. Vediamolo in azione con un banale esempio di somma di due numeri:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(3,5);

  //THEN
  Assert.Equal(8, sum);
}
```

Ci si ricorderà che in uno dei capitoli precedenti ho scritto che di solito dovremmo scrivere il codice di produzione più semplice che renda vero lo Statement. L'approccio menzionato ci incoraggerebbe a restituire semplicemente 8 dal metodo `Of()` perché sarebbe sufficiente per rendere vero lo Statement. Invece di farlo, tuttavia, potremmo decidere che la logica è così ovvia che possiamo semplicemente digitarla nella sua forma finale:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

e basta. Notare che non ho utilizzato il "Constrained Non-Determinism " nello Statement, perché il suo utilizzo impone in un certo senso l'utilizzo dell'approccio "digita l'implementazione ovvia". Questo è anche uno dei motivi per cui molti Statement che ho scritto finora nei capitoli precedenti sono stati implementati digitando l'implementazione corretta. Giusto per illustrarlo, diamo un'occhiata a come apparirebbe la dichiarazione di cui sopra se utilizzassi il "Constrained Non-Determinism":

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var a = Any.Integer();
  var b = Any.Integer();
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(a,b);

  //THEN
  Assert.Equal(a + b, sum);
}
```

L'implementazione più ovvia che renderebbe vero questo Statement è l'implementazione corretta -- non riesco a farla franca restituendo un valore costante come potrei quando non utilizzo il "Constrained Non-Determinism". Questo perché questa volta semplicemente non so quale sia il risultato atteso poiché dipende strettamente dai valori di input che nemmeno conosco.

18.2 Una versione fasulla (finché si può fare)

La seconda tecnica mi ha fatto sorridere quando ne sono venuto a conoscenza. Non ricordo di averla mai utilizzata nel codice di produzione reale, eppure la trovo così interessante che voglio mostrarla comunque. È così semplice che non si rimpiangeranno questi pochi minuti anche solo per allargare i propri orizzonti.

Supponiamo di avere già scritto uno Statement falso e di volerlo rendere vero scrivendo il codice di produzione. In questo momento, applichiamo *Una versione fasulla (finché si può fare)* in due passi:

1. Iniziamo con un passaggio di "versione fasulla". In questo caso, trasformiamo in vero uno Statement falso utilizzando l'implementazione più ovvia possibile, anche se non è quella corretta (da cui il nome del passaggio: "versione fasulla" l'implementazione reale per "imbrogliare" lo Statement). Di solito, all'inizio è sufficiente restituire una costante letterale.
2. Quindi procediamo con la fase "Una versione fasulla" - facciamo affidamento sul nostro senso di duplicazione tra lo Statement e la implementazione (fake) per trasformare gradualmente entrambe nelle loro forme più generali che eliminano questa duplicazione. Di solito, otteniamo questo risultato trasformando le costanti in variabili, le variabili in parametri, ecc.

Un esempio potrebbe essere utile proprio adesso, quindi applichiamo *versione fasulla...* allo stesso esempio di addizione della sezione *Digitare l'implementazione ovvia*. Lo Statement è lo stesso di prima:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  //GIVEN
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(3, 5);

  //THEN
  Assert.Equal(8, sum);
}
```

Per l'implementazione, però, utilizzeremo il codice più ovvio che trasformerà lo Statement a vero. Come accennato, questa implementazione più ovvia restituisce quasi sempre unaăcostante:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 8; //we faked the real implementation
  }
}
```

Lo Statement ora diventa vero (verde), anche se l'implementazione è ovviamente sbagliata. Ora è il momento di eliminare le duplicazioni tra lo Statement e il codice di produzione.

Innanzitutto, notiamo che il numero 8 è duplicato tra lo Statement e l'implementazione -- l'implementazione lo restituisce e lo Statement lo asserisce. Per ridurre questa duplicazione, suddividiamo l'8 nell'implementazione in un'addizione:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 3 + 5;
  }
}
```

Notare il trucco intelligente che hoăfatto. Hoăcambiato la duplicazione tra implementazione e *risultato atteso* dello Statement in duplicazione tra implementazione e i *valori di input* dello Statement. Ho cambiato il codice di produzione da utilizzare

```
return 3 + 5;
```

esattamente perché lo Statement utilizzava questi due valori in questo modo:

```
var sum = addition.Of(3, 5);
```

Questo tipo di duplicazione è diverso dal precedente in quanto può essere rimosso utilizzando parametri (questo si applica non solo ai parametri di input di un metodo ma a tutto ciò a cui abbiamo accesso prima di attivare il comportamento specificato -- parametri del costruttore, campi, ecc. in contrasto con il risultato che normalmente non conosciamo finché non invochiamo il comportamento). La duplicazione del numero 3 può essere eliminata modificando il codice di produzione in modo che utilizzi il valore passato dallo Statement. Così questo:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 3 + 5;
  }
}
```

Si trasforma in questo:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + 5;
  }
}
```

In questo modo abbiamo eliminato la duplicazione del numero 3 - abbiamo utilizzato un parametro del metodo per trasferire il valore di 3 dallo Statement all'implementazione di `Of()`, quindi ora lo abbiamo in un unico posto. Dopo questa trasformazione, ci resta duplicato solo il numero 5, quindi trasformiamolo nello stesso modo in cui abbiamo trasformato il 3:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

E questo è tutto - siamo arrivati all'implementazione corretta. Ho usato un esempio banale, dato che non voglio spendere troppo tempo su questo, ma se ne possono trovare di più avanzati nel libro di Kent Beck, volendo.

18.3 Triangolare

La triangolazione è considerata la tecnica più conservativa della terna descritta, perché seguirla richiede i più piccoli passaggi possibili per arrivare alla giusta soluzione. Il termine *Triangolazione* all'inizio sembra misterioso - almeno lo era per me, soprattutto perché non mi faceva venire in mente nulla che avesse a che fare con l'ingegneria del software. Il nome è stato preso dalla triangolazione radar dove gli output di almeno due radar devono essere utilizzati per determinare la posizione di un'unità. Inoltre, nella triangolazione radar, la posizione viene misurata indirettamente, combinando i seguenti dati: distanza (non posizione!) tra due radar, misura effettuata da ciascun radar e posizione dei radar (che conosciamo, perché siamo noi a mettete lì i radar). Da questi dati possiamo ricavare un triangolo, quindi possiamo utilizzare la trigonometria per calcolare la posizione del terzo punto del triangolo, che è la posizione desiderata dell'unità (i due punti rimanenti sono le posizioni dei radar). Tale misura è di natura indiretta, poiché non misuriamo la posizione direttamente, ma la calcoliamo da altre misure di supporto.

Queste due caratteristiche: la misura indiretta e luso di almeno due fonti di informazione sono al centro della triangolazione TDD. Ecco come può essere tradotto dai radar al codice:

1. **Misura indiretta**: nel codice, significa che deriviamo l'implementazione interna e la progettazione di un modulo da diversi esempi noti del suo comportamento desiderato visibile esternamente, osservando cosa varia in questi esempi e modificando il codice di produzione in modo che questa variabilità sia trattata in modo generico. Ad esempio, la variabilità potrebbe portarci a cambiare una costante in una variabile, perché diversi esempi utilizzano valori di input diversi.
2. **Utilizzo di almeno due fonti di informazione**: nel codice, significa che iniziamo con l'implementazione più semplice possibile del comportamento e la rendiamo più generale **solo** quando abbiamo due o più esempi diversi di questo comportamento (ad es. che descrivono la funzionalità desiderata per diversi input). Poi è possibile aggiungere nuovi esempi e ripetere la generalizzazione. Questo processo viene ripetuto fino a raggiungere l'implementazione desiderata. Robert C. Martin ha sviluppato unaămassima al riguardo, affermando che "Man mano che i test diventano più specifici, il codice diventa più generico".

Di solito, quando il TDD viene presentato in esempi semplici, la triangolazione è la tecnica principale utilizzata, quindi molti principianti credono erroneamente che il TDD sia solo una questione di triangolazione.

La ritengo una tecnica importante perché:

1. Molti praticanti del TDD la usano e la mostrano, quindi presumo che prima o poi la si vedrà e molto probabilmente ci saranno domande al riguardo.
2. Ci consente di arrivare alla giusta implementazione effettuando passi molto piccoli (i più piccoli di quelli visti finora in questo libro) e lo trovo molto utile quando sono incerto su come dovrebbero essere l'implementazione e la progettazione corrette.

18.3.1 Esempio 1 - addizionare numeri

Prima di mostrare un esempio più avanzato di triangolazione, vorrei tornare al nostro esempio esempio della somma di due numeri interi. Questo ci permetterà di vedere come la triangolazione differisce dalle altre due tecniche menzionate in precedenza.

Per scrivere gli esempi, utilizzeremo la funzionalità di xUnit.net per gli Statement parametrizzati, ovvero le teorie - questo ci consentirà di fornire molti esempi della funzionalità desiderata senza duplicare il codice.

Il primo esempio è simile al seguente:

```
[Theory]
[InlineData(0,0,0)]
public void ShouldAddTwoNumbersTogether(
  int addend1,
  int addend2,
  int expectedSum)
{
  //GIVEN
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(addend1, addend2);

  //THEN
  Assert.Equal(expectedSum, sum);
}
```

Notare che abbiamo parametrizzato non solo i valori di input ma anche il risultato atteso (`expectedSum`). Il primo esempio specifica che `0 + 0 = 0`.

L'implementazione, analogamente a *Una versione fasulla (finché si può fare)* per ora consiste nel restituire semplicemente una costante:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 0;
  }
}
```

Ora, contrariamente alla tecnica *Una versione fasulla...*, non proviamo a rimuovere la duplicazione tra lo Statement e il codice. Aggiungiamo invece un altro esempio della stessa regola. Cosa intendo con "la stessa regola"? Bene, dobbiamo considerare i nostri assi di variabilità. Nell'operazione di addizione, due cose possono variare - il primo addendo o il secondo - quindi abbiamo due assi di variabilità. Per il nostro secondo esempio, dobbiamo mantenerne uno invariato mentre cambiamo l'altro. Diciamo che decidiamo di mantenere il secondo valore di input uguale a quello dell'esempio precedente (che è 0) e di cambiare il primo valore in 1. Quindi questo singolo esempio:

```
[Theory]
[InlineData(0,0,0)]
```

Diventa un insieme di due esempi:

```
[Theory]
[InlineData(0,0,0)]
[InlineData(1,0,1)] //NEW!
```

Ancora una volta, notrea che il secondo valore di input rimane lo stesso in entrambi gli esempi e il primo varia. Anche il risultato atteso deve essere diverso.

Per quanto riguarda l'implementazione, proviamo ancora a rendere vero lo Statement utilizzando l'implementazione più stupida possibile:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 1) return 1;
    return 0;
  }
}
```

Abbiamo già due esempi, quindi se vediamo uno schema ripetitivo, potremmo provare a generalizzarlo. Supponiamo però di non avere ancora un'idea su come generalizzare l'implementazione, quindi aggiungiamo un terzo esempio:

```
[Theory]
[InlineData(0,0,0)]
[InlineData(1,0,1)]
[InlineData(2,0,2)]
```

E l'implementazione viene estesa a:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return 2;
    if(a == 1) return 1;
    return 0;
  }
}
```

Ora, osservando questo codice, possiamo notare un pattern - per ogni valore di input finora, restituiamo il valore del primo: per 1 restituiamo 1, per 2 restituiamo 2, per 0 restituiamo 0. Possiamo quindi generalizzare questa implementazione. Generalizziamo solo la parte relativa alla gestione del numero 2 per vedere se la direzione è giusta:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return a; //changed from 2 to a
    if(a == 1) return 1;
    return 0;
  }
}
```

Gli esempi dovrebbero essere ancora veri a questo punto, quindi non abbiamo violato il codice esistente. È ora di cambiare la seconda istruzione `if`:

```
public class Addition
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  public int Of(int a, int b)
  {
    if(a == 2) return a;
    if(a == 1) return a; //changed from 1 to a
    return 0;
  }
}
```

Abbiamo ancora la barra verde, quindi il passo successivo sarebbe generalizzare la parte `return 0` in `return a`:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return a;
    if(a == 1) return a;
    return a; //changed from 0 to a
  }
}
```

Gli esempi dovrebbero essere ancora veri. A proposito, la triangolazione non ci obbliga a fare piccoli passi come in questo caso, però volevo mostrare che lo rende possibile. La capacità di compiere piccoli passi quando necessario è qualcosa che apprezzo molto quando utilizzo il TDD. Ad ogni modo, possiamo notare che ciascuna condizione termina con lo stesso risultato, quindi non abbiamo affatto bisogno delle condizioni. Possiamo rimuoverle e lasciare solo:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a;
  }
}
```

Pertanto, abbiamo generalizzato il primo asse di variabilità, che è il primo addendo. È ora di variare il secondo, lasciando invariato il primo addendo. Ai seguenti esempi esistenti:

```
[Theory]
[InlineData(0,0,0)] //0+0=0
[InlineData(1,0,1)] //1+0=1
[InlineData(2,0,2)] //2+0=2
```

Aggiungiamo il seguente:

```
[InlineData(2,1,3)] //2+1=3
```

Notare che abbiamo già utilizzato il valore 2 per il primo addendo in uno degli esempi precedenti, quindi questa volta decidiamo di congelarlo e di variare il secondo addendo, che finora è sempre stato 0. L'implementazione sarebbe qualcosa del genere:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + 1;
    }
```

(continues on next page)

(continua dalla pagina precedente)

```
    else
    {
      return a;
    }
  }
}
```

Abbiamo già due esempi per la variazione del secondo addendo, quindi potremmo generalizzare. Diciamo, tuttavia, che non vediamo ancora il pattern. Aggiungiamo un altro esempio per un valore diverso del secondo addendo:

```
[Theory]
[InlineData(0,0,0)] //0+0=0
[InlineData(1,0,1)] //1+0=1
[InlineData(2,0,2)] //2+0=2
[InlineData(2,1,3)] //2+1=3
[InlineData(2,2,4)] //2+2=4
```

Quindi abbiamo aggiunto 2+2=4. Ancora una volta, limplementazione dovrebbe essere la più banale possibile:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + 1;
    }
    else if(b == 2)
    {
      return a + 2;
    }
    else
    {
      return a;
    }
  }
}
```

Ora possiamo vedere il pattern più chiaramente. Qualunque valore di `b` passiamo al metodo `Of()`, viene aggiunto ad `a`. Proviamo a generalizzare, questa volta utilizzando un passaggio un po' più grande:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + b; //changed from 1 to b
    }
    else if(b == 2)
    {
      return a + b; //changed from 2 to b
    }
    else
    {
      return a + b; //added "+ b"
    }
```

(continues on next page)

(continua dalla pagina precedente)

```
  }
}
```

Ancora una volta, questo passaggio è stato più ampio, poiché abbiamo modificato tre posizioni in un'unica modifica. Ricordare che la triangolazione ci permette di scegliere la dimensione del gradino, quindi questa volta ne ho scelto uno più grande perché mi sentivo più sicuro. Ad ogni modo, possiamo vedere che il risultato per ogni branch [*ramo*] è esattamente lo stesso: `a + b`, quindi possiamo rimuovere del tutto le condizioni e ottenere:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

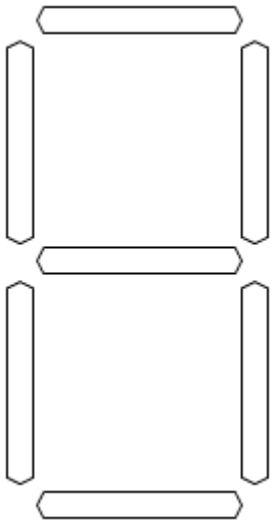
ed ecco fatto - abbiamo triangolato con successo la funzione di addizione. Ora, capisco che deve essere sembrato estremamente esagerato per ricavare un'aggiunta ovvia in questo modo. Da ricordare che ho fatto questo esercizio solo per mostrare i meccanismi, non per fornire un valido argomento a sostegno dell'utilità della triangolazione.

18.3.2 Esempio 2 - Display a LED

Non vi biasimo se il primo esempio non è servito a convincere che la triangolazione può essere utile. Dopotutto, si trattava di calcolare la somma di due numeri interi! Il prossimo esempio sarà qualcosa di meno ovvio. Vorrei avvisare, però, che mi prenderò del tempo per descrivere il problema e mostrerò solo una parte della soluzione, quindi se ne ha già abbastanza di triangolazioni, può saltare questo esempio e tornaci sopra più tardi.

Ora che abbiamo finito con il disclaimer, ecco la descrizione.

Immaginiamo di dover scrivere una classe che produca un display LED a 7 segmenti ASCII art. Nella vita reale, tali display vengono utilizzati per visualizzare le cifre:



Un esempio di arte ASCII prevista dalla nostra classe è simile a questa:

```
 _ 
| |
 _ 
| |
 _ 
```

Notare che esistono tre tipi di simboli:

- `.` significa spazio (non c'è alcun segmento lì) o un segmento non illuminato.
- `-` indica un segmento orizzontale illuminato
- `|` indica un segmento verticale illuminato

La funzionalità che dobbiamo implementare dovrebbe consentire non solo di visualizzare i numeri ma di illuminare qualsiasi combinazione di segmenti a piacimento. Possiamo quindi decidere di non illuminare nessun segmento, ottenendo così il seguente output:

```
...
...
...
...
...
```

Oppure per illuminare solo il segmento superiore, che porta al seguente output:

```
.-.
...
...
...
...
```

Come diciamo alla nostra classe di illuminare questo o quel segmento? Gli passiamo una stringa di nomi di segmenti. I segmenti si chiamano A, B, C, D, E, F, G e la mappatura di ciascun nome su un segmento specifico può essere visualizzata come:

```
.A.
F.B
.G.
E.C
.D.
```

Quindi, per ottenere l'output descritto in precedenza in cui è illuminato solo il segmento superiore, dobbiamo passare l'input costituito da `"A"`. Volendo illuminare tutti i segmenti, passiamo `"ABCDEFG"`. Se vogliamo mantenere tutti i segmenti disattivati, passiamo `""` (o un equivalente C#: `string.Empty`).

L'ultima cosa che devo dire prima di iniziare è che, per il bene di questo esercizio, ci concentreremo solo sull'input valido (ad esempio presupponiamo che non otterremo input come "AAAA", o "abc" o "ZXVN"). Naturalmente, nei progetti reali, dovrebbero essere specificati anche i casi di input non validi.

È il momento del primo Statement. Per cominciare, specificherò il caso di input vuoto che porta a tutti i segmenti spenti:

```
[Theory]
[InlineData("", new [] {
  "...",
  "...",
  "...",
  "...",
  "...",
})]
public void ShouldConvertInputToAsciiArtLedDisplay(
  string input, string[] expectedOutput
)
{
  //GIVEN
  var asciiArts = new LedAsciiArts();
```

(continues on next page)

(continua dalla pagina precedente)

```
  //WHEN
  var asciiArtString = asciiArts.ConvertToLedArt(input);

  //THEN
  Assert.Equal(expectedOutput, asciiArtString);
}
```

Ancora una volta, come ho descritto nell'esempio precedente, dal lato del codice di produzione, facciamo la cosa più semplice solo per rendere vero questo esempio. Nel nostro caso, questo sarebbe:

```
public string[] ConvertToLedArt(string input)
{
  return new [] {
    "...",
    "...",
    "...",
    "...",
    "...",
  };
}
```

L'esempio è ora implementato. Naturalmente questa non è l'implementazione finale dell'intera logica di conversione. Questo è il motivo per cui dobbiamo scegliere il prossimo esempio da specificare. Questa scelta determinerà quale asse di cambiamento perseguiremo per primo. Ho deciso di specificare il segmento più in alto (ovvero il segmento `A`) - abbiamo già un esempio che dice quando questo segmento è spento, ora ne abbiamo bisogno di uno che dica cosa dovrebbe succedere quando lo accendo. Riutilizzerò lo stesso corpo dello Statement e aggiungerò semplicemente un altro attributo `InlineData` per eseguire l'istruzione per il nuovo set di input e output previsto:

```
[InlineData("A", new [] {
  ".-.", // note the '-' character
  "...",
  "...",
  "...",
  "...",
})]
```

Questa volta, sto passando `"A"` come input e mi aspetto di ricevere quasi lo stesso output di prima, solo che questa volta la prima riga legge `".-."` invece di `"..."`.

Implemento questo esempio utilizzando, ancora una volta, il codice più ingenuo e più semplice da scrivere. Il risultato è:

```
public string[] ConvertToLedArt(string input)
{
  if(input == "A"))
  {
    return new [] {
      ".-.",
      "...",
      "...",
      "...",
      "...",
    };
  }
  else
  {
    return new [] {
      "...",
```

(continues on next page)

(continua dalla pagina precedente)

```
      "...",
      "...",
      "...",
      "...",
    };
  }
}
```

L'implementazione è piuttosto stupida, ma ora che abbiamo due esempi, possiamo individuare un pattern. Si noti che, a seconda della stringa di input, possono essere restituiti due possibili risultati. Tutte le righe sono uguali tranne la prima riga, che, finora, è l'unica che dipende dal valore di `input`. Pertanto, possiamo generalizzare il codice di produzione estraendo la duplicazione in qualcosa del genere:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      (input == "A") ? ".-." : "...",
      "...",
      "...",
      "...",
      "...",
    };
}
```

Notare che ho modificato il codice in modo che solo la prima riga dipenda da `input`. Ma non è finita qui. Osservando la condizione per la prima riga:

```
(input == "A") ? ".-." : "..."
```

possiamo inoltre notare che è solo il carattere centrale che cambia a seconda di ciò che passiamo. Sia il carattere più a sinistra che quello più a destra della prima riga sono sempre `.`. Quindi, generalizziamo ulteriormente, per finire con qualcosa del genere:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + ((input == "A") ? "-" : ".") + ".",
      "...",
      "...",
      "...",
      "...",
    };
}
```

Ora, se guardiamo più da vicino l'espressione:

```
((input == "A") ? "-" : ".")
```

Possiamo notare che la sua responsabilità è determinare se il valore del segmento corrente è basato su `input`. Possiamo usare questa conoscenza per estrarla in un metodo con un nome rivelatore-dell'intento. Il corpo del metodo è:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input == turnOnToken) ? turnOnValue : ".");
}
```

Dopo questa estrazione, il nostro metodo `ConvertToLedArt` diventa:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + DetermineSegmentValue(input, "A", "-") + ".",
      "...",
      "...",
      "...",
      "...",
    };
}
```

E abbiamo finito di triangolare il segmento `A`.

Ulteriori conclusioni dall'esempio del display a LED

Il fatto che io abbia finito di triangolare lungo un asse di variabilità non significa che non posso eseguire la triangolazione lungo altri assi. Ad esempio, quando esaminiamo nuovamente il codice del metodo `DetermineSegmentValue()`:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input == turnOnToken) ? turnOnValue : ".");
}
```

Possiamo vedere chiaramente che il metodo sta rilevando un token eseguendo un confronto diretto tra stringhe: `input == turnOnToken`. Ciò fallirà, ad es. se passo `"AB"`, probabilmente dovremo triangolare lungo questo asse per arrivare all'implementazione corretta. Non mostrerò i passaggi qui, ma il risultato finale di questa triangolazione sarebbe qualcosa del tipo:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input.Contains(turnOnToken) ? turnOnValue : ".");
}
```

E dopo averlo fatto, il metodo `DetermineSegmentValue` sarà qualcosa che potremo utilizzare per implementare l'illuminazione di altri segmenti, senza bisogno di scoprirlo di nuovo utilizzando la triangolazione per ogni segmento. Quindi, presupponendo che questo metodo sia nella sua forma finale, quando scriverò un esempio per il segmento B, lo renderò vero utilizzando il metodo `DetermineSegmentValue()` fin dall'inizio invece di inserire prima un `if` e poi generalizzare. L'implementazione sarà simile alla seguente:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + DetermineSegmentValue(input, "A", "-") + ".",
      ".." + DetermineSegmentValue(input, "B", "|"),
      "...",
      "...",
      "...",
    };
}
```

Notare che questa volta ho utilizzato l'approccio *Digitare l'implementazione ovvia* - questo perché, a causa della precedente triangolazione, questo passaggio è *diventato* ovvio.

Le due lezioni che ne derivano sono:

1. Quando smetto di triangolare lungo un asse, potrei ancora aver bisogno di triangolare lungo gli altri.
2. La triangolazione mi consente di fare passi più piccoli quando *necessario* e quando non lo faccio, utilizzo un altro approccio. Ci sono molte cose che non triangolo.

Spero che, mostrando questo esempio, ho presentato un caso più convincente a favore della triangolazione. Vorrei fermarmi qui, lasciando il resto dell'esercizio al lettore.

18.4 Riepilogo

In questo lungo capitolo, ho provato a dimostrare tre tecniche per passare da uno Statement falso a uno vero:

1. Digitare l'implementazione ovvia
2. Una versione fasulla (finché si può fare)
3. Triangolare

Spero che questa sia stata un'introduzione facile da digerire e per saperne di più, si può leggere il libro di Kent Beck, dove usa ampiamente queste tecniche in diversi piccoli esercizi.

CAPITOLO 19

Parte 2: Il Mondo Object-Oriented

{{keyToDo}} **Stato:** abbastanza stabile. Questo capitolo probabilmente riceverà una grande recensione in un lontano futuro. Anche se sono soddisfatto del contenuto, cercherò una struttura e una formulazione migliori, apportando piccoli cambiamenti qua e là. Potrei anche aggiungere diverse sezioni che spiegano cose ai capitoli esistenti su cose che trovo non siano state sufficientemente spiegate. Tuttavia, se lo si legge così com'è adesso, non si perderà nulla di significativo.

La maggior parte degli esempi nella parte precedente riguardavano unăsingolo oggetto che non aveva dipendenze da altri oggetti (ad eccezione di alcuni valori -- stringhe, numeri interi, enumerazioni, ecc.). Questo non è il modo in cui viene costruita la maggior parte dei sistemi OO. In questa parte, esamineremo finalmente gli scenari in cui più oggetti lavorano insieme come un sistema.

Ciò porta ad alcune questioni che devono essere discusse. Una di queste è l'approccio alla progettazione orientata agli oggetti e il modo in cui influenza gli strumenti che utilizziamo per testare il nostro codice. Probabilmente sarà noto uno strumento chiamato oggetti mock (almeno da uno dei capitoli introduttivi di questo libro) o, in un senso più ampio, di "test double" (https://it.wikipedia.org/wiki/Test_double). Se si apre il browser web e si digita "mock objects break encapsulation" [*gli oggetti mock interrompono l'incapsulamento*], si troveranno molte opinioni diverse: alcuni dicono che i mock sono fantastici, altri li incolpano di tutto il male del mondo e molte opinioni che ricadono nel mezzo. Le discussioni sono ancora accese, anche se i mock sono stati introdotti più di dieci anni fa. Il mio obiettivo in questo capitolo è delineare il contesto e le forze che portano all'adozione dei mock e come utilizzarli a proprio vantaggio, non per il fallimento.

Steve Freeman, uno dei padrini dell'utilizzo degli oggetti mock con TDD, ha scritto: "i mock sorgono naturalmente dal fare OO orientato alla responsabilità. Tutti questi argomenti mock/non-mock non colgono completamente il punto. Se non state scrivendo quel tipo di codice, gente, per favore non datemi filo da torcere". Presenterò i mock in un modo che non darà filo da torcere a Steve, spero.

Per fare ciò, devo trattare alcuni argomenti di progettazione orientata agli oggetti. In effetti, ho deciso di dedicare l'intera parte 2 esclusivamente a quello scopo. Pertanto, questo capitolo tratterà delle tecniche, delle pratiche e delle qualità orientate agli oggetti che è necessario conoscere per utilizzare il TDD in modo efficace nel mondo orientato agli oggetti. La qualità chiave su cui ci concentreremo è la componibilità degli oggetti.

{{keyExclamation}} **Insegnare una cosa alla volta.** Durante questa parte del libro, farò molti esempi di codice ed esempi di progettazione senza scrivere alcun test. Questo potrebbe far chiedersi se si sta ancora leggendo un libro TDD. {{keyExclamation}} Voglio che sia molto chiaro che omettendo i test in questi capitoli non sto sostenendo la scrittura di codice o il refactoring senza i test. L'unica ragione per cui lo faccio è che insegnare e imparare più cose allo stesso tempo può rendere tutto più difficile, sia per l'insegnante che per lo studente. Quindi, mentre vengono spiegati gli argomenti necessari per la progettazione orientata agli oggetti, voglio che ci si concentri solo su di essi. {{keyExclamation}} Nessuna preoccupazione. Dopo aver gettato le basi per gli oggetti mock, reintrodurrò il TDD nella parte 3 e scriverò molti test. Ci si fidi di me e siate paziente.

Dopo aver letto la parte 2, si capirà un approccio supponente alla progettazione orientata agli oggetti che si basa sull'idea che il sistema orientato agli oggetti sia una rete di nodi (oggetti) che si scambiano messaggi. Questo ci fornirà un buon punto di partenza per introdurre gli oggetti mock e il TDD basato su mock nella parte 3.

CAPITOLO 20

La Componibilità degli Oggetti

In questo capitolo, cercherò di delineare brevemente perché la componibilità degli oggetti è un obiettivo che vale la pena raggiungere e come può essere raggiunto. Inizierò con un esempio di codice non gestibile e ne risolverò gradualmente i difetti nei prossimi capitoli. Per ora, correggeremo solo uno dei difetti, quindi il codice che otterremo non sarà affatto perfetto, tuttavia, sarà migliore per una qualità.

Nei prossimi capitoli impareremo lezioni più preziose derivanti dalla modifica di questo piccolo pezzo di codice.

20.1 Un altro compito per Johnny e Benjamin

Ricordate Johnny e Benjamin? Sembra che abbiano gestito il loro compito precedente e stiano facendo qualcos'altro. Ascoltiamo la loro conversazione mentre stanno lavorando ad un altro progetto...

Benjamin: Allora, di cosa tratta questo incarico?

Johnny: In realtà, non è niente di entusiasmante -- dovremo aggiungere due funzionalità a un'applicazione legacy che non è preparata per le modifiche.

Benjamin: A cosa serve il codice?

Johnny: È una classeăC# che implementa le policy aziendali. Poiché l'azienda ha appena iniziato a utilizzare questo sistema automatizzato ed è stato avviato di recente, è stata implementata una sola politica: il piano di incentivi annuale. Molte aziende hanno quelli che chiamano piani di incentivi. Questi piani vengono utilizzati per promuovere comportamenti corretti e superare le aspettative da parte dei dipendenti di un'azienda.

Benjamin: Vuoi dire che il progetto è appena iniziato ed è già in cattive condizioni?

Johnny: Sì. I ragazzi che lo hanno scritto volevano "mantenerlo semplice", qualunque cosa significhi, e ora sembra piuttosto brutto.

Benjamin: Capisco...

Johnny: A proposito, ti piacciono gli indovinelli?

Benjamin: Sempre!

Johnny: Quindi eccone uno: come si definisce una fase di sviluppo quando si garantisce un'elevata qualità del codice?

Benjamin: ... ... Nessun indizio... Allora come si chiama?

Johnny: Si chiama "adesso".

Benjamin: Oh!

Johnny: Tornando all'argomento, ecco il piano di incentivi aziendale.

Ogni dipendente ha un grado di retribuzione. Un dipendente può essere promosso a un grado di retribuzione più elevato, ma i meccanismi di come funziona è qualcosa di cui non dovremo occuparci.

Normalmente, ogni anno, tutti ricevono un aumento del 10%. Ma per incoraggiare comportamenti che danno a un dipendente un grado di retribuzione più alto, tale dipendente non può ottenere aumenti indiscriminati su un determinato grado di retribuzione. Ad ogni grado è associata la retribuzione massima. Se viene raggiunta questa somma di denaro, un dipendente non riceve più un aumento finché non raggiunge un grado di retribuzione più elevato.

Inoltre, ogni dipendente che compie 5 anni di lavoro per l'azienda riceve un bonus speciale una tantum pari al doppio della retribuzione attuale.

Benjamin: Pare che il repository del codice sorgente abbia appena terminato la sincronizzazione. Diamo un assaggio al codice!

Johnny: Certo, ecco qui:

```
public class CompanyPolicies : IDisposable
{
  readonly SqlRepository _repository
    = new SqlRepository();

  public void ApplyYearlyIncentivePlan()
  {
    var employees = _repository.CurrentEmployees();

    foreach(var employee in employees)
    {
      var payGrade = employee.GetPayGrade();
      //evaluate raise
      if(employee.GetSalary() < payGrade.Maximum)
      {
        var newSalary
          = employee.GetSalary()
          + employee.GetSalary()
          * 0.1;
        employee.SetSalary(newSalary);
      }

      //evaluate one-time bonus
      if(employee.GetYearsOfService() == 5)
      {
        var oneTimeBonus = employee.GetSalary() * 2;
        employee.SetBonusForYear(2014, oneTimeBonus);
      }

      employee.Save();
    }
  }

  public void Dispose()
  {
    _repository.Dispose();
  }
}
```

Benjamin: Wow, ci sono un sacco di costanti letterali ovunque e la scomposizione funzionale è a malapena completata!

Johnny: Sì. Non aggiusteremo tutto questo oggi. Seguiremo comunque la regola dei boy scout e "lasceremo il campeggio più pulito di come lo abbiamo trovato".

Benjamin: Qual è il nostro compito?

Johnny: Prima di tutto, dobbiamo offrire ai nostri utenti la possibilità di scegliere tra un database SQL e uno No-SQL. Per raggiungere il nostro obiettivo, dobbiamo essere in qualche modo in grado di rendere il database della classe `CompanyPolicies` type-agnostic. Per ora, come puoi vedere, l'implementazione è accoppiata allo specifico `SqlRepository`, perché crea essa stessa un'istanza specifica:

```
public class CompanyPolicies : IDisposable
{
  readonly SqlRepository _repository
    = new SqlRepository();
```

Ora dobbiamo valutare le opzioni a nostra disposizione per scegliere quella migliore. Che opzioni vedi, Benjamin?

Benjamin: Beh, potremmo certamente estrarre un'interfaccia da `SqlRepository` e introdurre un'istruzione if nel costruttore in questo modo:

```
public class CompanyPolicies : IDisposable
{
  readonly Repository _repository;

  public CompanyPolicies()
  {
    if(...)
    {
      _repository = new SqlRepository();
    }
    else
    {
      _repository = new NoSqlRepository();
    }
  }
```

Johnny: Vero, ma questa opzione presenta poche carenze. Prima di tutto, ricorda che stiamo cercando di seguire la regola dei boy scout e utilizzando questa opzione introduciamo maggiore complessità nella classe `CommonPolicies`. Inoltre, supponiamo che domani qualcuno scriva un'altraăclasse, ad esempio, per il reporting e anche questa classe dovrà accedere al repository -- dovrà prendere la stessa decisione sui repository nel loro codice come facciamo noi nel nostro. Ciò significa effettivamente duplicare il codice. Quindi preferisco valutare ulteriori opzioni e vedere se possiamo trovare qualcosa di meglio. Qual è la nostra prossima opzione?

Benjamin: Un'altra opzione sarebbe quella di modificare lo stesso `SqlRepository` in modo che sia solo un wrapper per l'effettivo accesso al database, in questo modo:

```
public class SqlRepository : IDisposable
{
  readonly Repository _repository;

  public SqlRepository()
  {
    if(...)
    {
      _repository = new RealSqlRepository();
    }
    else
    {
      _repository = new RealNoSqlRepository();
    }
  }
```

(continues on next page)

(continua dalla pagina precedente)

```
  IList<Employee> CurrentEmployees()
  {
    return _repository.CurrentEmployees();
  }
```

Johnny: Certo, questo è un approccio che potrebbe funzionare e varrebbe la pena di essere preso in considerazione per codice legacy molto serio, poiché non ci obbliga affatto a modificare la classe `CompanyPolicies`. Tuttavia, ci sono alcuni problemi con esso. Innanzitutto il nome `SqlRepository` sarebbe fuorviante. In secondo luogo, guarda il metodo `CurrentEmployees()` -- tutto ciò che fa è delegare una chiamata all'implementazione scelta nel costruttore. Con ogni nuovo metodo richiesto dal repository, dovremo aggiungere nuovi metodi deleganti. In realtà, non è un grosso problema, ma forse possiamo fare di meglio?

Benjamin: Fammi pensare, fammi pensare... Ho valutato l'opzione in cui la classe `CompanyPolicies` sceglie tra i repository. Ho anche valutato l'opzione in cui hackeriamo `SqlRepository` per fare questa scelta. L'ultima opzione a cui riesco a pensare è lasciare questa scelta a un altro, il codice di "terze parti", che sceglierebbe il repository da utilizzare e lo passerebbe alla `CompanyPolicies` tramite il suo costruttore, in questo modo:

```
public class CompanyPolicies : IDisposable
{
  private readonly Repository _repository;

  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }
```

In questo modo, `CompanyPolicies` non saprà esattamente cosa gli viene passato tramite il suo costruttore e possiamo passare qualunque cosa vogliamo: un repository SQL o uno NoSQL!

Johnny: Fantastico! Questa è l'opzione che stiamo cercando! Per ora, credimi solo che questo approccio ci porterà a molte cose buone -- vedrai il perché più tardi.

Benjamin: OK, lasciami semplicemente estrarre l'istanza `SqlRepository` al di fuori della classe `CompanyPolicies` e renderla un'implementazione dell'interfaccia `Repository`, quindi creare un costruttore e passare l'istanza reale attraverso di esso...

Johnny: Certo, vado a prendere un caffè.

... 10 minuti dopo

Benjamin: Haha! Guarda questo! Sono un GRANDE!

```
public class CompanyPolicies : IDisposable
{
  //_repository is now an interface
  readonly Repository _repository;

  // repository is passed from outside.
  // We don't know what exact implementation it is.
  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  public void ApplyYearlyIncentivePlan()
  {
    //... body of the method. Unchanged.
  }

  public void Dispose()
```

(continues on next page)

(continua dalla pagina precedente)

```
  {
    _repository.Dispose();
  }
}
```

Johnny: Ehi, ehi, frena! C'è una cosa che non va in questo codice.

Benjamin: Eh? Pensavo che questo fosse ciò a cui miravamo.

Johnny: Sì, tranne il metodo `Dispose()`. Osserva attentamente la classe `CompanyPolicies`. è stata modificata in modo che non sia responsabile, da solo, della creazione di un repository, ma ne dispone comunque. Ciò potrebbe causare problemi perché l'istanza `CompanyPolicies` non ha alcun diritto di presumere che sia l'unico oggetto che utilizza il repository. In tal caso, non è possibile determinare il momento in cui il repository diventerà superfluo e potrà essere disposto in modo sicuro.

Benjamin: Ok, ho capito la teoria, ma perché in pratica è un male? Puoi farmi un esempio?

Johnny: Certo, lasciami fare un breve esempio. Non appena hai due istanze della classe `CompanyPolicies`, che condividono entrambe la stessa istanza di `Repository`, sei fritto. Questo perché un'istanza di `CompanyPolicies` potrebbe eliminare il repository mentre l'altra potrebbe comunque volerlo utilizzare.

Benjamin: Quindi chi eliminerà il repository?

Johnny: La stessa parte del codice che lo crea, ad esempio il metodo `Main`. Lascia che ti mostri un esempio di come potrebbe apparire:

```
public static void Main(string[] args)
{
  using(var repo = new SqlRepository())
  {
    var policies = new CompanyPolicies(repo);

    //use above created policies
    //for anything you like
  }
}
```

In questo modo il repository viene creato all'avvio del programma ed eliminato alla fine. Grazie a ciò, `CompanyPolicies` non ha campi usa e getta e non deve essere necessariamente usa e getta -- possiamo semplicemente eliminare [delete] il metodo `Dispose()`:

```
//not implementing IDisposable anymore:
public class CompanyPolicies
{
  //_repository is now an interface
  readonly Repository _repository;

  //New constructor
  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  public void ApplyYearlyIncentivePlan()
  {
    //... body of the method. No changes
  }

  //no Dispose() method anymore
}
```

Benjamin: Fantastico. E adesso? Sembra che la classe `CompanyPolicies` dipenda da un'astrazione del repository anziché da un'implementazione effettiva, come il repository SQL. Immagino che saremo in grado di creare un'altra classe che implementi l'interfaccia per l'accesso ai dati NoSQL e passarla semplicemente attraverso il costruttore invece di quello originale.

Johnny: Sì. Ad esempio, guarda il componente `CompanyPolicies`. Possiamo comporlo con un repository come questo:

```
var policies
  = new CompanyPolicies(new SqlRepository());
```

o così:

```
var policies
  = new CompanyPolicies(new NoSqlRepository());
```

senza modificare il codice di `CompanyPolicies`. Ciò significa che `CompanyPolicies` non ha bisogno di sapere di quale `Repository` è composto esattamente, purché questo `Repository` segua l'interfaccia richiesta e soddisfi le aspettative di `CompanyPolicies` (ad esempio, non generi eccezioni quando non è previsto fare così). Un'implementazione di `Repository` può essere essa stessa molto complessa e composta da un altro insieme di classi, ad esempio qualcosa del genere:

```
new SqlRepository(
  new ConnectionString("..."),
  new AccessPrivileges(
    new Role("Admin"),
    new Role("Auditor")
  ),
  new InMemoryCache()
);
```

ma `CompanyPolicies` non lo sa né se ne preoccupa, purché possa utilizzare la nostra nuova implementazione di `Repository` proprio come gli altri repository.

Benjamin: Capisco... Quindi, tornando al nostro compito, procediamo con la realizzazione di un'implementazione NoSQL dell'interfaccia `Repository`?

Johnny: Per prima cosa mostrami l'interfaccia che hai estratto mentre cercavo il caffè.

Benjamin: Ecco:

```
public interface Repository
{
  IList<Employee> CurrentEmployees();
}
```

Johnny: Ok, quindi ciò di cui abbiamo bisogno è creare semplicemente un'altra implementazione e passarla attraverso il costruttore a seconda dell'origine dati scelta e abbiamo finito con questa parte dell'attività.

Benjamin: Vuoi dire che c'è dell'altro?

Johnny: Sì, ma è qualcosa per domani. Sono esausto oggi.

20.2 Una Rapida Retrospettiva

In questo capitolo Benjamin ha imparato ad apprezzare la componibilità di un oggetto, ovvero la capacità di sostituirne le dipendenze, fornendo comportamenti diversi, senza la necessità di modificare il codice della classe dell'oggetto stesso. Pertanto, un oggetto, date le dipendenze sostituite, inizia a utilizzare i nuovi comportamenti senza notare che si è verificato alcun cambiamento.

Il codice menzionato presenta alcuni gravi difetti. Per ora, Johnny e Benjamin non hanno avvertito il disperato bisogno di affrontarli.

Inoltre, dopo che ci saremo nuovamente separati da Johnny e Benjamin, ribadiremo le idee in cui si sono imbattuti in modo più disciplinato.

CAPITOLO 21

Raccontare, non chiedere

In questo capitolo torneremo a Johnny e Benjamin mentre introducono un'altra modifica nel codice su cui stanno lavorando. Nel processo, scoprono l'impatto che i valori restituiti e i getter hanno sulla componibilità degli oggetti.

21.1 I Contractor

Johnny: Buongiorno. Pronti per un altro compito?

Benjamin: Naturalmente! Qual è il prossimo?

Johnny: Ricordi il codice su cui abbiamo lavorato ieri? Contiene una politica per i dipendenti regolari dell'azienda. Ma lazienda vuole iniziare ad assumere anche [contractor] e deve includere una politica per loro nella domanda..

Benjamin: Quindi questo è ciò che faremo oggi?

Johnny: Esatto. La politica sarà diversa per i [contractor]. Anche se, proprio come i dipendenti regolari, riceveranno aumenti e bonus, le regole saranno diverse. Ho creato una piccola tabella per consentire il confronto tra ciò che abbiamo per i dipendenti regolari e ciò che vogliamo aggiungere per i [contractor]:

Tipo di Impiegato	Aumento	Bonus
Impiegato Regolare	+10% dello stipendio attuale se non viene raggiunto il massimo in un determinato livello retributivo	+200% dello stipendio attuale una volta dopo cinque anni
[Contractor]	+5% della retribuzione media calcolata sugli ultimi 3 anni di servizio (o su tutti gli anni di servizio precedenti se hanno lavorato per meno di 3 anni)	+10% dello stipendio attuale quando un [contractor] riceve un punteggio superiore a 100 per l'anno precedente

Quindi, anche se il flusso di lavoro sarà lo stesso sia per un dipendente regolare che per un [contractor]:

1. Caricare dal repository
2. Valutare l'aumento
3. Valutare il bonus
4. Salvare

l'implementazione di alcuni passaggi sarà diversa per ciascuna tipo di dipendente.

Benjamin: Correggimi se sbaglio, ma questi passaggi "caricare" e "salvare" non sembrano appartenere ai restanti due: descrivono qualcosa di tecnico, mentre gli altri passaggi descrivono qualcosa strettamente correlato a come l'azienda opera...

Johnny: Bell'idea, comunque, è qualcosa di cui ci occuperemo più tardi. Ricorda la regola dei boy scout: non peggiorare le cose. Tuttavia, oggi risolveremo alcuni dei difetti di progettazione.

Benjamin: Aww... Io risolverei tutto subito.

Johnny: Ah ah, pazienza, Luke. Per ora, diamo un'occhiata al codice che abbiamo ora prima di pianificare ulteriori passaggi.

Benjamin: Fammi solo aprire il mio IDE... OK, eccolo qui:

```
public class CompanyPolicies
{
  readonly Repository _repository;

  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  public void ApplyYearlyIncentivePlan()
  {
    var employees = _repository.CurrentEmployees();

    foreach(var employee in employees)
    {
      var payGrade = employee.GetPayGrade();

      //evaluate raise
      if(employee.GetSalary() < payGrade.Maximum)
      {
        var newSalary
          = employee.GetSalary()
          + employee.GetSalary()
          * 0.1;
        employee.SetSalary(newSalary);
      }

      //evaluate one-time bonus
      if(employee.GetYearsOfService() == 5)
      {
        var oneTimeBonus = employee.GetSalary() * 2;
        employee.SetBonusForYear(2014, oneTimeBonus);
      }

      employee.Save();
    }
  }
}
```

Benjamin: Guarda, Johnny, la classe, in effetti, contiene tutti e quattro i passaggi che hai citato, ma non sono nominati esplicitamente, invece la loro implementazione interna per i dipendenti regolari è semplicemente inserita qui. Come dovremmo aggiungere la variazione del tipo di dipendente?

Johnny: È ora di considerare le nostre opzioni. Ne abbiamo alcune. Bene?

Benjamin: Per ora ne vedo due. Il primo sarebbe creare un'altra classe simile a `CompanyPolicies`, chiamata qualcosa come `CompanyPoliciesForContractors` e implementare lì la nuova logica. Ciò ci consentirebbe di lasciare la classe originale così com'è, ma dovremmo cambiare i posti che utilizzano `CompanyPolicies` per utilizzare entrambe le classi e scegliere quale utilizzare in qualche modo. Inoltre, dovremmo aggiungere un metodo separato al repository per recuperare i [contractor].

Johnny: Inoltre, perderemmo l'occasione di comunicare attraverso il codice che la sequenza di passaggi è intenzionalmente simile in entrambi i casi. Altri che leggeranno questo codice in futuro vedranno che l'implementazione per i dipendenti regolari segue i passaggi: caricare, valutare l'aumento, valutare il bonus, salvare. Quando esamineranno l'implementazione per i [contractor], vedranno lo stesso ordine di passaggi, ma non saranno in grado di dire se la somiglianza è intenzionale o un puro incidente.

Benjamin: Quindi la nostra seconda opzione è inserire un'istruzione `if` nei diversi passaggi all'interno della classe `CompanyPolicies`, per distinguere tra dipendenti regolari e [contractor]. La classe `Employee` avrebbe un metodo `isContractor()` e, a seconda di cosa restituirebbe, eseguiremmo la logica per dipendenti regolari o i [contractor]. Supponendo che l'attuale struttura del codice sia simile alla seguente:

```
foreach(var employee in employees)
{
  //evaluate raise
  ...

  //evaluate one-time bonus
  ...

  //save employee
}
```

la nuova struttura sarebbe simile a questa:

```
foreach(var employee in employees)
{
  if(employee.IsContractor())
  {
    //evaluate raise for contractor
    ...
  }
  else
  {
    //evaluate raise for regular
    ...
  }

  if(employee.IsContractor())
  {
    //evaluate one-time bonus for contractor
    ...
  }
  else
  {
    //evaluate one-time bonus for regular
    ...
  }

  //save employee
  ...
}
```

in questo modo dimostreremmo che i passaggi sono gli stessi, ma l'implementazione è diversa. Inoltre, ciò richiederebbe principalmente di aggiungere codice e di non spostare il codice esistente.

Johnny: Lo svantaggio è che renderemmo la classe ancora più brutta di quando abbiamo iniziato. Quindi, nonostante la facilità iniziale, renderemo un enorme disservizio ai futuri manutentori. Abbiamo almeno un'altra opzione. Cosa sarebbe?

Benjamin: Vediamo... potremmo spostare tutti i dettagli riguardanti l'implementazione dei passaggi dalla classe `CompanyPolicies` nella classe `Employee` stessa, lasciando solo i nomi e l'ordine dei passaggi in `CompanyPolicies`:

```
foreach(var employee in employees)
{
  employee.EvaluateRaise();
  employee.EvaluateOneTimeBonus();
  employee.Save();
}
```

Quindi, potremmo trasformare `Employee` in un'interfaccia, in modo che possa essere un `RegularEmployee` o un `ContractorEmployee` -- entrambe le classi avrebbero implementazioni diverse dei passaggi, ma `CompanyPolicies` non se ne accorgerebbe, poiché non sarebbe più abbinata all'implementazione dei passaggi -- solo i nomi e l'ordine.

Johnny: Questa soluzione avrebbe uno svantaggio -- dovremmo modificare in modo significativo il codice attuale, ma sai una cosa? Sono disposto a farlo, soprattutto perché oggi mi è stato detto che la logica è coperta da alcuni test che possiamo eseguire per vedere se è stata introdotta una regressione.

Benjamin: Ottimo, con cosa iniziamo?

Johnny: La prima cosa che si frappone tra noi e il nostro obiettivo sono questi getter nella classe `Employee`:

```
GetSalary();
GetGrade();
GetYearsOfService();
```

Espongono semplicemente troppe informazioni specifiche ai dipendenti regolari. Sarebbe impossibile utilizzare implementazioni diverse quando queste sono disponibili. Questi setter non aiutano molto:

```
SetSalary(newSalary);
SetBonusForYear(year, amount);
```

Anche se questi non sono poi così male, faremmo meglio a concederci maggiore flessibilità. Quindi nascondiamo tutto dietro metodi più astratti che rivelano solo la nostra intenzione.

Innanzitutto, dai un'occhiata a questo codice:

```
//evaluate raise
if(employee.GetSalary() < payGrade.Maximum)
{
  var newSalary
    = employee.GetSalary()
    + employee.GetSalary()
    * 0.1;
  employee.SetSalary(newSalary);
}
```

Ogni volta che si vede un blocco di codice separato dal resto con righe vuote e che inizia con un commento, c'è qualcosa che urla: "Voglio essere un metodo separato che contenga questo codice e abbia un nome dopo il commento!". Esaudiamo questo desiderio e rendiamolo un metodo separato nella classe `Employee`.

Benjamin: Ok, aspetta un attimo... ecco:

```
employee.EvaluateRaise();
```

Johnny: Fantastico! Ora, abbiamo un altro esempio di questa specie:

```
//evaluate one-time bonus
if(employee.GetYearsOfService() == 5)
{
  var oneTimeBonus = employee.GetSalary() * 2;
  employee.SetBonusForYear(2014, oneTimeBonus);
}
```

Benjamin: Questo dovrebbe essere ancora più semplice... Ok, dai un'occhiata:

```
employee.EvaluateOneTimeBonus();
```

Johnny: Quasi bene. Tralascerei solo l'informazione che il bonus è una tantum dal nome.

Benjamin: Perché? Non vogliamo includere ciò che accade nel nome del metodo?

Johnny: A dire il vero no. Ciò che vogliamo includere è la nostra intenzione. Il bonus una tantum è qualcosa di specifico per i dipendenti regolari e vogliamo eliminare i dettagli su questo o quel tipo di dipendente, in modo da poter collegare diverse implementazioni mentire nel nome del metodo. I nomi dovrebbero riflettere il fatto che vogliamo valutare un bonus, qualunque cosa significhi per un particolare tipo di dipendente. Quindi, facciamolo:

```
employee.EvaluateBonus();
```

Benjamin: Ok, ho capito. Nessun problema.

Johnny: Ora diamo un'occhiata al codice completo del metodo `EvaluateIncentivePlan` per vedere se è ancora abbinato ai dettagli specifici dei dipendenti regolari. Ecco il codice:

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
    employee.Save();
  }
}
```

Benjamin: Sembra che non ci sia più alcun collegamento con i dettagli sui dipendenti regolari. Pertanto, possiamo tranquillamente fare in modo che il repository restituisca una combinazione di clienti abituali e [contractor] senza che questo codice si accorga di nulla. Ora penso di capire cosa stavi cercando di ottenere. Se facciamo in modo che le interazioni tra gli oggetti avvengano a un livello più astratto, possiamo inserire implementazioni diverse con meno lavoro.

Johnny: Vero. Riesci a vedere un'altra cosa relativa alla mancanza di valori di ritorno su tutti i metodi di `Employee` nell'attuale implementazione?

Benjamin: Non proprio. Ha importanza?

Johnny: Bene, se i metodi `Employee` avessero valori di ritorno e questo codice dipendesse da essi, anche tutte le sottoclassi di `Employee` sarebbero costrette a fornire valori di ritorno e questi valori dovrebbero corrispondere alle aspettative del codice che chiama tali metodi, qualunque fossero queste aspettative. Ciò renderebbe più difficile lintroduzione di altri tipi di dipendenti. Ma ora che non ci sono valori da restituire, possiamo, ad esempio:

- introdurre un `TemporaryEmployee` che non ha aumenti, lasciando vuoto il suo metodo `EvaluateRaise()`, e il codice che utilizza i dipendenti non se ne accorgerà.
- introdurre un `ProbationEmployee` che non ha una politica di bonus, lasciando vuoto il suo metodo `EvaluateBonus()`, e il codice che utilizza i dipendenti non se ne accorgerà.
- introdurre un `InMemoryEmployee` che abbia un metodo `Save()` vuoto, e il codice che utilizza i dipendenti non se ne accorgerà.

Come si vede, chiedendo meno agli oggetti, e dicendogli di più, otteniamo maggiore flessibilità per creare implementazioni alternative e la componibilità, di cui abbiamo parlato ieri, aumenta!

Benjamin: Capisco... Quindi dire agli oggetti cosa fare invece di chiedere loro i dati rende le interazioni tra oggetti più astratte e quindi più stabili, aumentando la componibilità degli oggetti interagenti. Questa è una lezione preziosa -- è la prima volta che lo sento e sembra un concetto piuttosto potente.

21.2 Una Rapida Retrospettiva

In questo capitolo Benjamin ha imparato che la componibilità di un oggetto (per non parlare della chiarezza) è rafforzata quando le interazioni tra esso e i suoi pari sono: astratte, logiche e stabili. Inoltre, ha scoperto, con l'aiuto di Johnny, che esso viene ulteriormente rafforzato seguendo uno stile di progettazione in cui agli oggetti viene detto cosa fare invece di chiedere di fornire informazioni a qualcuno che poi decide per loro conto. Questo perché se un'API di un'astrazione è costruita rispondendo a domande specifiche, i client dell'astrazione tendono a porre molte domande e sono associati sia a quelle domande che ad alcuni aspetti delle risposte (cioè cosa c'è nei valori restituiti). Ciò rende più difficile la creazione di un'altra implementazione dell'astrazione, perché ogni nuova implementazione dell'astrazione deve non solo fornire risposte a tutte quelle domande, ma le risposte sono limitate a ciò che il cliente si aspetta. Quando all'astrazione viene semplicemente detto ciò che il suo *client* vuole che raggiunga, i *client* sono separati dalla maggior parte dei dettagli su come ciò avvenga. Questo semplifica l'introduzione di nuove implementazioni di astrazione: spesso ci consente anche di definire implementazioni con tutti i metodi vuoti senza che il client se ne accorga.

Queste sono tutte conclusioni importanti che ci porteranno verso il TDD con oggetti mock.

Lasciamo, per ora, Johnny e Benjamin. Nel prossimo capitolo ribadirò le loro scoperte e le inserirò in un contesto più ampio.

CAPITOLO 22

La necessità degli oggetti mock

Abbiamo già sperimentato gli oggetti mock nel capitolo sugli strumenti, anche se a quel punto vi ho dato una spiegazione semplicistica e ingannevole di cosa sia un oggetto mock, promettendovi che avrei rimediato in seguito. Ora è il momento.

Gli oggetti mock sono stati realizzati con un obiettivo specifico in mente. Spero che quando si capirà il vero obiettivo, probabilmente si capiranno molto meglio i mezzi per raggiungerlo.

In questo capitolo esploreremo le qualità della progettazione orientata agli oggetti che rendono gli oggetti mock uno strumento valido.

22.1 Componibilità... ancora!

Nei due capitoli precedenti abbiamo seguito Johnny e Benjamin alla scoperta dei vantaggi e dei prerequisiti della componibilità degli oggetti. La componibilità è la qualità numero uno del design che ricerchiamo. Dopo aver letto la storia di Johhny e Benjamin, ci sono alcune domande sulla componibilità. Si spera che siano tra quelle a cui verrà data risposta nei prossimi capitoli. Pronti?

CAPITOLO 23

Perché abbiamo bisogno della componibilità?

Potrebbe sembrare stupido porre questa domanda a questo punto -- se si è arrivati qui, probabilmente non si è abbastanza motivati da non aver bisogno di una giustificazione? Beh, vale comunque la pena discuterne un po'. Se tutto va bene, si imparerà leggendo questo capitolo di "ritorno alle origini" tanto quanto io ho imparato scrivendolo.

23.1 Gli approcci pre-object-oriented

Ai tempi della programmazione procedurale[1], quando volevamo eseguire codice diverso in base a qualche fattore, di solito lo ottenevamo utilizzando un'istruzione 'if'. Ad esempio, se la nostra applicazione avesse bisogno di poter utilizzare diversi tipi di allarmi, come uno sonoro (che emette un suono forte) e un allarme silenzioso (che non emette alcun suono, ma contatta invece silenziosamente la polizia) in modo intercambiabile, solitamente potremmo ottenere questo risultato utilizzando un condizionale come nella funzione seguente:

```
void triggerAlarm(Alarm* alarm)
{
  if(alarm->kind == LOUD_ALARM)
  {
    playLoudSound(alarm);
  }
  else if(alarm->kind == SILENT_ALARM)
  {
    notifyPolice(alarm);
  }
}
```

Il codice sopra prende una decisione in base al tipo di allarme incluso nella struttura dell'allarme:

```
struct Alarm
{
  int kind;
  //other data
};
```

[1] Sto semplificando la discussione di proposito, tralasciando ad es. i linguaggi funzionali e assumendo che "pre-object-oriented" significhi procedurale e strutturale. Anche se questo non è vero in generale, questa è la realtà per molti di noi. Gli esperti nella programmazione funzionale, comprendono già i vantaggi della componibilità.

Se il tipo di allarme è quello sonoro, esegue il comportamento associato ad un allarme sonoro. Se si tratta di un allarme silenzioso, viene eseguito il comportamento per gli allarmi silenziosi. Questo sembra funzionare. Sfortunatamente, se volessimo prendere una seconda decisione in base al tipo di allarme (ad esempio, dovessimo disabilitare l'allarme), dovremmo interrogare nuovamente il tipo di allarme. Ciò significherebbe duplicare il codice condizionale, semplicemente con una serie diversa di azioni da eseguire, a seconda del tipo di allarme con cui abbiamo a che fare:

```
void disableAlarm(Alarm* alarm)
{
  if(alarm->kind == LOUD_ALARM)
  {
    stopLoudSound(alarm);
  }
  else if(alarm->kind == SILENT_ALARM)
  {
    stopNotifyingPolice(alarm);
  }
}
```

Devo dire perché questa duplicazione è negativa? Sento un "no"? Allora mi scuso, ma lo dirò comunque. La duplicazione significa che ogni volta che viene introdotto un nuovo tipo di allarme, uno sviluppatore deve ricordarsi di aggiornare entrambe le posizioni che contengono 'if-else' -- il compilatore non lo obbligherà. Nel contesto dei team, dove uno sviluppatore riprende il lavoro lasciato da un altro e dove, di tanto in tanto, le persone se ne vanno per trovare un altro lavoro, aspettandosi che qualcuno si "ricordi" di aggiornare tutti i posti in cui la logica è duplicata è in cerca di guai.

Quindi, constatiamo che la duplicazione è negativa, ma possiamo fare qualcosa al riguardo? Per rispondere a questa domanda, diamo un'occhiata al motivo per cui è stata introdotta la duplicazione. E il motivo è: vogliamo poter fare due cose con i nostri allarmi: attivarli e disattivarli. In altre parole, abbiamo una serie di domande a cui vogliamo poter porre ad un allarme. Ogni tipo di allarme ha un modo diverso di rispondere a queste domande, ne risultano una serie di "risposte" specifiche per ciascun tipo di allarme:

Tipo di Allarme	Attivazione	Disattivazione
Allarme Sonoro	`playLoudSound()`	`stopLoudSound()`
Allarme Silenzioso	`notifyPolice()`	`stopNotifyingPolice()`

Quindi, almeno concettualmente, non appena conosciamo il tipo di allarme, sappiamo già di quale insieme di comportamenti (rappresentati come una riga nella tabella sopra) ha bisogno. Potremmo semplicemente decidere il tipo di allarme una volta e associare il giusto insieme di comportamenti alla struttura dei dati. Quindi, non dovremmo interrogare il tipo di allarme in più posti come abbiamo fatto, ma potremmo invece dire: "esegui il comportamento di attivazione dall'insieme di comportamenti associati a questo allarme, qualunque esso sia".

Sfortunatamente, la programmazione procedurale non consente facilmente il comportamento vincolato ai dati. Lintero paradigma della programmazione procedurale riguarda la separazione del comportamento dai dati! Beh, onestamente, avevano alcune risposte a queste preoccupazioni, ma queste risposte erano per lo più imbarazzanti (per quelli di voi che ricordano ancora il linguaggio C: sto parlando di macro e puntatori a funzioni). Pertanto, poiché i dati e il comportamento sono separati, dobbiamo interrogare i dati ogni volta che vogliamo scegliere un comportamento basato su di essi. Ecco perché abbiamo la duplicazione.

23.2 La programmazione orientata agli oggetti in soccorso!

D'altra parte, la programmazione orientata agli oggetti ha da tempo reso disponibili due meccanismi che abilitano ciò che non avevamo nei linguaggi procedurali::

1. Le classi -- che consentono un comportamento vincolante insieme ai dati.
2. Il polimorfismo -- che consente di eseguire comportamenti senza conoscere l'esatta classe che li contiene, ma conoscendo solo un insieme di comportamenti che supporta. Questa conoscenza si ottiene facendo sì che un tipo astratto (interfaccia o classe astratta) definisca questo insieme di comportamenti, senza alcuna implementazione reale. Da cui possiamo creare altre classi che forniscano la propria implementazione dei comportamenti dichiarati supportati

dal tipo astratto. Infine possiamo utilizzare le istanze di quelle classi in cui è prevista un'istanza di tipo astratto. Nel caso dei linguaggi tipizzati staticamente, ciò richiede l'implementazione di un'interfaccia o l'ereditarietà da una classe astratta.

Quindi, in caso di nostri allarmi, potremmo creare un'interfaccia con la seguente firma:

```
public interface Alarm
{
  void Trigger();
  void Disable();
}
```

e quindi creare due class: `LoudAlarm` e `SilentAlarm`, entrambe implementanti l'interfaccia `Alarm`. Esempio per `LoudAlarm`:

```
public class LoudAlarm : Alarm
{
  public void Trigger()
  {
    //play very loud sound
  }

  public void Disable()
  {
    //stop playing the sound
  }
}
```

Ora possiamo fare in modo che parti di codice utilizzino l'allarme, ma conoscendo solo l'interfaccia invece delle classi concrete. Ciò fa sì che le parti del codice che utilizzano l'allarme in questo modo non debbano controllare con quale allarme hanno a che fare. Quindi, quello che prima appariva così:

```
if(alarm->kind == LOUD_ALARM)
{
  playLoudSound(alarm);
}
else if(alarm->kind == SILENT_ALARM)
{
  notifyPolice(alarm);
}
```

diventa semplicemente:

```
alarm.Trigger();
```

dove `alarm` è `LoudAlarm` o `SilentAlarm` ma visto polimorficamente come `Alarm`, quindi non è più necessario 'if-else'.

Ma ehi, questo non è un imbroglio? Anche se posso eseguire il comportamento di attivazione su un allarme senza conoscere la classe effettiva dell'allarme, devo comunque decidere quale classe è nel posto in cui creo l'istanza effettiva:

```
// we must know the exact type here:
alarm = new LoudAlarm();
```

quindi sembra che dopo tutto non sto eliminando lo 'else-if' ma semplicemente spostandolo da qualche altra parte! Questo può essere vero (ne parleremo più approfonditamente nei capitoli futuri), ma la buona notizia è che ho eliminato almeno la duplicazione realizzando il nostro sogno di "scegliere il giusto insieme di comportamenti da utilizzare con determinati dati una sola volta".

Grazie a questo, creo l'allarme una sola volta, e poi posso prenderlo e passarlo a dieci, cento o mille posti diversi dove non dovrò più determinare il tipo di allarme per usarlo correttamente.

Questo consente di scrivere molte classi che non conoscono la vera classe dell'allarme con cui hanno a che fare, ma possono utilizzare l'allarme correttamente solo conoscendo un tipo astratto comune -- `Alarm`. Se riusciamo a farlo, arriviamo a una situazione in cui possiamo aggiungere più allarmi implementando `Alarm` e osservare gli oggetti esistenti che già utilizzano `Alarm` funzionare con questi nuovi allarmi senza alcuna modifica nel loro codice sorgente! C'è una condizione, però -- la **creazione delle istanze di alarm deve essere spostata fuori dalle classi che le utilizzano**. Questo perché, come abbiamo già osservato, per creare un allarme utilizzando un operatore `new`, dobbiamo conoscere il tipo esatto di allarme che stiamo creando. Quindi chi crea un'istanza di `LoudAlarm` o di `SilentAlarm`, perde la sua uniformità, poiché non può dipendere esclusivamente dall'interfaccia `Alarm`.

23.3 Il potere della composizione

Spostare la creazione di istanze di alarm lontano dalle classi che utilizzano quegli allarmi solleva un problema interessante: se un oggetto non crea gli oggetti che usa, allora chi lo fa? Una soluzione è quella di creare alcuni posti speciali nel codice che siano responsabili solo della composizione di un sistema da oggetti indipendenti dal contesto[2]. Lo abbiamo già visto mentre Johnny spiegava la componibilità a Benjamin. Ha usato il seguente esempio:

```
new SqlRepository(
  new ConnectionString("..."),
  new AccessPrivileges(
    new Role("Admin"),
    new Role("Auditor")
  ),
  new InMemoryCache()
);
```

Possiamo fare lo stesso con i nostri allarmi. Diciamo che abbiamo un'area da assicurare con tre edifici con diverse politiche di allarme:

- Uffici -- l'allarme dovrebbe avvisare le guardie in modo silenzioso durante il giorno (per impedire al personale dell'ufficio di farsi prendere dal panico) e sonoramente durante la notte, quando le guardie sono di pattuglia.
- Magazzini -- poiché è abbastanza lontano e gli operai sono pochi, vogliamo far scattare allarmi sonori e silenziosi allo stesso tempo.
- Guardiola -- poiché le guardie sono lì, non è necessario avvisarle. Tuttavia, un allarme silenzioso dovrebbe invece chiamare la polizia per chiedere aiuto ed è auspicabile anche un allarme sonoro.

Notare che oltre ad attivare semplicemente un allarme sonoro o silenzioso, dobbiamo supportare una combinazione ("allarmi sonori e silenziosi allo stesso tempo") e condizionale ("silenzioso di giorno e sonoro di notte"). potremmo semplicemente codificare alcuni `for` e `if-else` nel codice, ma invece, fattorizziamo queste due operazioni (combinazione e scelta) in classi separate che implementano l'interfaccia di alarm.

Chiamiamo la classe che implementa la scelta tra due allarmi `DayNightSwitchedAlarm`. Ecco il codice sorgente:

```
public class DayNightSwitchedAlarm : Alarm
{
  private readonly Alarm _dayAlarm;
  private readonly Alarm _nightAlarm;

  public DayNightSwitchedAlarm(
    Alarm dayAlarm,
    Alarm nightAlarm)
  {
    _dayAlarm = dayAlarm;
    _nightAlarm = nightAlarm;
  }

  public void Trigger()
```

(continues on next page)

[2] Maggiori informazioni sull'indipendenza dal contesto e su cosa siano questi "luoghi speciali" nei prossimi capitoli.

(continua dalla pagina precedente)

```
  {
    if(/* is day */)
    {
      _dayAlarm.Trigger();
    }
    else
    {
      _nightAlarm.Trigger();
    }
  }

  public void Disable()
  {
    _dayAlarm.Disable();
    _nightAlarm.Disable();
  }
}
```

Studiando il codice sopra, è evidente che questo non è un allarme *di per sé*, ad es. non emette alcun suono o notifica, ma contiene alcune regole su come utilizzare altri allarmi. Questo è lo stesso concetto dei deviatori di corrente nella vita reale, che agiscono come dispositivi elettrici ma non fanno altro che reindirizzare l'elettricità verso altri dispositivi.

Successivamente, utilizziamo lo stesso approccio e modelliamo la combinazione di due allarmi come una classe chiamata `HybridAlarm`. Ecco il codice sorgente:

```
public class HybridAlarm : Alarm
{
  private readonly Alarm _alarm1;
  private readonly Alarm _alarm2;

  public HybridAlarm(
    Alarm alarm1,
    Alarm alarm2)
  {
    _alarm1 = alarm1;
    _alarm2 = alarm2;
  }

  public void Trigger()
  {
    _alarm1.Trigger();
    _alarm2.Trigger();
  }

  public void Disable()
  {
    _alarm1.Disable();
    _alarm2.Disable();
  }
}
```

Con queste due classi insieme agli allarmi già esistenti, possiamo implementare i requisiti componendo istanze di quelle classi in questo modo::

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
```

(continues on next page)

(continua dalla pagina precedente)

```
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      new SilentAlarm("919"), //call police
      new LoudAlarm()
    )
  )
);
```

Si noti che il fatto di aver implementato la combinazione e la scelta degli allarmi come oggetti separati implementando l'interfaccia `Alarm` ci consente di definire nuovi e interessanti comportamenti degli allarmi utilizzando le parti che già abbiamo, ma componendole insieme in modo diverso. Ad esempio, potremmo avere, come nell'esempio sopra:

```
new DayNightSwitchAlarm(
  new SilentAlarm("222-333-444"),
  new LoudAlarm());
```

ciò significherebbe attivare un allarme silenzioso durante il giorno e uno sonoro durante la notte. Tuttavia, invece di questa combinazione, potremmo usare:

```
new DayNightSwitchAlarm(
  new SilentAlarm("222-333-444"),
  new HybridAlarm(
    new SilentAlarm("919"),
    new LoudAlarm()
  )
)
```

Ciò significherebbe che utilizziamo un allarme silenzioso per avvisare le guardie durante il giorno, ma una combinazione di silenzioso (avvisando la polizia) e sonoro durante la notte. Naturalmente non ci limitiamo a combinare solo un allarme silenzioso con uno sonoro. Possiamo anche combinarne due silenziosi:

```
new HybridAlarm(
  new SilentAlarm("919"),
  new SilentAlarm("222-333-444")
)
```

Inoltre, se all'improvviso decidessimo di non volere alcun allarme durante il giorno, potremmo utilizzare una classe speciale chiamata `NoAlarm` che implementerebbe l'interfaccia `Alarm` ma i metodi `Trigger` e `Disable` non farebbero nulla. Il codice della composizione sarebbe simile a questo:

```
new DayNightSwitchAlarm(
  new NoAlarm(), // no alarm during the day
  new HybridAlarm(
    new SilentAlarm("919"),
    new LoudAlarm()
  )
)
```

E, per ultimo ma non meno importante, potremmo rimuovere completamente tutti gli allarmi dall'edificio delle guardie utilizzando la seguente classe `NoAlarm` (che è anche un `Alarm`):

```
public class NoAlarm : Alarm
{
  public void Trigger()
  {
  }

  public void Disable()
  {
  }
}
```

e passandolo come allarme alla guardiola:

```
new GuardsBuilding(
  new NoAlarm()
)
```

Notate qualcosa di divertente negli ultimi esempi? In caso contrario, ecco una spiegazione: negli ultimi esempi abbiamo stravolto i comportamenti della nostra applicazione, ma tutto ciò è avvenuto nel codice di composizione! Non abbiamo dovuto modificare nessun'altra classe esistente! È vero, abbiamo dovuto scrivere una nuova classe chiamata `NoAlarm`, ma non abbiamo avuto bisogno di modificare alcun codice tranne quello di composizione per far funzionare gli oggetti di questa nuova classe con quelli di classi esistenti!

Questa capacità di cambiare il comportamento della nostra applicazione semplicemente cambiando il modo in cui gli oggetti sono composti insieme è estremamente potente (anche se talvolta lo si potrà ottenere parzialmente), specialmente nella progettazione evolutiva e incrementale, dove vogliamo evolvere alcuni pezzi di codice con il minor numero possibile di altri pezzi di codice per evolvere. Questa capacità si può ottenere solo se il sistema è costituito da oggetti componibili, da qui la necessità di componibilità: una risposta a una domanda sollevata all'inizio di questo capitolo.

23.4 Sommario -- siete ancora con me?

Abbiamo iniziato con quella che sembrava essere una ripetizione di un corso base di programmazione orientata agli oggetti, utilizzando un esempio base. È stato tuttavia necessario effettuare una transizione fluida ai vantaggi della componibilità introdotta alla fine. Spero che non sia stato troppo e che si possa ora capire perché sto dando così tanta importanza alla componibilità.

Nel prossimo capitolo daremo uno sguardo più approfondito alla composizione stessa degli oggetti.

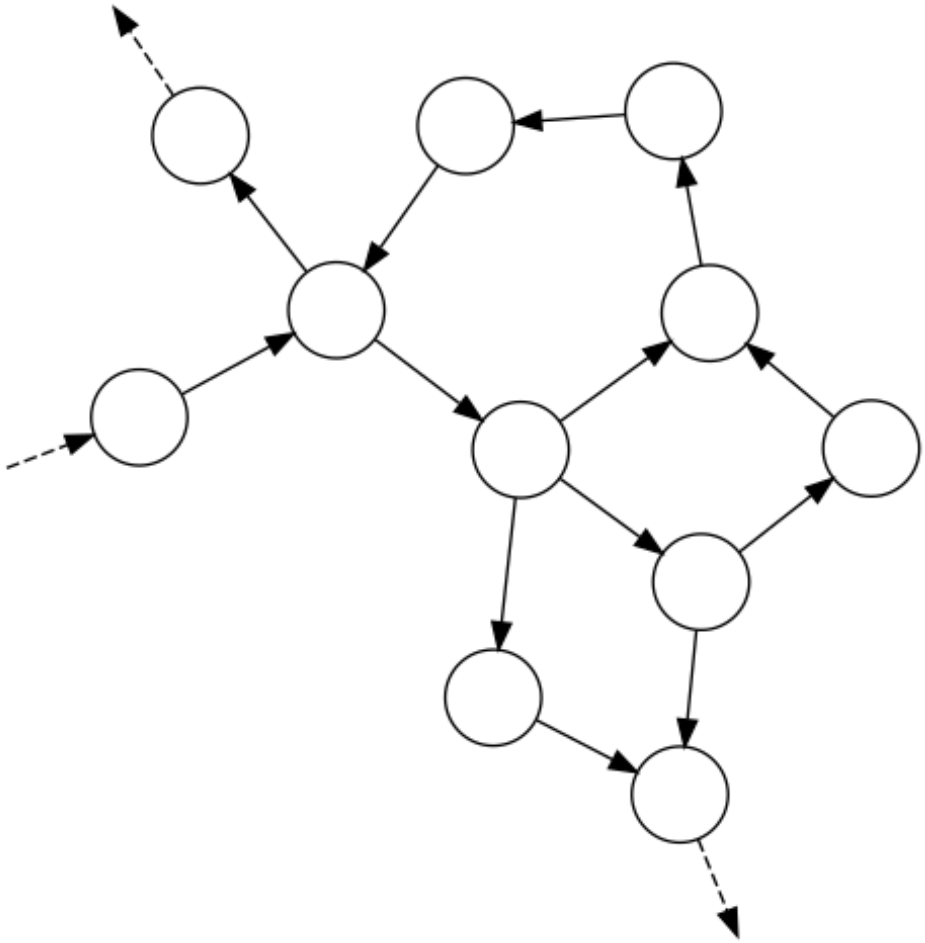
CAPITOLO 24

Reti, messaggi e protocolli

Nel capitolo precedente abbiamo parlato un po' del motivo per cui la componibilità è preziosa, ora arricchiamo un po' la terminologia per ottenere una comprensione più precisa.

24.1 Quindi, ancora una volta, cosa significa comporre oggetti?

In parole povere significa che un oggetto ha ottenuto un riferimento a un altro oggetto e può invocare dei suoi metodi. Essendo composti insieme, due oggetti formano un piccolo sistema che può essere ampliato con più oggetti secondo le necessità. Pertanto, un sistema object-oriented più grande forma qualcosa di simile a una rete:



Se portiamo la metafora della rete un po' oltre, possiamo notare alcune somiglianze con ad es. una rete TCP/IP:

1. Un oggetto può inviare **messaggi** ad altri oggetti (cioè chiamare dei loro metodi -- frecce nel diagramma sopra) tramite **interfacce**. Ogni messaggio ha un **mittente** e almeno un **destinatario**.
2. Per inviare un messaggio a un destinatario, un mittente deve acquisire un **indirizzo** del destinatario, che, nel mondo object-oriented, chiamiamo riferimento (e in linguaggi come C++, i riferimenti sono proprio questo: indirizzi in memoria).
3. La comunicazione tra il mittente e i destinatari deve seguire un determinato **protocollo**. Ad esempio, il mittente solitamente non può invocare un metodo che non passa nulla come argomenti, o dovrebbe aspettarsi un'eccezione se lo fa. (Niente paura se non si vede ora l'analogia -- proseguirò con ulteriori spiegazioni su questo argomento in seguito).

24.2 Allarmi, ancora!

Proviamo ad applicare questa terminologia ad un esempio. Immaginiamo di avere un sistema di allarme antincendio in un ufficio. Quando attivato, questo sistema di allarme fa scendere tutti gli ascensori al piano inferiore, li apre e poi li disattiva tutti. Tra gli altri, l'ufficio contiene ascensori automatici, che contengono i propri sistemi di controllo remoto e ascensori meccanici, che sono controllati dall'esterno da uno speciale meccanismo su misura.

Proviamo a modellare questo comportamento nel codice. Come intuito, avremo alcuni oggetti come allarme, ascensore automatico e ascensore meccanico. L'allarme controllerà gli ascensori quando attivato.

In primo luogo, non vogliamo che l'allarme debba distinguere tra un ascensore automatico e uno meccanico -- ciò non farebbe altro che aggiungere complessità al sistema di allarme, soprattutto perché si prevede di aggiungere un terzo tipo di ascensore -- uno più moderno -- in futuro. Quindi, se rendessimo l'allarme consapevole dei diversi tipi di ascensore, dovremmo modificarlo ogni volta che viene introdotto un nuovo tipo di ascensore. Pertanto, abbiamo bisogno di una **interfaccia** speciale (chiamiamola `Lift`) per comunicare sia con `AutoLift` che con `MechanicalLift` (e `ModernLift` in futuro). Attraverso questa interfaccia, un allarme potrà inviare messaggi ad entrambi i tipi di ascensori senza dover conoscere le differenze tra loro.

```
public interface Lift
{
  ...
}

public class AutoLift : Lift
{
  ...
}

public class MechanicalLift : Lift
{
  ...
}
```

Successivamente, per poter comunicare con ascensori specifici attraverso l'interfaccia `Lift`, un oggetto allarme deve acquisire gli **"indirizzi"** degli oggetti ascensore (ovvero i riferimenti ad essi). Possiamo passare questi riferimenti ad es. tramite un costruttore:

```
public class Alarm
{
  private readonly IEnumerable<Lift> _lifts;

  //obtain "addresses" here
  public Alarm(IEnumerable<Lift> lifts)
  {
    //store the "addresses" for later use
    _lifts = lifts;
  }
}
```

Quindi, l'allarme può inviare tre tipi di **messaggi**: `GoToBottomFloor()`, `OpenDoor()` e `DisablePower()` a qualsiasi ascensore attraverso l'interfaccia `Lift`:

```
public interface Lift
{
  void GoToBottomFloor();
  void OpenDoor();
  void DisablePower();
}
```

e, di fatto, invia tutti questi messaggi quando viene attivato. Il metodo `Trigger()` dell'allarme è simile al seguente:

```
public void Trigger()
{
  foreach(var lift in _lifts)
  {
    lift.GoToBottomFloor();
    lift.OpenDoor();
    lift.DisablePower();
```

(continues on next page)

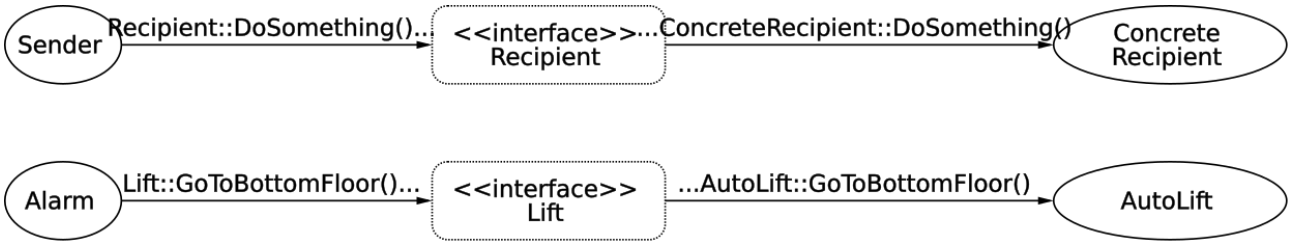
(continua dalla pagina precedente)

```
  }
}
```

A proposito, notare che l'ordine in cui i messaggi vengono inviati **è** importante. Ad esempio, se disattivassimo prima la corrente, chiedere allascensore spento di andare ovunque sarebbe impossibile. Questo è il primo segno di un **protocollo** esistente tra `Alarm` e `Lift`.

In questa comunicazione, `Alarm` è un **mittente** -- sa cosa invia (messaggi che controllano gli ascensori), sa perché (perché l'allarme viene attivato), ma non sa esattamente cosa fanno i destinatari quando ricevono il messaggio: sa solo cosa **vuole** che facciano, ma non sa **come** lo faranno. Il resto è lasciato agli oggetti che implementano `Lift` (vale a dire, `AutoLift` e `MechanicalLift`). Sono i **destinatari** -- non sanno da chi hanno ricevuto il messaggio (a meno che non gli venga detto in qualche modo nel contenuto del messaggio -- ma anche in questo caso possono essere ingannati), ma sanno come reagire , in base a chi sono (`AutoLift` ha il suo modo specifico di reagire e anche `MechanicalLift` ha il suo). Sanno anche che tipo di messaggio hanno ricevuto (un ascensore fa una cosa diversa quando gli viene chiesto di scendere al piano inferiore rispetto a quando gli viene chiesto di aprire la porta) e qual è il contenuto del messaggio (cioè argomenti di metodo -- in questo esempio semplicistico non ce n'è nessuno).

Per illustrare che esiste questa separazione tra un mittente e un destinatario, dirò che potremmo anche scrivere un'implementazione di un'interfaccia `Lift` che ignori semplicemente i messaggi ricevuti da `Alarm` (o finga di aver fatto quello che è stato richiesto) e `Alarm` non se ne accorgerà nemmeno. A volte diciamo che ciò è dovuto al fatto che decidere su una reazione specifica non è responsabilità di `Alarm`.



24.2.1 Nuovi requisiti

È stato deciso che ogni volta che si verifica un malfunzionamento nell'ascensore durante l'esecuzione della procedura di emergenza di allarme, l'oggetto lift [*ascensore*] dovrebbe segnalarlo lanciando un'eccezione chiamata `LiftUnoperationalException`. Ciò influisce sia su `Alarm` che sulle implementazioni dell'interfaccia `Lift`:

1. Le implementazioni `Lift` devono sapere che quando si verifica un malfunzionamento, dovrebbero segnalarlo lanciando un'eccezione.
2. `Alarm` deve essere pronto a gestire l'eccezione lanciata dagli ascensori e agire di conseguenza (ad esempio provare comunque a mettere in sicurezza altri ascensori).

Questo è un altro esempio di **protocollo** esistente tra `Alarm` e `Lift` che deve essere rispettato da entrambe le parti. Ecco un codice d'esempio di `Alarm` che rispetta la sua parte del protocollo, ovvero gestisce le segnalazioni di malfunzionamento nel suo metodo `Trigger()`:

```
public void Trigger()
{
  foreach(var lift in _lifts)
  {
    try
    {
      lift.GoToBottomFloor();
      lift.OpenDoor();
      lift.DisablePower();
    }
    catch(LiftUnoperationalException e)
    {
      report.ThatCannotSecure(lift);
```

(continues on next page)

(continua dalla pagina precedente)

```
    }
  }
}
```

24.3 Riepilogo

Ciascuno degli oggetti nella rete può ricevere messaggi e la maggior parte di essi invia messaggi ad altri oggetti. Nei prossimi capitoli mi riferirò a un oggetto che invia un messaggio come **sender** [*mittente*] e a un oggetto che riceve un messaggio come **recipient** [*destinatario*].

Per ora può sembrare ingiustificato introdurre questa metafora di reti, protocolli, interfacce, ecc. Tuttavia, ci sono due ragioni per farlo:

- Questo è il modo in cui interpreto il modello mentale di Alan Kay's su cosa sia la programmazione object-oriented.
- Trovo utile che alcune cose debbano essere spiegate nei prossimi capitoli: come creare connessioni tra oggetti, come progettare il confine di un oggetto e come ottenere una forte componibilità.

A proposito, lesempio di questo capitolo è un po ingenuo. Innanzitutto, nel codice di produzione reale, gli ascensori dovrebbero essere gestiti in parallelo, non in sequenza. Inoltre, probabilmente utilizzerei una sorta di pattern observer per separare le istruzioni fornite per ogni ascensore dal sollevare un evento (mostrerò un esempio di utilizzo degli observer in questo modo nel prossimo capitolo). Queste due scelte, a loro volta, probabilmente mi farebbero ripensare alla gestione degli errori - c'è la possibilità che non riuscirei a farla franca semplicemente rilevando le eccezioni. Ad ogni modo, spero che la forma ingenua abbia aiutato a spiegare l'idea di protocolli e messaggi senza alzare il livello di altri argomenti.

CAPITOLO 25

Comporre una rete di oggetti

25.1 Tre domande importanti

Ora che sappiamo che esiste una rete di oggetti, che ci sono connessioni, protocolli e simili, è tempo di menzionare l'unica cosa che tralasciata: come nasce una rete di oggetti?

Questa è, ovviamente, una questione fondamentale, perché se non siamo in grado di costruire una rete, non abbiamo una rete. Inoltre, questa è una domanda un po più complicata di quanto sembri a prima vista. Per rispondere dobbiamo trovare la risposta ad altre tre domande:

1. Quando vengono composti gli oggetti (cioè quando vengono effettuate le connessioni)?
2. Come fa un oggetto ad ottenere un riferimento ad un altro nella rete (cioè come vengono effettuate le connessioni)?
3. Dove sono composti gli oggetti (cioè dove vengono effettuate le connessioni)?

A prima vista, trovare la differenza tra queste domande può essere noioso, ma la buona notizia è che costituiscono l'argomento di questo capitolo, quindi spero che questo sarà chiarito a breve.

25.2 Un'anteprima di tutte e tre le risposte

Prima di approfondire, proviamo a rispondere a queste tre domande per un semplice codice di esempio di un'applicazione console:

```
public static void Main(string[] args)
{
  var sender = new Sender(new Recipient());

  sender.Work();
}
```

Questo è un pezzo di codice che crea due oggetti e li connette, quindi dice all'oggetto `sender` di lavorare su qualcosa. Per questo codice, le risposte alle tre domande sollevate sono:

1. Quando vengono composti gli oggetti? Risposta: durante l'avvio dell'applicazione (perché il metodo `Main()` viene chiamato all'avvio dell'applicazione console).
2. Come fa un oggetto (`Sender`) a ottenere un riferimento a un altro (`Recipient`)? Risposta: il riferimento si ottiene ricevendo un `Recipient` come parametro del costruttore.

3. Dove vengono composti gli oggetti? Risposta: nel punto di ingresso dell'applicazione (metodo `Main()`)

A seconda delle circostanze, potremmo avere diversi gruppi di risposte. Inoltre, per evitare di ripensare a questo argomento ogni volta che creiamo un'applicazione, mi piace avere una serie di risposte predefinite a queste domande. Vorrei mostrare queste risposte affrontando ciascuna delle tre domande in modo approfondito, una per una, nei prossimi capitoli.

CAPITOLO 26

Quando vengono composti gli oggetti?

La risposta rapida a questa domanda è: il prima possibile. Ora, non è stato molto utile, vero? Ecco quindi una precisazione.

Molti degli oggetti che utilizziamo nelle nostre applicazioni possono essere ben presto, creati e connessi, all'avvio dell'applicazione e possono rimanere attivi fino al termine dell'esecuzione dell'applicazione. Chiamiamo questa la **parte statica** della rete.

A parte questo, c'è qualcosa che chiamerò **parte dinamica** -- gli oggetti che vengono creati, connessi e distrutti molte volte durante il ciclo di vita dell'applicazione. Ci sono almeno due ragioni per cui esiste questa parte dinamica:

1. Alcuni oggetti rappresentano richieste o azioni dell'utente che arrivano durante il runtime dell'applicazione, vengono elaborate e quindi rimossi. Questi oggetti non possono essere creati prima, ma solo non appena si verificano gli eventi che rappresentano. Inoltre, questi oggetti non vivono per tutta la durata dell'applicazione ma vengono rimossi non appena termina l'elaborazione di una richiesta. Altri oggetti rappresentano ad es. elementi nella cache che vivono per un po' di tempo e poi scadono, quindi, ancora una volta, non disponiamo di informazioni sufficienti per comporre questi oggetti in anticipo e spesso non vivono quanto l'applicazione stessa. Tali oggetti vanno e vengono, creando connessioni temporanee.
2. Esistono oggetti che hanno una durata di vita pari a quella dell'applicazione, ma la natura delle loro connessioni è temporanea. Consideriamo un esempio in cui desideriamo crittografare l'archiviazione dei dati per l'esportazione, ma a seconda delle circostanze, a volte vogliamo esportarli utilizzando un algoritmo e talvolta utilizzando un altro. Se è così, a volte potremmo invocare il metodo di crittografia in questo modo:

```
database.encryptUsing(encryption1);
```

e talvolta così:

```
database.encryptUsing(encryption2);
```

Nel primo caso, `database` e `encryption1` sono connessi solo temporaneamente, per il tempo necessario ad eseguire la crittografia. Tuttavia, nulla impedisce la creazione di questi oggetti all'avvio dell'applicazione. Lo stesso vale per la connessione di `database` e di `encryption2` - anche questa connessione è temporanea.

Date queste definizioni, è perfettamente possibile che un oggetto faccia parte sia della parte statica che di quella dinamica -- alcune delle sue connessioni possono essere anticipate, mentre altre possono essere create in seguito, ad es. quando il suo riferimento viene passato all'interno di un messaggio inviato ad un altro oggetto (cioè quando viene passato come parametro del metodo).

CAPITOLO 27

Come fa un mittente ad ottenere un riferimento a un destinatario (cioè come vengono stabilite le connessioni)?

Esistono diversi modi in cui un mittente può ottenere un riferimento a un destinatario, ognuno dei quali è utile in determinate circostanze. Questi modi sono:

1. Riceverlo come parametro del costruttore
2. Riceverlo all'interno di un messaggio (cioè come parametro del metodo)
3. Riceverlo in risposta a un messaggio (ovvero come valore di ritorno del metodo)
4. Riceverlo come observer registrato

Diamo uno sguardo più da vicino a cosa tratta ciascuno di essi e quale scegliere in quali circostanze.

27.1 Riceverlo come parametro del costruttore

Due oggetti possono essere composti passandone uno al costruttore di un altro:

```
sender = new Sender(recipient);
```

Un mittente che riceve il destinatario salva poi un riferimento ad esso in un campo privato per dopo, in questo modo:

```
private Recipient _recipient;

public Sender(Recipient recipient)
{
  _recipient = recipient;
}
```

A partire da questo punto, il `Sender` [*Mittente*] può inviare messaggi al `Recipient` [*Destinatario*] a piacimento:

```
public void DoSomething()
{
  //... other code

  _recipient.DoSomethingElse();
```

(continues on next page)

(continua dalla pagina precedente)

```
  //... other code
}
```

27.1.1 Vantaggio: "ciò che si nasconde, lo so può cambiare"

La composizione utilizzando i costruttori presenta un vantaggio significativo. Diamo un'altra occhiata a come viene creato `Sender`:

```
sender = new Sender(recipient);
```

e come viene utilizzato:

```
sender.DoSomething();
```

Notare che solo il codice che crea un `Sender` deve essere consapevole di avere accesso a un `Recipient`. Quando si tratta di invocare un metodo, questo riferimento privato è invisibile dall'esterno. Ora, ricordate quando ho descritto il principio di separare l'uso dell'oggetto dalla sua costruzione? Se seguiamo questo principio qui, ci ritroveremo con il codice che crea un `Sender` che si trova in una posizione completamente diversa rispetto al codice che lo utilizza. Pertanto, ogni codice che utilizza un `Sender` non sarà affatto consapevole dell'invio di messaggi a un `Recipient`. Esiste una massima che dice: "ciò che nascondi, puoi cambiarlo"[1] -- in questo caso particolare, se decidiamo che il `Sender` non ha bisogno di un `Recipient` per svolgere il suo lavoro, tutto ciò che dobbiamo modificare è il codice di composizione per rimuovere il `Recipient`:

```
//no need to pass a reference to Recipient anymore
new Sender();
```

e il codice che utilizza `Sender` non ha bisogno di essere modificato affatto -- sembra ancora lo stesso di prima, poiché non ha mai conosciuto `Recipient`:

```
sender.DoSomething();
```

27.1.2 Comunicazione di intenti: è richiesto il destinatario

Un altro vantaggio dell'approccio del costruttore è che consente di dichiarare esplicitamente quali sono i destinatari richiesti per un particolare mittente. Ad esempio, un `Sender` accetta un `Recipient` nel suo costruttore:

```
public Sender(Recipient recipient)
{
 //...
}
```

La firma del costruttore rende esplicito che è necessario un riferimento a un `Recipient` affinché un `Sender` funzioni correttamente -- il compilatore non consentirà la creazione di un `Sender` senza passare *qualcosa* come `Recipient`[2].

27.1.3 Dove avviene

Passare in un costruttore è un'ottima soluzione nei casi in cui vogliamo comporre un mittente con un destinatario in modo permanente (ovvero per tutta la vita di un `Sender`). Per poter fare ciò, un `Recipient` deve, ovviamente, esistere prima di un `Sender`. Un altro requisito meno ovvio per questa composizione è che un `Recipient` deve essere utilizzabile almeno finché lo è un `Sender`. Un semplice esempio di violazione di questo requisito è questo codice:

```
sender = new Sender(recipient);

recipient.Dispose(); //but sender is unaware of it
                     //and may still use recipient later:
sender.DoSomething();
```

[1] Ho ricevuto questo detto da Amir Kolsky e Scott Bain
[2] Certo, potremmo passare un `null` ma poi saremo noi a cercare guai.

In questo caso, quando diciamo a `sender` di `DoSomething()` [*FareQualcosa*], usa un destinatario che è già stato eliminato, il che potrebbe portare ad alcuni bug fastidiosi.

27.2 Riceverlo all'interno di un messaggio (cioè come parametro del metodo)

Un altro modo comune di comporre insieme oggetti è passare un oggetto come parametro della chiamata al metodo di un altro oggetto:

```
sender.DoSomethingWithHelpOf(recipient);
```

In questo caso, gli oggetti vengono spesso composti temporaneamente, solo per il tempo di esecuzione di questo singolo metodo:

```
public void DoSomethingWithHelpOf(Recipient recipient)
{
  //... perform some logic

  recipient.HelpMe();

  //... perform some logic
}
```

27.2.1 Dove avviene

Contrariamente all'approccio del costruttore, in cui un `Sender` potrebbe nascondere al suo utente il fatto di aver bisogno di un `Recipient`, in questo caso, l'utente di `Sender` è esplicitamente responsabile di fornire un `Recipient`. In altre parole, deve esserci una sorta di accoppiamento tra il codice che utilizza `Sender` e un `Recipient`. Potrebbe sembrare che questo accoppiamento sia uno svantaggio, ma conosco alcuni scenari in cui è **richiesto** che il codice che utilizza `Sender` sia in grado di fornire il proprio `Recipient` -- ci consente di utilizzare lo stesso mittente con destinatari diversi in momenti diversi (molto spesso da parti diverse del codice):

```
//in one place
sender.DoSomethingWithHelpOf(recipient);

//in another place:
sender.DoSomethingWithHelpOf(anotherRecipient);

//in yet another place:
sender.DoSomethingWithHelpOf(yetAnotherRecipient);
```

Se questa capacità non è richiesta, preferisco di gran lunga l'approccio del costruttore in quanto rimuove l'accoppiamento (allora) non necessario tra il codice che utilizza `Sender` e un `Recipient`, offrendomi maggiore flessibilità.

27.3 Riceverlo in risposta a un messaggio (ovvero come valore di ritorno del metodo)

Questo metodo di composizione degli oggetti si basa su un oggetto intermediario -- spesso un'implementazione di un pattern factory -- per fornire i destinatari su richiesta. Per semplificare le cose, userò le factory negli esempi presentati in questa sezione, anche se quello che dico è vero anche per alcuni altri pattern creazionali (inoltre, più avanti in questo capitolo tratterò in modo approfondito alcuni aspetti del pattern factory).

Per poter richiedere i destinatari ad una factory, il mittente deve prima ottenere un riferimento ad essa. Tipicamente, una factory è composta da un mittente tramite il suo costruttore (un approccio già descritto). Per esempio:

```
var sender = new Sender(recipientFactory);
```

La factory può quindi essere utilizzata dal `Sender` a piacimento per ottenere nuovi destinatari:

```
public class Sender
{
  //...

  public void DoSomething()
  {
    //ask the factory for a recipient:
    var recipient = _recipientFactory.CreateRecipient();

    //use the recipient:
    recipient.DoSomethingElse();
  }
}
```

27.3.1 Dove avviene

Trovo questo tipo di composizione utile quando è necessario un nuovo destinatario ogni volta che viene chiamato `DoSomething()`. In questo senso, potrebbe assomigliare molto al caso dell'approccio precedentemente discusso di ricevere un destinatario all'interno di un messaggio. C'è una differenza, però. Contrariamente al passaggio di un destinatario all'interno di un messaggio, in cui il codice che utilizza il `Sender` passa un `Recipient` "dall'esterno" del `Sender`, in questo approccio ci basiamo su un oggetto separato utilizzato da un `Sender` "dall'interno".

Per essere più chiari, confrontiamo i due approcci. Il passaggio del destinatario all'interno di un messaggio si presenta così:

```
//Sender gets a Recipient from the "outside":
public void DoSomething(Recipient recipient)
{
  recipient.DoSomethingElse();
}
```

e ottenuto da una factory:

```
//a factory is used "inside" Sender
//to obtain a recipient
public void DoSomething()
{
  var recipient = _factory.CreateRecipient();
  recipient.DoSomethingElse();
}
```

Quindi, nel primo esempio, la decisione su quale `Recipient` utilizzare viene presa da chiunque chiami `DoSomething()`. Nell'esempio della factory, chi chiama `DoSomething()` non sa nulla del `Recipient` e non può influenzare direttamente quale `Recipient` viene utilizzato. La factory prende questa decisione.

27.3.2 Le factory con parametri

Finora, tutte le factory che abbiamo considerato avevano metodi di creazione con elenchi di parametri vuoti, ma questo non è un requisito di alcun tipo - volevo solo rendere gli esempi semplici, quindi ho tralasciato tutto ciò che non aiutava a chiarire il mio punto. Poiché la factory rimane il decisore su quale `Recipient` viene utilizzato, può fare affidamento su alcuni parametri esterni passati al metodo di creazione per aiutarlo a prendere la decisione.

27.3.3 Non solo factory

In questa sezione abbiamo utilizzato una factory come modello, ma l'approccio per ottenere un recipient [*destinatario*] in risposta a un messaggio è più ampio. Altri tipi di oggetti che rientrano in questa categoria includono, tra gli altri: i repository, le cache, i builder, le collection[3]. Anche se sono tutti concetti importanti (da cercare sul web), non sono necessari per proseguire in questo capitolo, quindi non li esaminerò ora.

27.4 Riceverlo come observer registrato

Ciò significa passare un destinatario a un mittente **già creato** (contrariamente al passaggio come parametro del costruttore in cui il destinatario veniva passato **durante** la creazione) come parametro di un metodo che memorizza il riferimento per un uso successivo. Di solito incontro due tipi di registrazioni:

1. un metodo "setter", in cui qualcuno registra un observer chiamando qualcosa come il metodo `sender.SetRecipient(recipient)`. Onestamente, anche se è un setter, non mi piace nominarlo secondo la convenzione "setWhatever()" -- dopo Kent Beck[4] trovo questa convenzione troppo incentrata sull'implementazione invece che sullo scopo. Pertanto, scelgo nomi diversi in base al concetto di dominio modellato dal metodo di registrazione o al suo scopo. In ogni caso, questo approccio consente a un solo observer e impostandone un altro si sovrascrive a quello precedente.
2. un metodo di "addition" - in cui qualcuno registra un observer chiamando qualcosa come `sender.addRecipient(recipient)` - in questo approccio, una collection di observer deve essere mantenuta da qualche parte e il recipient registrato come observer viene semplicemente aggiunto alla collection.

Si noti che esiste una somiglianza con l'approccio "passaggio all'interno di un messaggio" -- in entrambi, un recipient [*destinatario*] viene passato all'interno di un messaggio. La differenza è che questa volta, contrariamente all'approccio "passaggio all'interno di un messaggio", il recipient passato non viene utilizzato immediatamente (e poi dimenticato), ma piuttosto viene ricordato (registrato) per un uso successivo.

Spero di riuscire a chiarire la confusione con un breve esempio.

27.4.1 Esempio

Supponiamo di avere un sensore di temperatura in grado di segnalare il suo valore medio attuale e storico a chiunque lo sottoscriva. Se nessuno si abbona, il sensore fa comunque il suo lavoro, perché deve comunque raccogliere i dati per calcolare un valore medio basato sullo storico nel caso qualcuno si abboni in seguito.

Possiamo modellare questo comportamento utilizzando un pattern observer e consentire agli observer di registrarsi nell'implementazione del sensore. Se non è registrato alcun observer, i valori non vengono riportati (in altre parole, non è necessario un observer registrato per il funzionamento dell'oggetto, ma se ce n'è uno, può trarre vantaggio dai report). A questo scopo, facciamo dipendere il nostro sensore da un'interfaccia chiamata `TemperatureObserver` che potrebbe essere implementata da varie classi di observer concrete. La dichiarazione dell'interfaccia è simile alla seguente:

```
public interface TemperatureObserver
{
  void NotifyOn(
    Temperature currentValue,
    Temperature meanValue);
}
```

Ora siamo pronti per esaminare l'implementazione del sensore di temperatura stesso e il modo in cui utilizza l'interfaccia `TemperatureObserver`. Diciamo che la classe che rappresenta il sensore si chiama `TemperatureSensor`. Parte della sua definizione potrebbe assomigliare a questa:

```
public class TemperatureSensor
{
  private TemperatureObserver _observer
    = new NullObserver(); //ignores reported values
```

(continues on next page)

[3] Se non avete mai utilizzato collection prima e non si è un copy-editor, probabilmente si sta leggendo il libro sbagliato :-)
[4] Kent Beck, Implementation Patterns

(continua dalla pagina precedente)

```
  private Temperature _meanValue
    = Temperature.Celsius(0);

  // + maybe more fields related to storing historical data

  public void Run()
  {
    while(/* needs to run */)
    {
      var currentValue = /* get current value somehow */;
      _meanValue = /* update mean value somehow */;

      _observer.NotifyOn(currentValue, _meanValue);

      WaitUntilTheNextMeasurementTime();
    }
  }
}
```

Come si vede, per default, il sensore non riporta i suoi valori da nessuna parte (`NullObserver`), che è un valore di default sicuro (utilizzare un `null` come valore di default causerebbe delle eccezioni o ci costringerebbe a inserire un check sul null all'interno del metodo `Run()`). Abbiamo già visto questi "oggetti null"[5] un paio di volte (ad esempio nel capitolo precedente, quando abbiamo introdotto la classe `NoAlarm`) -- `NullObserver` è solo un'altra incarnazione di questo pattern.

27.4.2 Registrare gli observer

Tuttavia, vogliamo essere in grado di fornire il nostro observer un giorno, quando inizieremo a preoccuparci dei valori misurati e calcolati (il fatto che abbiamo "iniziato a preoccuparci" può essere indicato alla nostra applicazione, ad esempio da un pacchetto di networking o da un evento dall'interfaccia utente). Ciò significa che dobbiamo avere un metodo all'interno della classe `TemperatureSensor` per sovrascrivere questo observer "non fare nulla" di default con uno personalizzato **dopo** la creazione dell'istanza `TemperatureSensor`. Come ho detto, non mi piace la convenzione "SetXYZ()", quindi chiamerò il metodo di registrazione `FromNowOnReportTo()` e renderò l'observer un argomento. Ecco le parti rilevanti della classe `TemperatureSensor`:

```
public class TemperatureSensor
{
  private TemperatureObserver _observer
    = new NullObserver(); //ignores reported values

  //... ... ...

  public void FromNowOnReportTo(TemperatureObserver observer)
  {
    _observer = observer;
  }

  //... ... ...
}
```

Questo ci consente di sovrascrivere l'observer corrente con uno nuovo qualora avessimo bisogno di farlo. Notare che, come ho già detto, questo è il luogo in cui l'approccio della registrazione differisce dall'approccio "passaggio all'interno di un messaggio", in cui riceviamo anche un recipient [*destinatario*] in un messaggio, ma per un utilizzo immediato. Qui non utilizziamo il recipient (ovvero l'observer) quando lo otteniamo, ma lo salviamo per un uso successivo.

[5] Questo pattern ha un nome e il nome è... Null Object (sorpresa!). Questo pattern si può approfondire su http://www.cs.oberlin.edu/~jwalker/nullObjPattern/ e su http://www.cs.oberlin.edu/~jwalker/refs/woolf.ps (un documento un po' vecchio)

27.4.3 Comunicazione di intenti: dipendenza facoltativa

Consentire la registrazione dei destinatari dopo la creazione di un mittente è un modo per dire: "il destinatario è facoltativo -- se ce n'è uno, va bene, altrimenti farò il mio lavoro senza di esso". Per favore, non utilizzare questo tipo di meccanismo per i destinatari **richiesti** -- questi dovrebbero essere tutti passati attraverso un costruttore, rendendo più difficile la creazione di oggetti non validi che sono solo parzialmente pronti a funzionare.

Esaminiamo un esempio di una classe che:

- accetta un recipient [*destinatario*] nel suo costruttore,
- consente di registrare un recipient come observer,
- accetta un recipient per una singola chiamata al metodo

Questo esempio è annotato con commenti che riassumono ciò che dicono questi tre approcci:

```
public class Sender
{
  //"I will not work without a Recipient1"
  public Sender(Recipient1 recipient1) {...}

  //"I will do fine without Recipient2 but you
  //can overwrite the default here if you are
  //interested in being notified about something
  //or want to customize my default behavior"
  public void Register(Recipient2 recipient2) {...}

  //"I need a recipient3 only here and you get to choose
  //what object to give me each time you invoke
  //this method on me"
  public void DoSomethingWith(Recipient3 recipient3) {...}
}
```

27.4.4 Più di un observer

Ora, l'API dell'observer a cui abbiamo appena dato un'occhiata ci dà la possibilità di avere un singolo observer alla volta. Quando registriamo un nuovo observer, il riferimento a quello vecchio viene sovrascritto. Questo non è molto utile nel nostro contesto, vero? Con i sensori reali, spesso desideriamo che riportino le loro misure in più punti (ad esempio, vogliamo che le misure vengano stampate sullo schermo, salvate in un database e utilizzate come parte di calcoli più complessi). Ciò si può ottenere in due modi.

Il primo modo sarebbe semplicemente tenere una raccolta di observer nel sensore e aggiungere a questa raccolta ogni volta che viene registrato un nuovo osservatore:

```
private IList<TemperatureObserver> _observers
  = new List<TemperatureObserver>();

public void FromNowOnReportTo(TemperatureObserver observer)
{
  _observers.Add(observer);
}
```

In tal caso, il reporting significherebbe ripetere lelenco degli observer:

```
...
foreach(var observer in _observers)
{
  observer.NotifyOn(currentValue, meanValue);
}
...
```

L'approccio mostrato sopra colloca la politica di notifica agli observer all'interno del sensore. Molte volte questo potrebbe bastare. Tuttavia, il sensore è accoppiato alle risposte almeno alle seguenti domande:

- In quale ordine informiamo gli observer? Nell'esempio sopra, li informiamo secondo l'ordine di registrazione.
- Come gestiamo gli errori (ad esempio uno degli observer lancia un'eccezione) - smettiamo di notificare altri osservatori, o si esegue il log dell'errore e si continua, o magari facciamo qualcos'altro? Nell'esempio sopra, ci fermiamo sul primo observer che lancia un'eccezione e rilancia l'eccezione. Forse non è l'approccio migliore per il nostro caso?
- Il nostro modello di notifica è sincrono o asincrono? Nell'esempio sopra, stiamo utilizzando un ciclo `for` sincrono.

Possiamo ottenere un po' più di flessibilità estraendo la logica di notifica in un observer separato che riceverà una notifica e la passerà ad altri observer. Possiamo chiamarlo "un observer in broadcasting". L'implementazione di un tale observer potrebbe assomigliare a questa:

```
public class BroadcastingObserver
  : TemperatureObserver,
    TemperatureObservable //I'll explain it in a second
{
  private IList<TemperatureObserver> _observers
    = new List<TemperatureObserver>();

  public void FromNowOnReportTo(TemperatureObserver observer)
  {
    _observers.Add(observer);
  }

  public void NotifyOn(
    Temperature currentValue,
    Temperature meanValue)
  {
    foreach(var observer in _observers)
    {
      observer.NotifyOn(currentValue, meanValue);
    }
  }
}
```

Questo `BroadcastingObserver` potrebbe essere istanziato e registrato in questo modo:

```
//instantiation:
var broadcastingObserver
  = new BroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);

...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new DisplayingObserver())
...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new StoringObserver());
...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new CalculatingObserver());
```

Con questo design gli altri observer si registrano presso l'observer in broadcasting. Tuttavia, non hanno realmente bisogno di sapere con chi si stanno registrando - per nasconderlo, ho introdotto un'interfaccia speciale chiamata `TemperatureObservable`, che ha il metodo `FromNowOnReportTo()`:

```
public interface TemperatureObservable
{
  public void FromNowOnReportTo(TemperatureObserver observer);
}
```

In questo modo, il codice che registra un observer non ha bisogno di sapere quale sia l'oggetto concreto osservabile.

Lulteriore vantaggio di modellare il broadcasting come observer è che ci consentirebbe di modificare la politica di broadcasting senza toccare né il codice del sensore né gli altri observer. Ad esempio, potremmo sostituire il nostro observer basato su coclo `for` con qualcosa come `ParallelBroadcastingObserver` che notificherebbe a ciascuno dei suoi observerin modo asincrono anziché sequenziale. L'unica cosa che dovremmo cambiare è l'oggetto observer registrato con un sensore. Quindi invece di:

```
//instantiation:
var broadcastingObserver
  = new BroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);
```

Avremmo

```
//instantiation:
var broadcastingObserver
  = new ParallelBroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);
```

e il resto del codice rimarrebbe invariato. Questo perché il sensore implementa:

- L'interfaccia `TemperatureObserver`, da cui dipende il sensore,
- L'interfaccia `TemperatureObservable` da cui dipende il codice che registra gli observer.

Ad ogni modo, come ho detto, usa la registrazione delle istanze in modo molto saggio e solo ce n'è specificamente bisogno. Inoltre, se lo si usa, valutare in che modo consentire la modifica degli observer in fase di esecuzione influisce sugli scenari multithreading. Questo perché una raccolta di observer potrebbe potenzialmente essere modificata da due thread contemporaneamente.

CAPITOLO 28

Dove vengono composti gli oggetti?

Ok, abbiamo esaminato alcuni modi per passare un destinatario a un mittente. Lo abbiamo fatto dalla prospettiva "interna" di un mittente a cui viene assegnato un destinatario. Ciò che abbiamo tralasciato, in gran parte, è la prospettiva esterna, cioè chi dovrebbe passare il destinatario nel mittente?

Per quasi tutti gli approcci descritti nel capitolo precedente, non esiste alcuna limitazione -- si passa il destinatario da dove è necessario passarlo.

Esiste, tuttavia, un approccio più limitato e questo **passa un parametro costruttore**.

Perché? Perché cerchiamo di essere fedeli al principio di "separare la creazione degli oggetti dall'uso" e questo, a sua volta, è il risultato della nostra ricerca della componibilità.

Ad ogni modo, se un oggetto non può né utilizzare né creare un altro oggetto, dobbiamo creare oggetti speciali solo per creare altri oggetti (ci sono alcuni design pattern su come progettare tali oggetti, ma il più popolare e utile è il **factory**) o rinviare la creazione fino al punto di ingresso dell'applicazione (esiste anche un pattern per questo, chiamato **composition root**).

Quindi, abbiamo due casi da considerare. Inizierò con il secondo -- composition root.

28.1 Composition Root

Supponiamo, solo per divertimento, che stiamo creando un gioco per cellulare in cui un giocatore deve difendere un castello. Questo gioco ha due livelli. Ogni livello ha un castello da difendere. Quando riusciamo a difendere il castello abbastanza a lungo, il livello si considera completato e si passa a quello successivo. Quindi, possiamo suddividere la logica del dominio in tre classi: un `Game` che ha due `Level` e ognuno di essi che contiene un `Castle`. Ipotizziamo anche che le prime due classi violino il principio di separare l'uso dalla costruzione, cioè che un `Game` crei i propri livelli e ogni `Level` crei il proprio castello.

Una classe `Game` viene creata nel metodo `Main()` dell'applicazione:

```
public static void Main(string[] args)
{
  var game = new Game();

  game.Play();
}
```

`Game` crea crea i propri oggetti `Level` di classi specifiche che implementano l'interfaccia `Level` e li memorizza in un array:

```
public class Game
{
  private Level[] _levels = new[] {
    new Level1(), new Level2()
  };

  //some methods here that use the levels
}
```

E le implementazioni di `Level` creano i propri castelli e li assegnano a campi del tipo dell'interfaccia `Castle`:

```
public class Level1
{
  private Castle _castle = new SmallCastle();

  //some methods here that use the castle
}

public class Level2
{
  private Castle _castle = new BigCastle();

  //some methods here that use the castle
}
```

Ora, ho detto (e spero che si veda nel codice sopra) che le classi `Game`, `Level1` e `Level2` violano il principio di separare l'uso dalla costruzione. Questo non ci piace, vero? Proviamo a renderli più conformi al principio.

28.1.1 Realizzare la separazione dell'uso dalla costruzione

Innanzitutto, eseguiamo il refactoring di `Level1` e `Level2` secondo il principio spostando l'istanziazione dei relativi castelli all'esterno. Poiché l'esistenza di un castello è necessaria affinché un livello abbia un senso -- lo diremo nel codice utilizzando l'approccio di far passare un castello attraverso il costruttore di `Level`:

```
public class Level1
{
  private Castle _castle;

  //now castle is received as
  //constructor parameter
  public Level1(Castle castle)
  {
    _castle = castle;
  }

  //some methods here that use the castle
}

public class Level2
{
  private Castle _castle;

  //now castle is received as
  //constructor parameter
  public Level2(Castle castle)
  {
    _castle = castle;
```

(continues on next page)

(continua dalla pagina precedente)

```
  }

  //some methods here that use the castle
}
```

È stato facile, vero? Tuttavia, ci lascia con un problema da risolvere: se le istanziazioni dei castelli non sono più in `Level1` e in `Level2`, allora devono essere passate da chi crea i livelli. Nel nostro caso, questo ricade sulle spalle della classe `Game`:

```
public class Game
{
  private Level[] _levels = new[] {
    //now castles are created here as well:
    new Level1(new SmallCastle()),
    new Level2(new BigCastle())
  };

  //some methods here that use the levels
}
```

Ma, notare che -- questa classe soffre della stessa violazione di non separare l'uso degli oggetti dalla loro costruzione come hanno fatto i livelli. Pertanto, per rendere anche questa classe conforme al principio, dobbiamo trattarla come abbiamo fatto con le classi del livello -- spostare la creazione dei livelli all'esterno:

```
public class Game
{
  private Level[] _levels;

  //now levels are received as
  //constructor parameter
  public Game(Level[] levels)
  {
    _levels = levels;
  }

  //some methods here that use the levels
}
```

Ecco, l'abbiamo fatto, ma ancora una volta i livelli ora devono essere forniti da chi crea `Game`. Dove li mettiamo? Nel nostro caso, l'unica scelta rimasta è il metodo `Main()` della nostra applicazione, quindi è esattamente qui che creeremo tutti gli oggetti che passeremo a un `Game`:

```
public static void Main(string[] args)
{
  var game =
    new Game(
      new Level[] {
        new Level1(new SmallCastle()),
        new Level2(new BigCastle())
  });

  game.Play();
}
```

A proposito, `Level1` e `Level2` differiscono solo per i tipi di castello e questa differenza non c'è più, dato che l'abbiamo rifattorizzata, quindi possiamo renderli un'unica classe e chiamarla ad es. `TimeSurvivalLevel` (perché tale livello si considera completato quando riusciamo a difendere il nostro castello per un determinato periodo). Dopo questa mossa, ora abbiamo:

```
public static void Main(string[] args)
{
  var game =
    new Game(
      new Level[] {
        new TimeSurvivalLevel(new SmallCastle()),
        new TimeSurvivalLevel(new BigCastle())
  });

  game.Play();
}
```

Osservando il codice qui sopra, potremmo giungere ad un'altra conclusione divertente -- anche questo viola il principio di separare l'uso dalla costruzione! Per prima cosa creiamo e colleghiamo la rete di oggetti e poi inviamo il messaggio `Play()` all'oggetto `game`. Possiamo risolvere anche questo?

Direi "no", per due motivi:

1. Non cè altro posto in cui possiamo rinviare la creazione. Certo, potremmo spostare la creazione dell'oggetto `Game` e delle sue dipendenze in un oggetto separato responsabile solo della creazione (ad esempio una factory), ma ci lascerebbe comunque con la domanda: dove creiamo la factory? Naturalmente, potremmo usare un metodo statico per chiamarlo nel nostro `Main()` in questo modo: `var app = ApplicationRoot.Create()`, ma questo delegherebbe la composizione, non la estrarrebbe.
2. Il punto centrale del principio che stiamo cercando di applicare è il disaccoppiamento, cioè darci la possibilità di cambiare una cosa senza dover cambiarne unaltra. Se ci pensiamo, non ha senso disaccoppiare il punto di ingresso dell'applicazione dall'applicazione stessa, poiché questa è la parte dell'applicazione più specifica e non riutilizzabile per l'applicazione che possiamo immaginare.

Ciò che considero importante è che siamo arrivati a un punto in cui la rete di oggetti viene creata utilizzando l'approccio del costruttore e non abbiamo più spazio per rinviare la creazione della rete (in altre parole, è il più vicino possibile al punto di ingresso dell'applicazione). Questo posto è detto **composition root**.

Diciamo che la composition root è "il più vicino possibile" al punto di ingresso dell'applicazione perché potrebbero esserci diversi framework che controllano l'applicazione e non si avrà sempre il metodo `Main()` a disposizione[1].

Oltre alle invocazioni del costruttore, la composition root può contenere anche, ad esempio, registrazioni di observer (vedere approccio della registrazione nel passare i destinatari) se tali observer sono già noti a questo punto. È inoltre responsabile dell'eliminazione di tutti gli oggetti creati che richiedono l'eliminazione esplicita al termine dell'esecuzione dell'applicazione. uesto perché li crea e quindi è l'unico punto nel codice che può determinare in modo sicuro quando non sono necessari.

La composition root sopra sembra piuttosto piccola, ma si può immaginare che cresca molto nelle applicazioni più grandi. Esistono tecniche per rifattorizzare la composition root per renderla più leggibile e più pulita -- esploreremo tali tecniche in un capitolo dedicato.

28.2 Factory

In precedenza ho descritto come non sia sempre possibile far passare tutto attraverso il costruttore. Uno degli approcci di cui abbiamo discusso che possiamo utilizzare in questi casi è **una factory**.

Quando in precedenza abbiamo parlato di factory, ci siamo concentrati sul fatto che siano solo una fonte di oggetti. Questa volta daremo uno sguardo molto più approfondito a cosa sono le factory e quali sono i loro vantaggi.

Ma prima, diamo un'occhiata all'esempio di una factory che emerge nel codice che non la utilizzava, come mera conseguenza del tentativo di seguire il principio di separare l'uso degli oggetti dalla loro costruzione.

[1] Per i dettagli, consultare "Dependency Injection in .NET" di Mark Seemann.

28.2.1 Factory emergente -- esempio

Si consideri il seguente codice che riceve un frame proveniente dalla rete (come dati grezzi), poi lo inserisce in un oggetto, lo convalida e lo applica al sistema:

```
public class MessageInbound
{
  //...initialization code here...

  public void Handle(Frame frame)
  {
    // determine the type of message
    // and wrap it with an object
    ChangeMessage change = null;
    if(frame.Type == FrameTypes.Update)
    {
      change = new UpdateRequest(frame);
    }
    else if(frame.Type == FrameTypes.Insert)
    {
      change = new InsertRequest(frame);
    }
    else
    {
      throw
        new InvalidRequestException(frame.Type);
    }

    change.ValidateUsing(_validationRules);
    _system.Apply(change);
  }
}
```

Notare che questo codice viola il principio di separare l'uso dalla costruzione. `change` viene prima creato, a seconda del tipo di frame, e poi utilizzato (convalidato e applicato) nello stesso metodo. D'altra parte, se volessimo separare la costruzione di `change` dal suo utilizzo, dobbiamo notare che è impossibile passare un'istanza di `ChangeMessage` attraverso il costruttore `MessageInbound`, perché questo richiederebbe di creare il `ChangeMessage` prima di creare il `MessageInbound`. Raggiungere questo obiettivo è impossibile perché possiamo creare messaggi solo dopo aver conosciuto i dati del frame che `MessageInbound` riceve.

Pertanto, la nostra scelta è quella di creare un oggetto speciale in cui sposteremo la creazione di nuovi messaggi. Questo produrrebbe nuove istanze quando richiesto, da qui il nome **factory** [*fabbrica*]. La stessa factory può essere passata attraverso un costruttore poiché non richiede l'esistenza di un frame -- ne ha bisogno solo quando gli viene chiesto di creare un messaggio.

Sapendo questo, possiamo rifattorizzare il codice sopra come segue:

```
public class MessageInbound
{
  private readonly
    MessageFactory _messageFactory;
  private readonly
    ValidationRules _validationRules;
  private readonly
    ProcessingSystem _system;

  public MessageInbound(
    //this is the factory:
    MessageFactory messageFactory,
```

(continues on next page)

(continua dalla pagina precedente)

```
    ValidationRules validationRules,
    ProcessingSystem system)
  {
    _messageFactory = messageFactory;
    _validationRules = validationRules;
    _system = system;
  }

  public void Handle(Frame frame)
  {
    var change = _messageFactory.CreateFrom(frame);
    change.ValidateUsing(_validationRules);
    _system.Apply(change);
  }
}
```

In questo modo abbiamo separato la costruzione del messaggio dal suo utilizzo.

A proposito, notare che abbiamo estratto non solo un singolo costruttore, ma l'intera logica di creazione dell'oggetto. Adesso è nella factory:

```
public class InboundMessageFactory
 : MessageFactory
{
  ChangeMessage CreateFrom(Frame frame)
  {
    if(frame.Type == FrameTypes.Update)
    {
      return new UpdateRequest(frame);
    }
    else if(frame.Type == FrameTypes.Insert)
    {
      return new InsertRequest(frame);
    }
    else
    {
      throw
        new InvalidRequestException(frame.Type);
    }
  }
}
```

E questo è tutto. Adesso abbiamo una factory e il modo in cui siamo arrivati a questo punto è stato cercando di aderire al principio di separare l'uso dalla costruzione.

Ora che abbiamo finito con l'esempio, siamo pronti per qualche spiegazione più generale sulle factory.

28.2.2 Motivi per utilizzare le factory

Come dimostrato nell'esempio, le factory sono oggetti responsabili della creazione di altri oggetti. Sono utilizzate per ottenere la separazione della costruzione degli oggetti dal loro utilizzo. Sono utili per creare oggetti che vivono meno degli oggetti che li utilizzano. Una durata così breve deriva dal fatto che non tutto il contesto necessario per creare un oggetto è noto in anticipo (vale a dire finché un utente non immette le credenziali, non saremmo in grado di creare un oggetto che rappresenti il suo account). Passiamo la parte del contesto che conosciamo in anticipo (un cosiddetto **contesto globale**) nella factory tramite il suo costruttore e forniamo il resto che diventa disponibile in seguito (il cosiddetto **contesto locale**) sotto forma di parametri del metodo della factory quando diventa disponibile:

```
var factory = new Factory(globalContextKnownUpFront);

//... some time later:
factory.CreateInstance(localContext);
```

Un altro caso di utilizzo di una factory è quando dobbiamo creare un nuovo oggetto ogni volta che viene effettuata una richiesta (viene ricevuto un messaggio dalla rete o qualcuno fa clic su un pulsante):

```
var factory = new Factory(globalContext);

//...

//we need a fresh instance
factory.CreateInstance();

//...

//we need another fresh instance
factory.CreateInstance();
```

Nell'esempio precedente vengono create due istanze indipendenti, anche se entrambe sono create in modo identico (non esiste un contesto locale che le differenzi).

Entrambi questi motivi erano presenti nell'esempio dell'ultimo capitolo:

1. Non siamo riusciti a creare un `ChangeMessage` prima di conoscere il `Frame` effettivo.
2. Per ogni `Frame` ricevuto, dovevamo creare una nuova istanza di `ChangeMessage`.

28.2.3 La più semplice factory

L'esempio più semplice possibile di un oggetto factory è qualcosa del genere:

```
public class MyMessageFactory
{
  public MyMessage CreateMyMessage()
  {
    return new MyMessage();
  }
}
```

Anche in questa forma primitiva la factory ha già un certo valore (ad esempio possiamo rendere `MyMessage` un tipo astratto e restituire istanze delle sue sottoclassi dalla factory, e l'unico posto interessato dal cambiamento è la factory stessa[2]). Più spesso però, quando si parla di factory semplici, penso a qualcosa del genere:

```
//Let's assume MessageFactory
//and Message are interfaces
public class XmlMessageFactory : MessageFactory
{
  public Message CreateSessionInitialization()
  {
    return new XmlSessionInitialization();
  }
}
```

Notare le due cose che la factory nel secondo esempio ha e che quella nel primo esempio non aveva:

- implementa un'interfaccia (viene introdotto un livello di indirezione)

[2] A. Shalloway et al., Essential Skills For The Agile Developer.

- il suo metodo `CreateSessionInitialization()` dichiara che un tipo restituito è un'interfaccia (viene introdotto un altro livello di riferimento indiretto)

Pertanto, abbiamo introdotto due ulteriori livelli di indirezione. Per poter utilizzare le factory in modo efficace, è necessario capire perché e come questi livelli di indirezione sono utili, soprattutto quando si parla con persone, che spesso non comprendono i vantaggi dell'utilizzo delle factory, "perché abbiamo già l'operatore `new` per creare oggetti". Il punto è che nascondendo (incapsulando) determinate informazioni otteniamo una maggiore flessibilità:

28.2.4 Le factory consentono di creare oggetti in modo polimorfico (incapsulamento del tipo)

Ogni volta che invochiamo un operatore `new`, dobbiamo mettergli accanto il nome di un tipo concreto:

```
new List<int>(); //OK!
new IList<int>(); //won't compile...
```

Ciò significa che ogni volta che cambiamo idea e invece di usare `List<int>()` vogliamo usare un oggetto di un'altra classe (ad esempio `SortedList<int>()`), dobbiamo cambiare il codice del delete del vecchio nome del tipo e inserire il nuovo nome del tipo o fornire una sorta di condizione (`if-else`). Entrambe le opzioni presentano degli svantaggi:

- cambiare il nome del tipo richiede una modifica del codice nella classe che chiama il costruttore ogni volta che cambiamo idea, legandoci di fatto ad un'unica implementazione,
- Le istruzioni condizionali ci richiedono di conoscere in anticipo tutte le possibili sottoclassi e la nostra classe manca dell'estensibilità di cui spesso abbiamo bisogno.

Le factory consentono di far fronte a queste carenze. Poiché otteniamo oggetti da una factory invocando un metodo, non dicendo esplicitamente quale classe vogliamo istanziare, possiamo sfruttare il polimorfismo, ovvero la nostra factory potrebbe avere un metodo come questo:

```
IList<int> CreateContainerForData() {...}
```

che restituisce qualsiasi istanza di una classe reale che implementa `IList<int>` (ad esempio, `List<int>`):

```
public IList<int> /* return type is interface */
CreateContainerForData()
{
  return new List<int>(); /* instance of concrete class */
}
```

Naturalmente, non ha molto senso che il tipo restituito dalla factory sia una classe di libreria o un'interfaccia come nell'esempio precedente (piuttosto usiamo le factory per creare istanze delle nostre classi), ma si è capito, no?

In ogni caso, è tipico che un tipo restituito dichiarato di una factory sia un'interfaccia o, nel peggiore dei casi, una classe astratta. Ciò significa che chi utilizza la factory, sa solo che riceve un oggetto di una classe che implementa un'interfaccia o deriva da una classe astratta. Ma non sa esattamente di che tipo *concreto* si tratta. Pertanto, una factory può restituire oggetti di tipo diverso in momenti diversi, in base ad alcune regole che solo lei conosce.

È ora di guardare un esempio più realistico di come applicarlo. Diciamo di avere una factory di messaggi come questa:

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    if(rawData.IsSessionInit())
    {
      return new SessionInit(rawData);
    }
    else if(rawData.IsSessionEnd())
    {
```

(continues on next page)

(continua dalla pagina precedente)

```
      return new SessionEnd(rawData);
    }
    else if(rawData.IsSessionPayload())
    {
      return new SessionPayload(rawData);
    }
    else
    {
      throw new UnknownMessageException(rawData);
    }
  }
}
```

La factory può creare molti tipi diversi di messaggi a seconda di cosa c'è all'interno dei dati grezzi, ma dal punto di vista dell'utente della factory questo è irrilevante. Tutto ciò che sa è che riceve un `Message`, quindi (e il resto del codice che opera sui messaggi nell'intera applicazione) può essere scritto come logica di uso generale, senza contenere "casi speciali" dipendenti dal tipo di messaggio:

```
var message = _messageFactory.NewInstanceFrom(rawData);
message.ValidateUsing(_primitiveValidations);
message.ApplyTo(_sessions);
```

Notare che non è necessario modificare questo codice nel caso in cui desideriamo aggiungere un nuovo tipo di messaggio compatibile con il flusso esistente di elaborazione dei messaggi[3]. L'unico posto che dobbiamo modificare in questo caso è la factory. Ad esempio, si immagini di aver deciso di aggiungere un messaggio di aggiornamento della sessione. La factory modificata sarebbe simile a questa:

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    if(rawData.IsSessionInit())
    {
      return new SessionInit(rawData);
    }
    else if(rawData.IsSessionEnd())
    {
      return new SessionEnd(rawData);
    }
    else if(rawData.IsSessionPayload())
    {
      return new SessionPayload(rawData);
    }
    else if(rawData.IsSessionRefresh())
    {
      //new message type!
      return new SessionRefresh(rawData);
    }
    else
    {
      throw new UnknownMessageException(rawData);
    }
  }
}
```

[3] sebbene sia necessario cambiare quando cambia la regola "prima convalida, poi applica alle sessioni".

e il resto del codice potrebbe rimanere intatto.

Usare la factory per nascondere il vero tipo di messaggio restituito semplifica la manutenzione del codice, perché ci sono meno posti nel codice influenzati dall'aggiunta di nuovi tipi di messaggi al sistema o dalla rimozione di quelli esistenti (nel nostro esempio, nel caso in cui lo facciamo non è più necessario avviare una sessione)[4] -- la factory lo nasconde e il resto dell'applicazione è codificato rispetto allo scenario generale.

L'esempio sopra ha dimostrato come una factory possa nascondere che molte classi possono svolgere lo stesso ruolo (cioè messaggi diversi potrebbero svolgere il ruolo di un `Message`), ma possiamo anche usare le factory per nascondere che la stessa classe gioca molti ruoli. Un oggetto della stessa classe può essere restituito da un metodo factory diverso, ogni volta come un'interfaccia diversa e i client non possono accedere ai metodi implementati da altre interfacce. An object of the same class can be returned from different factory method, each time as a different interface and clients cannot access the methods it implements from other interfaces.

28.2.5 La factory sono esse stesse polimorfiche (incapsulamento delle regole)

Un altro vantaggio delle factory rispetto alle chiamate al costruttore inline è che se una factory riceve un oggetto questo può essere passato come interfaccia, il che ci consente di utilizzare un'altra factory che implementa la stessa interfaccia al suo posto tramite polimorfismo. Ciò consente di sostituire la regola utilizzata per creare oggetti con un'altra, sostituendo un'implementazione di factory con un'altra.

Torniamo all'esempio della sezione precedente, in cui avevamo una `Version1ProtocolMessageFactory` che poteva creare diversi tipi di messaggi in base ad alcuni flag impostati sui dati grezzi (ad esempio `IsSessionInit()`, `IsSessionEnd()` ecc.). Immaginiamo di aver deciso che non ci piace più questa versione. Il motivo è che avere così tanti flag booleani diversi è troppo complicato, poiché con un progetto del genere rischiamo di ricevere un messaggio in cui due o più flag sono impostati a true (ad esempio qualcuno potrebbe inviare un messaggio che indica che si tratta sia di un'inizializzazione della sessione che di fine della sessione). Supportare tali casi (ad esempio convalidando e rifiutando tali messaggi) richiede un lavoro aggiuntivo nel codice. Vogliamo migliorarlo prima che sempre più clienti inizino a utilizzare il protocollo. Pertanto, viene concepita una nuova versione del protocollo -- una versione 2. Questa versione, invece di utilizzare diversi flag, utilizza un enum (chiamato `MessageTypes`) per specificare il tipo di messaggio:

```
public enum MessageTypes
{
  SessionInit,
  SessionEnd,
  SessionPayload,
  SessionRefresh
}
```

pertanto, invece di interrogare flag diversi, la versione 2 consente di interrogare un singolo valore che definisce il tipo di messaggio.

Sfortunatamente, per mantenere la compatibilità con alcuni client, entrambe le versioni del protocollo devono essere supportate, ciascuna versione ospitata su un endpoint separato. L'idea è che quando tutti i client migreranno alla nuova versione, quella vecchia verrà ritirata.

Prima di introdurre la versione 2, la "composition root" aveva un codice simile a questo:

```
var controller = new MessagingApi(new Version1ProtocolMessageFactory());
//...
controller.HostApi(); //start listening to messages
```

dove `MessagingApi` ha un costruttore che accetta l'interfaccia `MessageFactory`:

```
public MessagingApi(MessageFactory messageFactory)
{
  _messageFactory = messageFactory;
}
```

e qualche codice generale per la gestione dei messaggi:

[4] Notare che questa è un'applicazione della linea guida della "Gang of Four": "incapsulare ciò che varia".

```
var message = _messageFactory.NewInstanceFrom(rawData);
message.ValidateUsing(_primitiveValidations);
message.ApplyTo(_sessions);
```

Questa logica deve rimanere la stessa in entrambe le versioni del protocollo. Come possiamo ottenere questo risultato senza duplicare questo codice per ogni versione?

La soluzione è creare un'altra message factory, ovvero un'altra classe che implementa l'interfaccia `MessageFactory`. Chiamiamola `Version2ProtocolMessageFactory` e implementiamola in questo modo:

```
//note that now it is a version 2 protocol factory
public class Version2ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    switch(rawData.GetMessageType())
    {
      case MessageTypes.SessionInit:
        return new SessionInit(rawData);
      case MessageTypes.SessionEnd:
        return new SessionEnd(rawData);
      case MessageTypes.SessionPayload:
        return new SessionPayload(rawData);
      case MessageTypes.SessionRefresh:
        return new SessionRefresh(rawData);
      default:
        throw new UnknownMessageException(rawData);
    }
  }
}
```

Notare che questa factory può restituire oggetti delle stesse classi della factory della versione 1, ma prende la decisione utilizzando il valore ottenuto dal metodo `GetMessageType()` invece di fare affidamento sui flag.

Avere questa factory ci consente di creare un'istanza `MessagingApi` che funziona con il protocollo della versione 1:

```
new MessagingApi(new Version1ProtocolMessageFactory());
```

o il protocollo della versione 2:

```
new MessagingApi(new Version2ProtocolMessageFactory());
```

e, poiché per il momento dobbiamo supportare entrambe le versioni, la nostra "composition root" avrà questo codice da qualche parte[5]:

```
var v1Controller = new MessagingApi(new Version1ProtocolMessageFactory());
var v2Controller = new MessagingApi(new Version2ProtocolMessageFactory());
//...
v1Controller.HostApi(); //start listening to messages
v2Controller.HostApi(); //start listening to messages
```

Notare che la classe `MessagingApi` stessa non ha bisogno di modifiche. Poiché dipende dall'interfaccia `MessageFactory`, tutto quello che dovevamo fare era fornire un oggetto factory diverso che prendesse la sua decisione in modo diverso.

Questo esempio mostra qualcosa che mi piace chiamare "incapsulamento di regole". La logica all'interno della factory è una regola su come, quando e quali oggetti creare. Pertanto, se facciamo in modo che la nostra factory implementi un'interfaccia e facciamo in modo che altri oggetti dipendano solo da questa interfaccia, saremo in grado di cambiare le

[5] Le due versioni dell'API sarebbero probabilmente ospitate su URL o porte diverse. In uno scenario reale, questi diversi valori dovrebbero probabilmente essere passati anche come parametri del costruttore.

regole di creazione degli oggetti fornendo un'altra factory senza dover modificare questi oggetti (come nel nostro caso in cui non abbiamo dovuto modificare la classe `MessagingApi`).

28.2.6 Le factory possono nascondere alcune delle dipendenze degli oggetti creati (incapsulamento del contesto globale)

Consideriamo un altro esempio semplice. Abbiamo un'applicazione che, ancora una volta, può elaborare i messaggi. Una delle cose che vengono fatte con questi messaggi è salvarli in un database e un'altra è la validazione. L'elaborazione del messaggio è, come negli esempi precedenti, gestita da una classe `MessageProcessing`, che, questa volta, non utilizza alcuna factory, ma crea i messaggi in base ai dati del frame stesso. Diamo un'occhiata a questa classe:

```
public class MessageProcessing
{
  private DataDestination _database;
  private ValidationRules _validation;

  public MessageProcessing(
    DataDestination database,
    ValidationRules validation)
  {
    _database = database;
    _validation = validation;
  }

  public void ApplyTo(MessageData data)
  {
    //note this creation:
    var message =
      new Message(data, _database, _validation);

    message.Validate();
    message.Persist();

    //... other actions
  }
}
```

C'è una cosa notevole nella classe `MessageProcessing`. Dipende dalle interfacce `DataDestination` e da `ValidationRules` ma non le utilizza. L'unica cosa per cui ha bisogno di queste interfacce è fornirle come parametri al costruttore di un `Message`. Man mano che il numero di parametri del costruttore di `Message` cresce, anche `MessageProcessing` dovrà essere modificato per accettare più parametri. Pertanto, la classe `MessageProcessing` viene inquinata da qualcosa di cui non ha direttamente bisogno.

Possiamo rimuovere queste dipendenze da `MessageProcessing` introducendo una factory che si occuperà di creare i messaggi al suo posto. In questo modo, dobbiamo solo passare `DataDestination` e `ValidationRules` alla factory, perché `MessageProcessing` non ne ha mai avuto bisogno per nessun motivo diverso dalla creazione di messaggi. Questa factory potrebbe assomigliare a questa:

```
public class MessageFactory
{
  private DataDestination _database;
  private ValidationRules _validation;

  public MessageFactory(
    DataDestination database,
    ValidationRules validation)
  {
    _database = database;
```

(continues on next page)

(continua dalla pagina precedente)

```
    _validation = validation;
  }

  //clients only need to pass data here:
  public Message CreateFrom(MessageData data)
  {
    return
      new Message(data, _database, _validation);
  }
}
```

Ora, notiamo che la creazione dei messaggi è stata spostata nella factory, insieme alle dipendenze necessarie a questo scopo. `MessageProcessing` non ha più bisogno di avere queste dipendenze e può rimanere più fedele al suo vero scopo:

```
public class MessageProcessing
{
  private MessageFactory _factory;

  //now we depend on the factory only:
  public MessageProcessing(
    MessageFactory factory)
  {
    _factory = factory;
  }

  public void ApplyTo(MessageData data)
  {
    //no need to pass database and validation
    //since they already are inside the factory:
    var message = _factory.CreateFrom(data);

    message.Validate();
    message.Persist();

    //... other actions
  }
}
```

Quindi, invece delle interfacce `DataDestination` e `ValidationRules`, `MessageProcessing` dipende solo dalla factory. Questo potrebbe non sembrare un compromesso molto interessante (togliere due dipendenze e introdurne una), ma si noti che ogni volta che `MessageFactory` necessita di un'altra dipendenza simile alle due esistenti, la factory è tutto ciò che si dovrà cambiare. `MessageProcessing` rimarrà intatta e ancora accoppiato solo alla factory.

L'ultima cosa che voglio menzionare è che non tutte le dipendenze possono essere nascoste all'interno di una factory. Notare che la factory deve ancora ricevere `MessageData` da chiunque stia richiedendo un `Message`, perché `MessageData` non è disponibile quando viene creata la factory. Ci si potrebbe ricordare che chiamo tali dipendenze un **contesto locale** (perché è specifico per un singolo utilizzo di una factory e passato da dove viene chiamato il metodo di creazione della factory). D'altra parte, ciò che una factory accetta attraverso il suo costruttore può essere chiamato un **contesto globale** (perché è lo stesso per tutta la vita della factory). Usando questa terminologia, il contesto locale non può essere nascosto agli utenti della factory, ma quello globale sì. Grazie a questo, le classi che utilizzano la factory non hanno bisogno di conoscere il contesto globale e possono rimanere più pulite, impegnate in meno cose e più concentrate.

28.2.7 Le factory aumentano la leggibilità e rivelano l'intenzione (incapsulamento della terminologia)

Supponiamo di scrivere un gioco di ruolo d'azione composto da molti livelli di gioco (da non confondere con i livelli di esperienza). I giocatori possono iniziare una nuova partita o continuarne una salvata. Quando scelgono di iniziare una nuova partita, vengono immediatamente portati al primo livello con l'inventario vuoto e senza abilità. Altrimenti, quando

scelgono di continuare un vecchio gioco, devono selezionare un file con uno stato salvato (quindi il livello di gioco, le abilità e l'inventario vengono caricati dal file). Pertanto, nel nostro gioco abbiamo due flussi di lavoro separati che terminano con l'invocazione di due metodi diversi: `OnNewGame()` per una nuova modalità di gioco e `OnContinue()` per riprendere una partita salvata:

```
public void OnNewGame()
{
  //...
}

public void OnContinue(PathToFile savedGameFilePath)
{
  //...
}
```

In ciascuno di questi metodi, dobbiamo in qualche modo assemblare un'istanza della classe `Game`. Il costruttore di `Game` consente di comporlo con un livello iniziale, l'inventario del personaggio e una serie di abilità che il personaggio può utilizzare:

```
public class FantasyGame : Game
{
  public FantasyGame(
      Level startingLevel,
      Inventory inventory,
      Skills skills)
  {
  }
}
```

Nel nostro codice non esiste una classe speciale per il "nuovo gioco" o per il "gioco ripreso". Un nuovo gioco è semplicemente un gioco che inizia dal primo livello con un inventario vuoto e senza abilità:

```
var newGame = new FantasyGame(
  new FirstLevel(),
  new BackpackInventory(),
  new KnightSkills());
```

In altre parole, il concetto di "nuovo gioco" è espresso da una composizione di oggetti piuttosto che da un'unica classe, chiamata ad es. `NewGame`.

Allo stesso modo, quando vogliamo creare l'oggetto di game che rappresenti una partita ripresa, lo facciamo in questo modo:

```
try
{
  saveFile.Open();

  var loadedGame = new FantasyGame(
    saveFile.LoadLevel(),
    saveFile.LoadInventory(),
    saveFile.LoadSkills());
}
finally
{
  saveFile.Close();
}
```

Anche in questo caso il concetto di partita ripresa è rappresentato da una composizione anziché da una singola classe, proprio come nel caso di nuova partita. D'altra parte, i concetti di "gioco nuovo" e "gioco ripreso" fanno parte del dominio, quindi dobbiamo esplicitarli in qualche modo altrimenti perdiamo leggibilità.

Uno dei modi per farlo è utilizzare una factory[6]. Possiamo creare una tale factory e inserirvi due metodi: uno per creare un nuovo gioco, un altro per creare un gioco ripreso. Il codice della factory potrebbe assomigliare a questo:

```
public class FantasyGameFactory : GameFactory
{
  public Game NewGame()
  {
    return new FantasyGame(
      new FirstLevel(),
      new BackpackInventory(),
      new KnightSkills());
  }

  public Game GameSavedIn(PathToFile savedGameFilePath)
  {
    var saveFile = new SaveFile(savedGameFilePath);
    try
    {
      saveFile.Open();

      var loadedGame = new FantasyGame(
        saveFile.LoadLevel(),
        saveFile.LoadInventory(),
        saveFile.LoadSkills());

      return loadedGame;
    }
    finally
    {
      saveFile.Close();
    }
  }
}
```

Ora possiamo utilizzare la factory nel luogo in cui ci viene notificata la scelta dell'utente. Ricordate? Questo era il posto:

```
public void OnNewGame()
{
  //...
}

public void OnContinue(PathToFile savedGameFilePath)
{
  //...
}
```

Quando riempiamo i corpi del metodo con l'utilizzo di factory, il codice finisce così:

```
public void OnNewGame()
{
  var game = _gameFactory.NewGame();
  game.Start();
}

public void OnContinue(PathToFile savedGameFilePath)
{
  var game = _gameFactory.GameSavedIn(savedGameFilePath);
```

(continues on next page)

[6] Esistono modi più semplici, ma nessuno è così flessibile come utilizzare le factory.

(continua dalla pagina precedente)

```
  game.Start();
}
```

Notare che l'utilizzo di una factory rendere più leggibile il codice e ne rivela le intenzioni. Invece di utilizzare un insieme anonimo di oggetti connessi, i due metodi mostrati sopra richiedono l'utilizzo della terminologia del dominio (richiedendo esplicitamente `NewGame()` o `GameSavedIn(path)`). Pertanto, i concetti del dominio di "nuovo gioco" e "gioco ripreso" diventano espliciti. Ciò giustifica la prima parte del nome che ho dato a questa sezione (vale a dire "Le factory aumentano la leggibilità e rivelano l'intenzione").

C'è, tuttavia, la seconda parte del nome della sezione: "incapsulamento della terminologia" che devo spiegare. Ecco una spiegazione: notare che la factory è responsabile di sapere cosa significano esattamente i termini "nuovo gioco" e "gioco ripreso". Poiché il significato dei termini è incapsulato nella factory, possiamo cambiare questo significato in tutta l'applicazione semplicemente modificando il codice all'interno della factory. Ad esempio, possiamo dire che il nuovo gioco inizia con un inventario che non è vuoto, ma contiene una spada base e uno scudo, modificando il metodo `NewGame()` della factory in questo:

```
  public Game NewGame()
  {
    return new FantasyGame(
      new FirstLevel(),
      new BackpackInventory(
        new BasicSword(),
        new BasicShield()),
      new KnightSkills());
  }
```

Mettendo tutto insieme, le factory consentono di dare nomi ad alcune composizioni specifiche di oggetti per aumentare la leggibilità e consentono di nascondere il significato di alcuni termini del dominio per facilitarne il cambiamento in futuro perché possiamo modificare il significato del termine incapsulato cambiando il codice nei metodi della factory.

28.2.8 Le factory contribuiscono ad eliminare la ridondanza

La ridondanza nel codice significa che almeno due cose devono cambiare per lo stesso motivo e nello stesso modo[Pag. 169, 2]. Di solito viene inteso come duplicazione del codice, ma ritengo che "duplicazione concettuale" sia un termine migliore. Ad esempio, i due metodi seguenti non sono ridondanti, anche se il codice sembra duplicato (a proposito, quello che segue non è un esempio di buon codice, ma solo una semplice illustrazione):

```
public int MetersToCentimeters(int value)
{
  return value*100;
}

public int DollarsToCents(int value)
{
  return value*100;
}
```

Come ho detto, non ritengo che ciò sia ridondante, perché i due metodi rappresentano concetti diversi che cambierebbero per ragioni diverse. Anche se dovessi estrarre la "logica comune" dai due metodi, l'unico nome sensato che potrei trovare sarebbe qualcosa come `MultiplyBy100()` che, secondo me, non aggiungerebbe alcun valore.

Notare che finora abbiamo considerato quattro cose che le factory incapsulano nella creazione di oggetti:

1. Tipo
2. Regola
3. Contesto globale
4. Terminologia

Quindi, se le factory non esistessero, tutti questi concetti trapelerebbero nelle classi circostanti (abbiamo visto un esempio quando stavamo parlando dell'incapsulamento del contesto globale). Ora, non appena c'è più di una classe che necessita di creare istanze, queste cose si diffondono in tutte queste classi, creando ridondanza. In tal caso, qualsiasi modifica al modo in cui vengono create le istanze probabilmente significa una modifica a tutte le classi che necessitano di creare tali istanze.

Per fortuna, avendo una factory -- un oggetto che si occupa di creare altri oggetti e nient'altro -- possiamo riutilizzare il set di regole, il contesto globale e le decisioni relative al tipo in molte classi senza un inutile lavoro superfluo. Tutto quello che dobbiamo fare è fare riferimento alla factory e richiederle un oggetto.

Ci sono altri vantaggi con le factory, ma spero di essere riuscito a spiegare perché le considero un concetto davvero vantaggioso a un costo ragionevolmente basso.

28.3 Riepilogo

Nell'ultimo capitolo, in diversi capitoli, ho provato a mostrarvi una varietà di modi di comporre insieme gli oggetti. Nessuna paura, per la maggior parte, c'è da ricordare solo di seguire il principio di separare l'uso dalla costruzione e dovrebbe bastare.

Le regole qui delineate si applicano alla maggior parte degli oggetti nella nostra applicazione. Aspetta, ho detto la maggior parte? Non in tutto? Quindi ci sono delle eccezioni? Sì, ci sono e ne parleremo tra poco quando presenterò gli *oggetti valore* [value object], ma prima dobbiamo esaminare ulteriormente l'influenza che la componibilità ha sul nostro approccio alla progettazione object-oriented.

CAPITOLO 29

Interfacce

Alcuni oggetti sono più difficili da comporre con altri oggetti, altri sono più facili. Ovviamente, puntiamo a una maggiore componibilità. Numerosi fattori influenzano questo. Ne ho già discusso indirettamente alcuni, quindi è tempo di riassumere le cose e colmare le lacune. Questo capitolo tratterà del ruolo svolto dalle interfacce nel raggiungimento di un'elevata componibilità e il prossimo tratterà il concetto di protocollo.

29.1 Classi e interfacce

Come abbiamo detto, un sender [*mittente*] si compone con un recipient [*destinatario*] ottenendo un riferimento ad esso. Inoltre, abbiamo detto che vogliamo che i nostri mittenti siano in grado di inviare messaggi a molti destinatari diversi. Questo, ovviamente, viene fatto usando il polimorfismo.

Quindi, una delle domande che dobbiamo porci nella nostra ricerca di unelevata componibilità è: da cosa dovrebbe dipendere un mittente per poter lavorare con il maggior numero possibile di destinatari? Dovrebbe dipendere da classi o da interfacce? In altre parole, quando inseriamo un oggetto come destinatario del messaggio in questo modo:

```
public Sender(Recipient recipient)
{
  this._recipient = recipient;
}
```

Il `Recipient` dovrebbe essere una classe o un'interfaccia?

Se assumiamo che `Recipient` sia una classe, possiamo ottenere la componibilità che desideriamo derivando da essa un'altra classe e implementando metodi astratti o sovrascrivendo quelli virtuali. Tuttavia, dipendere da una classe come tipo base per un destinatario presenta i seguenti svantaggi:

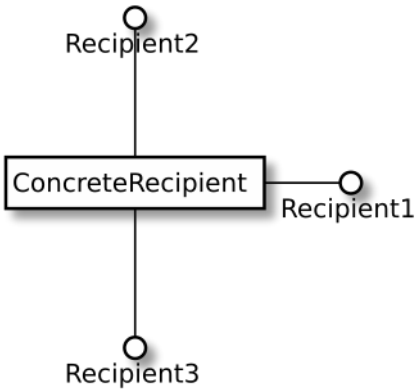
1. La classe recipient può avere delle dipendenze reali. Ad esempio, se il nostro `Recipient` dipende dallo stack "Windows Communication Foundation (WCF)", tutte le classi che dipendono direttamente da `Recipient` dipenderanno indirettamente da WCF, incluso il nostro `Sender`. Il danno maggiore del problema si ha quando una classe `Recipient` fa qualcosa come aprire una connessione di rete in un costruttore -- le sottoclassi non sono in grado di impedirlo, non importa se gli piaccia o no perché una sottoclasse deve chiamare un costruttore della superclasse.
2. Il costruttore di `Recipient` deve essere invocato da qualsiasi classe derivata, il che può rappresentare un problema più o meno grande, a seconda del tipo di parametri che il costruttore accetta e di cosa fa.
3. Nei linguaggi che supportano solo l'ereditarietà singola, la derivazione dalla classe `Recipient` utilizza l'unico slot di ereditarietà, limitando il design.

4. Dobbiamo contrassegnare tutti i metodi della classe `Recipient` come `virtual` per consentirne l'override da parte delle sottoclassi. in caso contrario, non avremo la piena componibilità. Le sottoclassi non saranno in grado di ridefinire tutti i comportamenti del `Recipient`, quindi saranno molto limitate in ciò che possono fare.

Come vedi, ci sono alcune difficoltà nell'usare le classi come "slot per la componibilità", anche se la composizione è tecnicamente possibile in questo modo. Le interfacce sono di gran lunga migliori, proprio perché non presentano gli svantaggi di cui sopra.

Si decide quindi che se un mittente vuole essere componibile con diversi destinatari, deve accettare un riferimento ad un destinatario sotto forma di riferimento di interfaccia. Possiamo dire che, essendo leggere e prive di comportamento, **le interfacce possono essere trattate come "slot" o "socket" per collegare diversi oggetti**.

Di fatto, nei diagrammi UML, un modo per rappresentare una classe che implementa un'interfaccia è disegnarla con un plug. Sembra quindi che il concetto di "interfaccia come slot per la pluggability [*collegabilità*]" non sia così insolito.



Come dai capitoli precedenti, stiamo portando all'estremo il concetto di "pluggability " [*collegabilità*] e componibilità, rendendola una delle massime priorità.

29.2 Eventi/callback e interfacce: qualche parola sui ruoli

Ho appena detto che la componibilità è "una delle massime priorità" nel nostro approccio progettuale? Wow, è una bella affermazione, vero? Sfortunatamente per me, solleva la seguente argomentazione: "Ehi, le interfacce non sono il modo più estremo per ottenere la componibilità! Che dire ad es. degli eventi in C#? O le callback supportate da altri linguaggi? Non renderebbe le classi ancora più indipendenti dal contesto e componibili, se le collegassimo tramite eventi o callback, non tramite interfacce?"

Sì, lo farebbero, ma ci priverebbero anche di un altro aspetto molto importante del nostro approccio progettuale che finora non ho menzionato esplicitamente. Questo aspetto è: i ruoli. Quando usiamo le interfacce, possiamo dire che ciascuna interfaccia rappresenta un ruolo da svolgere per un oggetto reale. Quando questi ruoli sono espliciti, aiutano a progettare e descrivere la comunicazione tra gli oggetti.

Diamo un'occhiata a un esempio di come la mancata definizione esplicita di ruoli possa togliere un po' di chiarezza alla progettazione. Questo è un metodo di esempio che invia alcuni messaggi a due destinatari tenuti come interfacce:

```
//role players:
private readonly Role1 recipient1;
private readonly Role2 recipient2;

public void SendSomethingToRecipients()
{
  recipient1.DoX();
  recipient1.DoY();
  recipient2.DoZ();
}
```

e lo confrontiamo con un effetto simile ottenuto utilizzando l'invocazione della callback:

```
//callbacks:
private readonly Action DoX;
private readonly Action DoY;
private readonly Action DoZ;

public void SendSomethingToRecipients()
{
  DoX();
  DoY();
  DoZ();
}
```

Possiamo vedere che nel secondo caso stiamo perdendo la nozione di quale messaggio appartiene a quale destinatario: ogni richiamata è autonoma dal punto di vista del mittente. Questo è un peccato perché, nel nostro approccio progettuale, vogliamo evidenziare i ruoli che ciascun destinatario gioca nella comunicazione, per renderla leggibile e logica. Inoltre, ironicamente, il disaccoppiamento tramite eventi o callback può rendere più difficile la componibilità. Questo perché i ruoli ci dicono quali insiemi di comportamenti stanno assieme e quindi devono cambiare insieme. Se ciascun comportamento viene attivato utilizzando un evento o una callback separata, ci viene imposto un sovraccarico per ricordare quali comportamenti dovrebbero essere modificati insieme e quali possono cambiare indipendentemente.

Ciò non significa che eventi o callback siano dannosi. È solo che non sono adatti a sostituire le interfacce: in realtà, il loro scopo è leggermente diverso. Usiamo eventi o callback non per dire a qualcuno di fare qualcosa, ma per indicare cosa è successo (è per questo che li chiamiamo eventi, dopotutto...). Ciò si adatta bene allo schema dellobserver di cui abbiamo già parlato nel capitolo precedente. Quindi, invece di utilizzare oggetti observer, potremmo prendere in considerazione l'utilizzo di eventi o callback (come in ogni cosa, ci sono dei compromessi per ciascuna delle soluzioni). In altre parole, eventi e callback hanno il loro utilizzo nella composizione, ma sono adatti a un caso così specifico da non poter essere trattati come un scelta di default. Il vantaggio delle interfacce è che legano insieme messaggi che rappresentano astrazioni coerenti e trasmettono ruoli nella comunicazione. Ciò migliora la leggibilità e la chiarezza.

29.3 Piccole interfacce

Ok, quindi abbiamo detto che le interfacce sono "la strada da percorrere" per raggiungere la forte componibilità a cui miriamo. Il semplice utilizzo delle interfacce ci garantisce che la componibilità sarà forte? La risposta è "no" -- sebbene l'utilizzo delle interfacce come "slot" sia un passo necessario nella giusta direzione, da solo non produce la migliore componibilità.

Una delle altre cose che dobbiamo considerare è la dimensione delle interfacce. Precisiamo una cosa ovvia a riguardo:

A parità di condizioni, le interfacce più piccole (ovvero con meno metodi) sono più facili da implementare rispetto a quelle più grandi.

L'ovvia conclusione di ciò è che se vogliamo avere una forte componibilità, i nostri "slot", cioè le interfacce, devono essere il più piccoli possibili (ma non i più piccoli -- vedere la sezione precedente su interfacce ed eventi/callback). Naturalmente, non possiamo raggiungere questo obiettivo rimuovendo ciecamente i metodi dalle interfacce, perché ciò guasterebbe le classi che utilizzano questi metodi, ad es. quando qualcuno utilizza un'implementazione dell'interfaccia come questa:

```
public void Process(Recipient recipient)
{
  recipient.DoSomething();
  recipient.DoSomethingElse();
}
```

È impossibile rimuovere uno dei metodi dall'interfaccia `Recipient` poiché causerebbe un errore di compilazione indicante che stiamo tentando di utilizzare un metodo che non esiste.

Allora, cosa facciamo quindi? Cerchiamo di separare gruppi di metodi utilizzati da mittenti diversi e di spostarli su interfacce separate in modo che ciascun mittente abbia accesso solo ai metodi di cui ha bisogno. Dopotutto, una classe può implementare più di un'interfaccia, in questo modo:

```
public class ImplementingObject
: InterfaceForSender1,
  InterfaceForSender2,
  InterfaceForSender3
{ ... }
```

L'idea di creare un'interfaccia separata per ogni mittente invece di un'unica grande interfaccia per tutti i mittenti è nota come Principio di Segregazione dell'Interfaccia[1].

29.3.1 Un semplice esempio: separazione della lettura dalla scrittura

Supponiamo di avere una classe nella nostra applicazione che rappresenta la struttura organizzativa dell'azienda. Questa applicazione espone due API. La prima serve per le notifiche sui cambiamenti della struttura organizzativa da parte di un amministratore (in modo che la nostra classe possa aggiornare i suoi dati). La seconda riguarda le operazioni lato client sui dati organizzativi, come la lista di tutti i dipendenti. L'interfaccia per la classe della struttura organizzativa può contenere metodi utilizzati da entrambe queste API:

```
public interface
OrganizationStructure
{
  ////////////////////
  //used by administrator:
  ////////////////////

  void Make(Change change);
  //...other administrative methods

  ////////////////////
  //used by clients:
  ////////////////////

  void ListAllEmployees(
    EmployeeDestination destination);
  //...other client-side methods
}
```

Tuttavia, la gestione dell'API amministrativa viene eseguita da un codice diverso rispetto a quella dell'API lato client. Pertanto, la parte amministrativa non ha alcuna conoscenza relativa all'elenco dei dipendenti e viceversa: quella lato client non ha interesse ad apportare modifiche amministrative. Possiamo usare questa conoscenza per dividere la nostra interfaccia in due:

```
public interface
OrganizationalStructureAdminCommands
{
  void Make(Change change);
  //... other administrative methods
}

public interface
OrganizationalStructureClientCommands
{
  void ListAllEmployees(
    EmployeeDestination destination);
  //... other client-side methods
}
```

[1] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Notare che ciò non limita l'implementazione di queste interfacce: una classe reale può comunque implementarle entrambe se lo desidera:

```
public class InMemoryOrganizationalStructure
: OrganizationalStructureAdminCommands,
  OrganizationalStructureClientCommands
{
  //...
}
```

In questo approccio creiamo più interfacce (cosa che a qualcuno potrebbe non piacere), ma questo non dovrebbe disturbarci molto perché, in cambio, ogni interfaccia è più facile da implementare (perché il numero di metodi da implementare è inferiore rispetto al caso in cui di una grande interfaccia). Ciò significa che viene migliorata la componibilità, che è ciò che desideriamo di più.

Si ripaga. Ad esempio, un giorno, potremmo ottenere l'obbligo di tracciare tutte le scritture sulla struttura organizzativa (ovvero le operazioni relative all'amministrazione). In tal caso, tutto ciò che dobbiamo fare è creare una classe proxy che implementi l'interfaccia `OrganizationalStructureAdminCommands`, che avvolga i metodi della classe originale con una notifica a un observer (che può essere il "trace" richiesto o qualsiasi altra cosa ci piaccia):

```
public class NotifyingAdminCommands : OrganizationalStructureAdminCommands
{
  public NotifyingCommands(
    OrganizationalStructureAdminCommands wrapped,
    ChangeObserver observer)
  {
    _wrapped = wrapped;
    _observer = observer;
  }

  void Make(Change change)
  {
    _wrapped.Make(change);
    _observer.NotifyAbout(change);
  }

  //...other administrative methods
}
```

Notare che durante la definizione della classe precedente, abbiamo dovuto implementare solo un'interfaccia: `OrganizationalStructureAdminCommands` e abbiamo potuto ignorare l'esistenza di `OrganizationalStructureClientCommands`. Ciò è dovuto alla suddivisione dell'interfaccia che abbiamo effettuato in precedenza. Se non avessimo separato le interfacce per l'accesso di admin e client, la classe `NotifyingAdminCommands` dovrebbe implementare il metodo `ListAllEmployees` (e altri) e delegarlo all'istanza wrapped [*impacchettata*] originale. Questo non è difficile, ma è un lavoro inutile. Dividere l'interfaccia in due più piccole ci ha risparmiato questo problema.

Le interfacce dovrebbero modellare i ruoli

Nell'esempio sopra, abbiamo diviso l'interfaccia più grande in due più piccole, in realtà esponendo che gli oggetti della classe `InMemoryOrganizationalStructure` possono svolgere due ruoli.

Considerare i ruoli è un altro modo efficace per separare le interfacce. Ad esempio, nella struttura organizzativa menzionata sopra, potremmo avere oggetti della classe `Employee`, ma ciò non significa che questa classe debba implementare un'interfaccia chiamata `IEmployee` o `EmployeeIfc` o qualcosa del genere. Onestamente, questa è una situazione da cui potremmo iniziare, quando non abbiamo ancora idee migliori, ma da cui vorremmo uscire il prima possibile attraverso il refactoring. Ciò che vorremmo fare appena possibile è riconoscere ruoli validi. Nel nostro esempio, dal punto di vista della struttura, il dipendente potrebbe svolgere un ruolo di `Node`. Se ha un genitore (ad esempio un'unità organizzativa) a cui appartiene, dal suo punto di vista potrebbe svolgere il ruolo di `ChildUnit`. Allo stesso modo, se nella struttura sono presenti figli (es. dipendenti da lui gestiti), potrà essere considerato il loro `Parent` o `DirectSupervisor`. Tutti questi ruoli dovrebbero essere modellati utilizzando le interfacce implementate dalla classe `Employee`:

```
public class Employee : Node, ChildUnit, DirectSupervisor
{
 //...
```

e a ciascuna di queste interfacce dovrebbero essere forniti solo i metodi necessari dal punto di vista degli oggetti che interagiscono con un ruolo modellato con questa interfaccia.

Le interfacce dovrebbero dipendere da astrazioni, non da dettagli di implementazione

È forte la tentazione di pensare che ogni interfaccia sia per definizione un'astrazione. Credo il contrario -- mentre le interfacce astraggono il tipo concreto della classe che lo implementa, potrebbero comunque contenere altre cose non astratte che sono dettagli di implementazione. Diamo un'occhiata alla seguente interfaccia:

```
public interface Basket
{
  void WriteTo(SqlConnection sqlConnection);
  bool IsAllowedToEditBy(SecurityPrincipal user);
}
```

Si vedono gli argomenti di questi metodi? `SqlConnection` è un oggetto della libreria per interfacciarsi direttamente con un database SQL Server, quindi è una dipendenza molto concreta. `SecurityPrincipal` è una delle classi principali della libreria di autorizzazioni di .NET che funziona con un database di utenti su un sistema locale o in Active Directory. Quindi, ancora una volta, una dipendenza molto concreta. Con dipendenze del genere, sarà molto difficile scrivere altre implementazioni di questa interfaccia, perché saremo costretti a trascinarci dietro dipendenze concrete e per lo più non saremo in grado di aggirare il problema se vogliamo qualcosa di diverso. Pertanto, possiamo dire che questi tipi concreti menzionati sono dettagli di implementazione esposti nell'interfaccia. Pertanto, questa interfaccia è un'astrazione fallita. È essenziale astrarre questi dettagli di implementazione, ad es. come questo:

```
public interface Basket
{
  void WriteTo(ProductOutput output);
  bool IsAllowedToEditBy(BasketOwner user);
}
```

Questo è meglio. Ad esempio, poiché `ProductOutput` è un'astrazione di livello superiore (molto probabilmente un'interfaccia, come abbiamo discusso in precedenza), nessuna implementazione del metodo `WriteTo` deve essere legata a un particolare tipo di archiviazione. Ciò significa che abbiamo più libertà di sviluppare diverse implementazioni di questo metodo. Inoltre, ogni implementazione del metodo `WriteTo` è più utile in quanto è riutilizzabile con diversi tipi di `ProductOutput`.

Un altro esempio potrebbe essere un'interfaccia dati, ovvero un'interfaccia solo con getter e setter. Guardando questo esempio:

```
public interface Employee
{
  HumanName Name { get; set; }
  HumanAge  Age { get; set; }
  Address Address { get; set; }
  Money Pay { get; set; }
  EmploymentStatus EmploymentStatus { get; set; }
}
```

in quanti modi diversi possiamo implementare tale interfaccia? Non molti -- l'unica domanda a cui possiamo rispondere in modo diverso nelle diverse implementazioni di `Employee` è: "qual è l'archiviazione dei dati?". Tutto tranne questa domanda viene esposto, rendendola un'astrazione molto povera. Questo è simile a quello con cui Johnny e Benjamin stavano combattendo nel sistema dei salari quando volevano introdurre un altro tipo di dipendente -- un dipendente [contractor]. Quindi, molto probabilmente, un'astrazione migliore sarebbe qualcosa del genere:

```
public interface Employee
{
  void Sign(Document document);
  void Send(PayrollReport payrollReport);
  void Fire();
  void GiveRaiseBy(Percentage percentage);
}
```

Quindi la regola generale è: rendere le interfacce vere e proprie astrazioni estraendo da esse i dettagli dell'implementazione. Solo allora si sarà liberi di creare diverse implementazioni dell'interfaccia che non siano vincolate da dipendenze non volute o di cui non ce n'è bisogno.

CAPITOLO 30

Protocolli

Già sappiamo che gli oggetti sono collegati (composti) insieme e comunicano attraverso interfacce, proprio come in una rete IP. C'è un'altra somiglianza, altrettanto importante. Sono i *protocolli*. In questa sezione esamineremo i protocolli tra gli oggetti e la loro posizione nel nostro approccio progettuale.

30.1 I protocolli esistono

Non voglio introdurre alcuna definizione scientifica, quindi stabiliamo semplicemente che i protocolli sono insiemi di regole su come gli oggetti comunicano tra loro.

Veramente? Ci sono delle regole? Non è sufficiente che gli oggetti possano essere composti insieme tramite interfacce, come ho spiegato nelle sezioni precedenti? Ebbene no, non basta e facciamo un breve esempio.

Immaginiamo una classe `Sender` che, in uno dei suoi metodi, chieda a `Recipient` (supponiamo che `Recipient` sia un'interfaccia) di estrarre il codice di stato da un qualche tipo di oggetto di risposta e prende una decisione in base a quel codice se notificare o meno ad un observer di un errore:

```
if(recipient.ExtractStatusCodeFrom(response) == -1)
{
  observer.NotifyErrorOccured();
}
```

Questo design è un po semplicistico ma non importa. Il suo ruolo è quello di sottolineare un certo punto. Chiunque sia il `recipient`, è previsto che segnali un errore restituendo il valore `-1`. In caso contrario, il `Sender` (che controlla esplicitamente questo valore) non sarà in grado di reagire adeguatamente alla situazione di errore. Allo stesso modo, se non è presente alcun errore, il destinatario non deve segnalarlo restituendo `-1`, perché in tal caso il `Sender` lo riconoscerà erroneamente come un errore. Quindi, ad esempio, questa implementazione di `Recipient`, sebbene implementi l'interfaccia richiesta da `Sender`, è sbagliata, perché non si comporta come `Sender` si aspetta:

```
public class WrongRecipient : Recipient
{
  public int ExtractStatusFrom(Response response)
  {
    if( /* success */ )
    {
      return -1; // but -1 is for errors!
    }
```

(continues on next page)

(continua dalla pagina precedente)

```
    else
    {
      return 1; // -1 should be used!
    }
  }
}
```

Quindi, non possiamo semplicemente scrivere qualcosa in una classe che implementa un'interfaccia, a causa di un protocollo che impone determinati vincoli sia al mittente che al destinatario.

Questo protocollo può non solo determinare i valori restituiti necessari affinché due oggetti interagiscano correttamente, ma può anche determinare i tipi di eccezioni lanciate o l'ordine delle chiamate ai metodi. Ad esempio, chiunque utilizzi un qualche tipo di oggetto connessione immaginerebbe il seguente modo di utilizzare la connessione: prima aprirla, poi farne qualcosa e chiuderla una volta terminato, ad es.

```
connection.Open();
connection.Send(data);
connection.Close();
```

Supponendo che la `connection` di cui sopra sia un'implementazione dell'interfaccia `Connection`, se dovessimo implementarla in questo modo:

```
public class WrongConnection : Connection
{
  public void Open()
  {
    // imagine implementation
    // for *closing* the connection is here!!
  }

  public void Close()
  {
    // imagine implementation for
    // *opening* the connection is here!!
  }
}
```

allora si compilerebbe bene ma fallirebbe gravemente una volta eseguito. Questo perché il comportamento sarebbe contrario al protocollo impostato tra l'astrazione `Connection` e il suo utilizzatore. Tutte le implementazioni di `Connection` devono seguire questo protocollo.

Quindi, ancora una volta, ci sono regole che limitano il modo in cui due oggetti possono comunicare. Sia il mittente che il destinatario di un messaggio devono rispettare le regole, altrimenti non saranno in grado di lavorare insieme.

La buona notizia è che la maggior parte delle volte, *noi* siamo quelli che progettano questi protocolli, insieme alle interfacce, progettandoli in modo che siano più facili o più difficili da seguire attraverso diverse implementazioni di un'interfaccia. Naturalmente, siamo con tutto il cuore per la parte "più facile".

30.2 Stabilità del protocollo

Ricordate l'ultima storia di Johnny e Benjamin quando hanno dovuto apportare una modifica al design per aggiungere un altro tipo di dipendenti ([contractor]) all'applicazione? Per fare ciò, hanno dovuto modificare le interfacce esistenti e aggiungerne di nuove. È stato un sacco di lavoro. Non vogliamo fare così tanto lavoro ogni volta che apportiamo una modifica, soprattutto quando introduciamo una nuova variazione di un concetto che è già presente nel nostro design (ad esempio, Johnny e Benjamin avevano già il concetto di "dipendente" e ne stavano aggiungendo una nuova variante, detta contractor).

Per raggiungere questo obiettivo, abbiamo bisogno che i protocolli siano più stabili, cioè meno inclini al cambiamento. Traendo alcune conclusioni dalle esperienze di Johnny e Benjamin, possiamo dire che hanno avuto problemi con la stabilità dei protocolli perché i protocolli erano:

1. complicati anziché semplici
2. concreti anziché astratti
3. granfi anziché piccoli

Sulla base dell'analisi dei fattori che compromettono la stabilità dei protocolli, possiamo elaborare alcune condizioni in cui questi protocolli potrebbero essere più stabili:

1. i protocolli dovrebbero essere semplici
2. i protocolli dovrebbero essere astratti
3. i protocolli dovrebbero essere logici
4. i protocolli dovrebbero essere piccoli

E alcune euristiche ci aiutano ad avvicinarci a queste qualità:

30.3 Creare messaggi che riflettano le intenzioni del mittente

I protocolli sono più semplici se sono progettati dal punto di vista dell'oggetto che invia il messaggio, non di quello che lo riceve. In altre parole, le firme dei metodi dovrebbero riflettere lintenzione dei mittenti piuttosto che le capacità dei destinatari.

Ad esempio, diamo un'occhiata a un codice per il login che utilizza un'istanza di una classe `AccessGuard`:

```
accessGuard.SetLogin(login);
accessGuard.SetPassword(password);
accessGuard.Login();
```

In questo piccolo frammento, il mittente deve inviare tre messaggi all'oggetto `accessGuard`: `SetLogin()`, `SetPassword()` e `Login()`, anche se non c'è una reale necessità di dividere la logica in tre passaggi -- vengono comunque eseguiti tutti nello stesso posto. Il creatore della classe `AccessGuard` potrebbe aver pensato che questa divisione renda la classe più "generale", ma sembra che si tratti di una "ottimizzazione prematura" che rende solo più difficile per il mittente lavorare con l'oggetto `accessGuard`. Pertanto, il protocollo più semplice dal punto di vista del mittente sarebbe:

```
accessGuard.LoginWith(login, password);
```

30.3.1 Denominare secondo l'intenzione

Un'altra lezione appresa dall'esempio precedente è: i setter (come `SetLogin` e `SetPassword` nel nostro esempio) raramente riflettono le intenzioni dei mittenti -- più spesso sono "cose" artificiali introdotte per gestire direttamente lo stato dell'oggetto. Questo potrebbe anche essere stato il motivo per cui qualcuno ha introdotto tre messaggi invece di uno -- forse la classe `AccessGuard` è stata implementata per contenere due campi (login e password) all'interno, quindi il programmatore potrebbe aver pensato che qualcuno volesse manipolarli separatamente dalla fase di accesso... In ogni caso, i setter dovrebbero essere evitati o sostituiti con qualcosa che rifletta meglio le intenzioni. Ad esempio, quando si ha a che fare con il pattern observer, non vogliamo dire: `SetObserver(screen)`, ma piuttosto qualcosa come `FromNowOnReportCurrentWeatherTo(screen)`.

Il problema della denominazione può essere riassunto con la seguente affermazione: il nome di un'interfaccia dovrebbe essere assegnato in base al *ruolo* svolto dalle sue implementazioni e i metodi dovrebbero essere nominati in base alle *responsabilità* che vogliamo che il ruolo abbia. Mi piace l'esempio che Scott Bain fa nel suo libro Emergent Design[1]: se ti chiedessi di darmi il numero della tua patente, potresti reagire diversamente a seconda che tu abbia la patente in tasca o nel portafoglio , o nella tua borsa, o a casa tua (nel qual caso dovresti chiamare qualcuno che lo legga per te). Il punto è: a me, in quanto mittente di questo messaggio "dammi il numero della tua patente", non interessa come lo ottieni. Dico `RetrieveDrivingLicenseNumber()`, non `OpenYourWalletAndReadTheNumber()`.

[1] Scott Bain, Emergent Design

Questo è importante perché se il nome rappresenta l'intenzione del mittente, il metodo non dovrà essere rinominato quando verranno create nuove classi che soddisfano questa intenzione in modo diverso.

30.4 Modellare le interazioni dopo il dominio del problema

A volte al lavoro mi viene chiesto di condurre un workshop di design. L'esempio che do spesso ai miei colleghi è quello di progettare un sistema per la prenotazione degli ordini (i clienti effettuano ordini e i corrieri del negozio possono prenotare chi può consegnare un certo ordine). La cosa che mi ha colpito le prime volte che ho partecipato a questo workshop è che, anche se l'applicazione riguardava esclusivamente gli ordini e la loro prenotazione, quasi nessuno dei partecipanti ha introdotto alcun tipo di interfaccia o classe `Order` contenente il metodo `Reserve()`. La maggior parte dei partecipanti presuppone che `Order` sia una struttura dati e gestisce la prenotazione aggiungendola a una "collezione di articoli prenotati" che può essere immaginata come il seguente frammento di codice:

```
// order is just a data structure,
// added to a collection
reservedOrders.Add(order)
```

Sebbene ciò raggiunga l'obiettivo in termini tecnici (ovvero che l'applicazione funzioni), il codice non riflette il dominio.

Se ruoli, responsabilità e collaborazioni tra oggetti riflettono il dominio, allora qualsiasi cambiamento naturale nel dominio lo è anche nel codice. Se così non fosse, i cambiamenti che sembrano piccoli dal punto di vista del dominio del problema finirebbero per toccare molte classi e metodi in modi molto insoliti. In altre parole, le interazioni tra gli oggetti diventano meno stabili (che è proprio ciò che vogliamo evitare).

D'altra parte, supponiamo di aver modellato la progettazione in base al dominio e di aver introdotto un ruolo `Order` appropriato. Allora, la logica per prenotare un ordine potrebbe assomigliare a questa:

```
order.ReserveBy(deliverer);
```

Notare che questa linea è stabile quanto il dominio stesso. Deve cambiare, ad es. quando gli ordini non vengono più prenotati o qualcuno diverso dai corrieri inizia a prenotare gli ordini. Quindi, direi che la stabilità di questa piccola interazione è dannatamente alta.

Anche nei casi in cui la comprensione del dominio si evolve e cambia rapidamente, la stabilità del dominio, sebbene non così elevata come al solito, è comunque una delle più alte che il mondo intorno a noi ha da offrire.

30.4.1 Un altro esempio

Supponiamo di avere un codice per la gestione degli allarmi. Quando viene attivato un allarme, tutti i cancelli vengono chiusi, le sirene vengono attivate e viene inviato un messaggio alle forze speciali con la massima priorità affinché arrivino e arrestino l'intruso. Qualsiasi errore in questa procedura porta all'interruzione dell'energia elettrica nell'edificio. Se questo flusso di lavoro è codificato in questo modo:

```
try
{
  gates.CloseAll();
  sirens.TurnOn();
  specialForces.NotifyWith(Priority.High);
}
catch(SecurityFailure failure)
{
  powerSystem.TurnOffBecauseOf(failure);
}
```

Allora il rischio che questo codice cambi per ragioni diverse dal cambiamento del funzionamento del dominio (ad esempio non chiudiamo più i cancelli ma attiviamo invece le pistole laser) è piccolo. Pertanto, le interazioni che utilizzano astrazioni e metodi che esprimono direttamente le regole di dominio sono più stabili.

Quindi, per riassumere -- se un progetto riflette il dominio, è più facile prevedere come un cambiamento delle regole del dominio influenzerà il progetto. Ciò contribuisce alla manutenibilità e alla stabilità delle interazioni e del design nel suo insieme.

30.5 Ai destinatari del messaggio dovrebbe essere detto cosa fare, invece di chiedergli informazioni

Diciamo che stiamo pagando un'imposta annuale sul reddito ogni anno e siamo troppo occupati (cioè abbiamo troppe responsabilità) per farlo da soli. Pertanto, assumiamo un ragioniere esperto per calcolare e pagare le tasse. È un esperto nel pagare le tasse, sa come calcolare tutto, dove presentarlo, ecc. ma c'è una cosa che non conosce -- il contesto. In altre parole, non sa quale banca stiamo utilizzando o cosa abbiamo guadagnato quest'anno per cui dobbiamo pagare le tasse. Questo è qualcosa che dobbiamo dargli.

Ecco l'accordo tra noi e l'esperto fiscale riassunto in una tabella:

Chi?	Bisogni	Può fornire
Noi	L'imposta pagata	contesto (banca, documenti sul reddito)
Esperto fiscale	contesto (banca, documenti sul reddito)	Il servizio di pagamento dell'imposta

Siamo noi ad assumere l'esperto e noi ad avviare la trattativa, quindi dobbiamo fornire il contesto, come mostrato nella tabella sopra. Se dovessimo modellare questo accordo come un'interazione tra due oggetti, potrebbe ad es. assomigliare a questo:

```
taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  ourBank);
```

Un giorno, la nostra amica Joan ci dice che anche lei ha bisogno di un esperto fiscale. Siamo soddisfatti della persona che abbiamo assunto, quindi lo consigliamo a Joan. Ha i suoi documenti di reddito, ma sono funzionalmente simili ai nostri, solo con numeri diversi qua e là e forse qualche formattazione diversa. Inoltre, Joan utilizza una banca diversa, ma oggigiorno interagire con qualsiasi banca è quasi identico. Pertanto, il nostro esperto fiscale sa come gestire la sua richiesta. Se modelliamo questo come interazione tra oggetti, potrebbe assomigliare a questo:

```
taxExpert.PayAnnualIncomeTax(
  joansIncomeDocuments,
  joansBank);
```

Pertanto, quando interagisce con Joan, l'esperto fiscale può comunque utilizzare le sue capacità per calcolare e pagare le tasse allo stesso modo del nostro caso. Questo perché le sue competenze sono indipendenti dal contesto.

Un altro giorno decidiamo che non siamo più contenti del nostro esperto fiscale, quindi decidiamo di stringere un accordo con uno nuovo. Per fortuna, non abbiamo bisogno di sapere come svolgono il loro lavoro gli esperti fiscali: diciamo loro semplicemente di farlo, così possiamo interagire con quello nuovo proprio come con quello precedente:

```
//this is the new tax expert,
//but no change to the way we talk to him:

taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  ourBank);
```

Questo piccolo esempio non va preso alla lettera. Le interazioni sociali sono molto più complicate e complesse di quelle che fanno normalmente gli oggetti. Ma, con questo, spero di essere riuscito a illustrare un aspetto importante dello stile di comunicazione preferito nella progettazione object-oriented: l'euristica *Dire non chiedere*.

Dire non chiedere significa che ogni oggetto, in quanto esperto nel proprio lavoro, lo gestisce bene delegando altre responsabilità ad altri oggetti che sono esperti nei rispettivi lavori e fornendo loro tutto il contesto di cui hanno bisogno per il compito assegnato come parametri dei messaggi inviati loro.

Questo può essere illustrato con un pattern di codice generico:

```
recipient.DoSomethingForMe(allTheContextYouNeedToKnow);
```

In questo modo si ottiene un doppio vantaggio:

1. Il nostro destinatario (ad esempio il `taxExpert` dell'esempio) può essere utilizzato da altri mittenti (ad esempio pagare le tasse di Joan) senza bisogno di cambiare. Tutto ciò di cui ha bisogno è un contesto diverso passato all'interno di un costruttore e dei messaggi.
2. Noi, come mittenti, possiamo facilmente utilizzare destinatari diversi (ad esempio diversi esperti fiscali che svolgono il compito assegnato in modo diverso) senza imparare come interagire con ciascuno di essi.

Se si guarda la cosa, così come la banca e i documenti sono un contesto per lesperto fiscale, lesperto fiscale è un contesto per noi. Pertanto, potremmo dire che *un progetto che segue il principio Dire Non Chiedere crea classi indipendenti dal contesto*.

Ciò ha uninfluenza molto profonda sulla stabilità dei protocolli. Per quanto gli oggetti siano indipendenti dal contesto, essi (e le loro interazioni) non hanno bisogno di cambiare quando cambia il contesto.

Ancora una volta, citando Scott Bain, ciò che si nasconde, si può cambiare. Pertanto, dire a un oggetto cosa fare richiede meno conoscenza che chiedere dati e informazioni. Usando ancora la metafora della patente di guida: potrei chiedere a un'altra persona il numero della patente di guida per assicurarmi che abbia la patente e che sia valida (controllando il numero da qualche parte). Potrei anche chiedere a un'altra persona di fornirmi le indicazioni per raggiungere il luogo in cui voglio che la prima persona guidi. Ma non è più semplice dire semplicemente "comprami del pane e del burro"? Quindi, chiunque lo chieda, ha la libertà di guidare o camminare (se conosce un buon negozio nelle vicinanze) o chiedere a un'altra persona di farlo. Non mi interessa, basta che domani mattina trovi il pane e il burro nel mio frigorifero.

Tutti questi vantaggi sono, tra l'altro, esattamente ciò a cui Johnny e Benjamin miravano durante il refactoring del sistema di gestione delle retribuzioni. Sono partiti da questo codice, dove *ponevano* molte domande agli `employee`:

```
var newSalary
  = employee.GetSalary()
  + employee.GetSalary()
  * 0.1;
employee.SetSalary(newSalary);
```

a questo progetto in cui *ha detto* a `employee` di fare il suo lavoro:

```
employee.EvaluateRaise();
```

In questo modo, sono riusciti a far interagire questo codice sia con `RegularEmployee` che con `ContractorEmployee` allo stesso modo.

Questa linea guida dovrebbe essere trattata molto, molto seriamente e applicata in modo quasi estremo. Naturalmente ci sono pochi posti in cui non si applica e ci torneremo più tardi.

Oh, quasi dimenticavo una cosa! Il contesto che stiamo passando non è necessariamente quello dei dati. È ancora più frequente passare comportamenti che dati. Ad esempio, nella nostra interazione con l'esperto fiscale:

```
taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  ourBank);
```

La classe `Bank` probabilmente non è un dato. Piuttosto, immagino che la `Bank` implementi un'interfaccia simile a questa:

```
public interface Bank
{
  void TransferMoney(
    Amount amount,
    AccountId sourceAccount,
    AccountId destinationAccount);
}
```

Quindi, questa `Bank` espone comportamenti, non dati, e segue anche lo stile Dire Non Chiedere (fa qualcosa di buono e prende tutto il contesto di cui ha bisogno dall'esterno).

30.5.1 Quando Dire Non Chiedere non si applica

Come ho detto prima, ci sono posti in cui Dire Non Chiedere non si applica. Ecco alcuni esempi che mi vengono in mente:

1. Le factory -- questi sono oggetti che producono altri oggetti per noi, quindi sono intrinsecamente "pull-based" -- viene sempre chiesto loro di consegnare oggetti.
2. Le collection -- sono semplicemente contenitori per oggetti, quindi tutto ciò che vogliamo da loro è aggiungere oggetti e recuperare oggetti (per indice, per predicato, usando una chiave, ecc.). Notare, tuttavia, che quando scriviamo una classe che racchiude una collection al suo interno, vogliamo che questa classe esponga un'interfaccia modellata in modo Dire Non Chiedere.
3. Data source, come i database -- ancora una volta, si tratta di archivi di dati, quindi è più probabile che dovremo chiedere questi dati per ottenerli.
4. Alcune API a cui si accede tramite la rete -- sebbene sia utile utilizzare quanto più possibile il "Dire Non Chiedere", le API Web hanno una limitazione -- è difficile o impossibile trasmettere comportamenti come oggetti polimorfici attraverso di esse. Di solito possiamo solo trasmettere dati.
5. Le cosiddette "fluent API", chiamate anche "linguaggi interni specifici del dominio"[2]

Anche nei casi in cui otteniamo altri oggetti da una chiamata al metodo, vogliamo essere in grado di applicare il "Dire Non Chiedere" a questi altri oggetti. Ad esempio, vogliamo evitare la seguente catena di chiamate:

```
Radio radio = radioRepository().GetRadio(12);
var userName = radio.GetUsers().First().GetName();
primaryUsersList.Add(userName);
```

In questo modo rendiamo la comunicazione legata ai seguenti presupposti:

1. La radio ha molti utenti
2. La radio deve avere almeno un utente
3. Ogni utente deve avere un nome
4. Il nome non è nullo

D'altra parte, consideriamo questa implementazione:

```
Radio radio = radioRepository().GetRadio(12);
radio.AddPrimaryUserNameTo(primaryUsersList);
```

Non presenta nessuno dei punti deboli dell'esempio precedente. Pertanto, è più stabile di fronte al cambiamento.

30.6 La maggior parte dei getter dovrebbero essere rimossi, i valori restituiti dovrebbero essere evitati

La linea guida summenzionata di "Tell Don't Ask" ha un'implicazione pratica di sbarazzarsi di (quasi) tutti i getter. Abbiamo detto che ogni oggetto dovrebbe attenersi al proprio lavoro e dire agli altri oggetti di fare il proprio lavoro, passando loro il contesto, no? Se è così, allora perché dovremmo "ottenere" (get) qualcosa da altri oggetti?

Per me, l'idea di "no getter" all'inizio era molto estrema, ma in breve tempo ho imparato che questo è in realtà il modo in cui dovrei scrivere codice object-oriented. Ho iniziato a imparare a programmare utilizzando linguaggi procedurali come il C, dove un programma era diviso in procedure o funzioni e strutture dati. Poi sono passato ai linguaggi object-oriented che avevano meccanismi di astrazione molto migliori, ma il mio stile di codifica non è cambiato molto. Avrei ancora procedure e funzioni, semplicemente suddivise in oggetti. Avrei ancora strutture dati, ma ora più astratte, ad es. oggetti con setter, getter e alcuni metodi di query.

[2] Questo argomento non rientra nell'ambito del libro, ma si può dare un'occhiata a: M. Fowler, Domain-Specific Languages, Addison-Wesley 2010

Ma quali alternative abbiamo? Bene, ho già introdotto "Tell Don't Ask", quindi dovrebbe essere nota la risposta. Tuttavia, voglio mostrare un altro esempio, questa volta specifico sui getter e i setter.

Diciamo che abbiamo un software che gestisce le sessioni utente. Una sessione è rappresentata nel codice utilizzando una classe `Session`. Vogliamo essere in grado di fare tre cose con le nostre sessioni: visualizzarle sulla GUI, inviarle attraverso la rete e renderle persistenti. Nella nostra applicazione, vogliamo che ciascuna di queste responsabilità sia gestita da una classe separata, perché riteniamo che sia positivo se non sono legate insieme.

Quindi, abbiamo bisogno di tre classi che si occupino dei dati posseduti dalla sessione. Ciò significa che ciascuna di queste classi dovrebbe in qualche modo ottenere l'accesso ai dati. Altrimenti, come possono essere questi dati, ad es. persistere? Sembra che non abbiamo scelta e dobbiamo esporli utilizzando i getter.

Naturalmente, potremmo riconsiderare la nostra scelta di creare classi separate per l'invio, la persistenza, ecc. e considerare la scelta di inserire tutta questa logica all'interno di una classe `Session`. Se lo facessimo, tuttavia, renderemmo il concetto di dominio core [*principale*] (una sessione) dipendente da un brutto insieme di librerie di terze parti (come una particolare libreria GUI), il che significherebbe che ad es. ogni volta che il concetto di visualizzazione della GUI cambia, saremo costretti ad armeggiare con il codice del dominio principale, il che è piuttosto rischioso. Inoltre, se lo facessimo, la `Session` sarebbe difficile da riutilizzare, perché in ogni posto in cui vorremmo riutilizzare questa classe, dovremmo portare con noi tutte queste pesanti librerie da cui dipende. Inoltre, non saremmo in grado ad es. di utilizzare `Session` con diverse GUI o librerie di persistenza. Quindi, ancora una volta, sembra che la nostra unica scelta (non così buona, come vedremo) sia quella di introdurre getter per le informazioni memorizzate all'interno di una sessione, in questo modo:

```
public interface Session
{
  string GetOwner();
  string GetTarget();
  DateTime GetExpiryTime();
}
```

Quindi sì, in un certo senso, abbiamo disaccoppiato `Session` da queste librerie di terze parti e potremmo anche dire che abbiamo raggiunto l'indipendenza dal contesto per quanto riguarda la stssa `Session`: ora possiamo estrarre tutti i suoi dati, ad es. in un codice GUI e visualizzarlo come tabella. La `Session` non ne sa nulla. Vediamo:

```
// Display sessions as a table on GUI
foreach(var session in sessions)
{
  var tableRow = TableRow.Create();
  tableRow.SetCellContentFor("owner", session.GetOwner());
  tableRow.SetCellContentFor("target", session.GetTarget());
  tableRow.SetCellContentFor("expiryTime", session.GetExpiryTime());
  table.Add(tableRow);
}
```

Sembra che abbiamo risolto il problema separando i dati dal contesto in cui vengono utilizzati e portando i dati in un luogo che abbia il contesto, cioè sappia cosa fare con questi dati. Siamo felici? Potremmo esserlo, a meno che non guardiamo come appaiono le altre parti: ci si ricordi che oltre a visualizzare le sessioni, vogliamo anche inviarle e renderle persistenti. La logica dell'invio è simile alla seguente:

```
//part of sending logic
foreach(var session in sessions)
{
  var message = SessionMessage.Blank();
  message.Owner = session.GetOwner();
  message.Target = session.GetTarget();
  message.ExpiryTime = session.GetExpiryTime();
  connection.Send(message);
}
```

e la logica della persistenza in questo modo:

```
//part of storing logic
foreach(var session in sessions)
{
  var record = Record.Blank();
  dataRecord.Owner = session.GetOwner();
  dataRecord.Target = session.GetTarget();
  dataRecord.ExpiryTime = session.GetExpiryTime();
  database.Save(record);
}
```

Si vede qualcosa di inquietante qui? Se no, immaginiamo cosa succede quando aggiungiamo un'altra informazione alla `Session`, ad esempio una priorità. Ora abbiamo tre posti da aggiornare e dobbiamo ricordarci di aggiornarli tutti ogni volta. Questo si chiama "ridondanza" o "cercare guai". Inoltre, la componibilità di queste tre classi è piuttosto scarsa, perché dovranno cambiare molto solo perché cambiano i dati in una sessione.

La ragione di ciò è che abbiamo reso la classe `Session` effettivamente una struttura dati. Non implementa alcun comportamento relativo al dominio, espone semplicemente i dati. Ci sono due implicazioni di questo:

1. Costringe tutti gli utenti di questa classe a definire comportamenti relativi alla sessione per conto della `Session`, il che significa che questi comportamenti sono sparsi ovunque[3]. Se si vuole apportare una modifica alla sessione, è necessario individuare tutti i comportamenti correlati e correggerli.
2. Poiché un insieme di comportamenti degli oggetti è generalmente più stabile dei suoi dati interni (ad esempio, una sessione potrebbe avere più di un obiettivo un giorno, ma avvieremo e arresteremo sempre le sessioni), ciò porta a interfacce e protocolli fragili -- certamente il contrario di ciò per cui stiamo lottando.

Peccato, questa soluzione è piuttosto pessima, ma sembra che non abbiamo più opzioni. Dovremmo semplicemente accettare che ci saranno problemi con questa implementazione e andare avanti? Per fortuna, non dobbiamo farlo. Finora, abbiamo riscontrato che le seguenti opzioni sono problematiche:

1. La classe `Session` contenente la logica di visualizzazione, archiviazione e invio, ovvero tutto il contesto necessario -- eccessivo accoppiamento con dipendenze pesanti.
2. La classe `Session` per esporre i suoi dati tramite getter, in modo da poterli inserire dove abbiamo abbastanza contesto per sapere come usarli -- la comunicazione è troppo fragile e si insinua della ridondanza (a proposito, anche questo design sarà pessimo per il multithreading, ma ne parleremo un'altra volta).

Per fortuna, abbiamo una terza alternativa, che è migliore delle due già menzionate. Possiamo semplicemente **passare** il contesto **nella** classe `Session`. "Non è questo solo un altro modo di fare ciò che abbiamo delineato al punto 1? Se passiamo il contesto, `Session` non è ancora accoppiata a questo contesto?", ci si potrebbe chiedere. La risposta è: non necessariamente perché possiamo fare in modo che la classe `Session` dipenda solo dalle interfacce invece che dalla cosa reale per renderla sufficientemente indipendente dal contesto.

Vediamo come funziona nella pratica. Innanzitutto, rimuoviamo questi getter dalla `Session` e introduciamo un nuovo metodo chiamato `DumpInto()` che prenderà un'implementazione dell'interfaccia `Destination` come parametro:

```
public interface Session
{
  void DumpInto(Destination destination);
}
```

L'implementazione di `Session`, ad es. una `RealSession` può passare tutti i campi in questa destinazione in questo modo:

```
public class RealSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    destination.AcceptOwner(this.owner);
```

(continues on next page)

[3] Questo a volte è detto Feature Envy (*invidia dei dati*). Significa che una classe è più interessata ai dati di altre classi che ai propri.

(continua dalla pagina precedente)

```
    destination.AcceptTarget(this.target);
    destination.AcceptExpiryTime(this.expiryTime);
    destination.Done();
  }

  //...
}
```

E il ciclo delle sessioni ora assomiglia a questo:

```
foreach(var session : sessions)
{
  session.DumpInto(destination);
}
```

In questo progetto, `RealSession` stesso decide quali parametri passare e in quale ordine (se è importante) -- nessuno chiede i suoi dati. Questo metodo `DumpInto()` è abbastanza generale, quindi possiamo usarlo per implementare tutti e tre i comportamenti menzionati (visualizzazione, persistenza, invio), creando un'implementazione per ciascun tipo di destinazione, ad es. per la GUI, potrebbe assomigliare a questo:

```
public class GuiDestination : Destination
{
  private TableRow _row;
  private Table _table;

  public GuiDestination(Table table, TableRow row)
  {
    _table = table;
    _row = row;
  }

  public void AcceptOwner(string owner)
  {
    _row.SetCellContentFor("owner", owner);
  }

  public void AcceptTarget(string target)
  {
    _row.SetCellContentFor("target", target);
  }

  public void AcceptExpiryTime(DateTime expiryTime)
  {
    _row.SetCellContentFor("expiryTime", expiryTime);
  }

  public void Done()
  {
    _table.Add(_row);
  }
}
```

Il protocollo è ora più stabile per quanto riguarda i consumatori dei dati di sessione. In precedenza, quando avevamo i getter nella classe `Session`:

```
public class Session
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  string GetOwner();
  string GetTarget();
  DateTime GetExpiryTime();
}
```

i getter **dovevano** restituire **qualcosa**. E se avessimo sessioni che potrebbero scadere e decidessimo di volerle ignorare quando lo fanno (ovvero non visualizzarle, archiviarle, inviarle o fare qualsiasi altra cosa con esse)? In caso di "approccio getter" visto nello snippet sopra, dovremmo aggiungere un altro getter, ad es. chiamato `IsExpired()` nella classe della sessione e ricordare di aggiornare ciascun consumatore allo stesso modo -- per controllare la scadenza prima di consumare i dati... si vede dove si sta andando, vero? D'altra parte, con l'attuale design dell'interfaccia `Session`, possiamo ad es. introdurre una funzionalità in cui le sessioni scadute non vengono elaborate affatto in un unico posto:

```
public class TimedSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    if(!IsExpired())
    {
      destination.AcceptOwner(this.owner);
      destination.AcceptTarget(this.target);
      destination.AcceptExpiryTime(this.expiryTime);
      destination.Done();
    }
  }

  //...
}
```

e non è necessario modificare nessun altro codice per farlo funzionare[4].

Un altro vantaggio di progettare/creare `Session` in modo che non restituisca nulla dai suoi metodi è che abbiamo maggiore flessibilità nell'applicazione di pattern come proxy e decorator alle implementazioni di `Session`. Ad esempio, possiamo utilizzare il pattern proxy per implementare sessioni nascoste che non vengono visualizzate/memorizzate/inviate affatto, ma allo stesso tempo si comportano come un'altra sessione in tutti gli altri casi. Un proxy di questo tipo inoltra tutti i messaggi che riceve all'oggetto `Session` originale, racchiuso, ma scarta le chiamate `DumpInto()`:

```
public class HiddenSession : Session
{
  private Session _innerSession;

  public HiddenSession(Session innerSession)
  {
    _innerSession = innerSession;
  }

  public void DoSomething()
  {
    // forward the message to wrapped instance:
    _innerSession.DoSomething();
  }

  //...
```

(continues on next page)

[4] Possiamo ulteriormente rifattorizzarlo in una macchina a stati utilizzando un pattern della Gang of Four: *State*. Ci sarebbero due stati in una macchina a stati: partito e scaduto.

(continua dalla pagina precedente)

```
  public void DumpInto(Destination destination)
  {
    // discard the message - do nothing
  }

  //...
}
```

I client di questo codice non noteranno affatto questo cambiamento. Quando non siamo obbligati a restituire nulla siamo più liberi di fare ciò che vogliamo. Ancora una volta, "Tell, Don't Ask".

30.7 I protocolli dovrebbero essere brevi e astratti

Ho già detto che le interfacce dovrebbero essere piccole e astratte, quindi non mi sto semplicemente ripetendo qui? La risposta è: esiste una differenza tra la dimensione dei protocolli e la dimensione delle interfacce. Come esempio estremo, prendiamo la seguente interfaccia:

```
public interface Interpreter
{
  public void Execute(string command);
}
```

L'interfaccia è piccola? Ovviamente! È astratta? Beh, più o meno sì. "Tell Don't Ask"? Sicuro! Ma vediamo come viene utilizzato da uno dei suoi collaboratori:

```
public void RunScript()
{
  _interpreter.Execute("cd dir1");
  _interpreter.Execute("copy *.cs ../../dir2/src");
  _interpreter.Execute("copy *.xml ../../dir2/config");
  _interpreter.Execute("cd ../../dir2/");
  _interpreter.Execute("compile *.cs");
  _interpreter.Execute("cd dir3");
  _interpreter.Execute("copy *.cs ../../dir4/src");
  _interpreter.Execute("copy *.xml ../../dir4/config");
  _interpreter.Execute("cd ../../dir4/");
  _interpreter.Execute("compile *.cs");
  _interpreter.Execute("cd dir5");
  _interpreter.Execute("copy *.cs ../../dir6/src");
  _interpreter.Execute("copy *.xml ../../dir6/config");
  _interpreter.Execute("cd ../../dir6/");
  _interpreter.Execute("compile *.cs");
}
```

Il punto è: il protocollo non è né astratto né piccolo. Pertanto, realizzare implementazioni di un'interfaccia utilizzata come tale può essere piuttosto faticoso.

30.8 Riepilogo

In questo lungo capitolo ho cercato di mostrarvi il valore, spesso sottovalutato, della progettazione di protocolli di comunicazione tra oggetti. Non sono una "cosa bella da avere", ma piuttosto una parte fondamentale dell'approccio progettuale che rende utili gli oggetti mock, come si vedrà quando finalmente ci arriveremo. Ma prima è necessario assimilare qualche altra idea di progettazione object-oriented. Prometto che darà i suoi frutti.

CAPITOLO 31

Le Classi

Abbiamo già trattato interfacce e protocolli. Nella nostra ricerca della componibilità, dobbiamo considerare anche le classi. Classi:

- implementare le interfacce (ovvero recitare i ruoli)
- comunicare tramite interfacce con altri servizi
- seguire i protocolli in questa comunicazione

Quindi, in un certo senso, ciò che è "dentro" una classe è un sottoprodotto di come gli oggetti di questa classe agiscono "all'esterno". Tuttavia, ciò non significa che non ci sia nulla da dire sulle classi stesse che contribuisca a una migliore componibilità.

31.1 Principio di Responsabilità Unica

Ho già detto che vogliamo che il nostro sistema sia una rete di oggetti componibili. Un oggetto è un granello di componibilità -- non possiamo ad es. scollegare metà di un oggetto e collegarne un'altra metà. Pertanto, una domanda valida da porsi è: quanto dovrebbe essere grande un oggetto per rendere confortevole la componibilità -- per permetterci di scollegare tutta la logica che vogliamo, lasciando il resto intatto e pronto a funzionare con i nuovi destinatari che colleghiamo?

La risposta arriva con un *Principio di Responsabilità Unica* ("Single Responsibility Principle", in breve: SRP) per le classi[1], che dice[2]:

{{keyToDo}} Il codice di una classe dovrebbe avere un solo motivo per cambiare.

È stato scritto molto su questo principio sul web, quindi non ne saprò più saggio del motore di ricerca web (la mia ricerca recente ha prodotto oltre 74mila risultati). Tuttavia credo sia utile spiegare questo principio in termini di componibilità.

Di solito, la parte difficile di questo principio è come capire "un motivo per cambiare". Robert C. Martin spiega[3] che si tratta di un'unica fonte di entropia che genera modifiche alla classe. Il che ci porta ad un altro problema, quello di definire una "fonte di entropia". Quindi penso che sia meglio fare solo un esempio.

[1] Questo principio può essere applicato anche ai metodi, ma non tratteremo questa parte, perché non è direttamente legata al concetto di componibilità e questo non è un libro di design ;-).

[2] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[3] https://stackoverflow.fogbugz.com/default.asp?W29030

31.1.1 Separazione delle responsabilità

Ricordate il codice utilizzato da Johnny e Benjamin per applicare i piani di incentivazione ai dipendenti? In caso contrario, eccolo qui (è solo un singolo metodo, non un'intera classe, ma dovrebbe essere sufficiente per le nostre esigenze):

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
    employee.Save();
  }
}
```

Quindi... quanti motivi per cambiare ha questo pezzo di codice? Se non parlassimo di "motivi per cambiare" ma semplicemente di "cambiamento", la risposta sarebbe "molti". Ad esempio, qualcuno potrebbe decidere che non stiamo più dando aumenti e la riga `employee.EvaluateRaise()` scomparirebbe. Allo stesso modo, se si decidesse di non concedere bonus, la riga `employee.EvaluateBonus()` dovrebbe essere rimossa. Quindi, ci sono senza dubbio molti modi in cui questo metodo potrebbe cambiare. Ma sarebbe per ragioni diverse? In realtà no. Il motivo in entrambi i casi sarebbe (probabilmente) che l'amministratore delegato ha approvato un nuovo piano di incentivi. Quindi, esiste una "fonte di entropia" per questi due cambiamenti, sebbene ci siano molti modi in cui il codice può cambiare. Quindi, le due modifiche avvengono per lo stesso motivo.

Ora la parte più interessante della discussione: che ne dite di salvare i dipendenti? La ragione per cambiare il modo in cui salviamo i dipendenti è la stessa che per i bonus e le retribuzioni? Ad esempio, potremmo decidere di non salvare ciascun dipendente separatamente, perché ciò causerebbe un enorme carico di prestazioni sul nostro archivio dati, ma di salvarli insieme in un unico batch dopo aver terminato l'elaborazione dell'ultimo. Ciò fa sì che il codice cambi, ad es. come questo:

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  }

  //now all employees saved once
  _repository.SaveAll(employees);
}
```

Quindi, come già intuito, il motivo di questo cambiamento è diverso da quello della modifica del piano di incentivi, quindi si tratta di una responsabilità separata e la logica per la lettura e l'archiviazione dei dipendenti dovrebbe essere separata da questa classe. Il metodo dopo la separazione e l'estrazione in una nuova classe sarebbe simile a questo:

```
public void ApplyYearlyIncentivePlanTo(IEnumerable<Employee> employees)
{
  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  }
}
```

Nell'esempio sopra, abbiamo spostato la lettura e la scrittura dei dipendenti, in modo che siano gestite da un codice diverso, quindi le responsabilità sono separate. Disponiamo ora di un codice che aderisce al Principio di Responsabilità Unica?

Possiamo, ma consideriamo questa situazione: la valutazione degli aumenti e dei bonus inizia a rallentare e, invece di farlo per tutti i dipendenti in un ciclo sequenziale `for`, preferiremmo parallelizzarlo per elaborare tutti i dipendenti contemporaneamente in un thread separato. Dopo aver applicato questa modifica, il codice potrebbe assomigliare a questo (utilizza un'API specifica di C# per il loop parallelo, ma spero di aver reso l'idea):

```
public void ApplyYearlyIncentivePlanTo(IEnumerable<Employee> employees)
{
  Parallel.ForEach(employees, employee =>
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  });
}
```

È questo un nuovo motivo per cambiare? Ovviamente è! Le decisioni sulla parallelizzazione dell'elaborazione provengono da fonti diverse rispetto alle modifiche del piano di incentivi. Quindi, potremmo dire che abbiamo incontrato un'altra responsabilità e l'abbiamo separata. Il codice che rimane nel metodo `ApplyYearlyIncentivePlanTo()` ora assomiglia a questo:

```
public void ApplyYearlyIncentivePlanTo(Employee employee)
{
  employee.EvaluateRaise();
  employee.EvaluateBonus();
}
```

Il looping, che è una responsabilità separata, è ora gestito da una classe diversa.

31.1.2 Fino a che punto andiamo?

L'esempio sopra pone alcune domande:

1. Possiamo raggiungere un punto in cui abbiamo separato tutte le responsabilità?
2. Se sì, come possiamo essere sicuri di averlo raggiunto?

La risposta alla prima domanda è: probabilmente no. Sebbene alcune ragioni per cambiare siano dettate dal buon senso e altre possano essere tratte dalla nostra esperienza di sviluppatori o dalla conoscenza del del problema, ce ne sono sempre alcune che sono inaspettate e finché non emergono non possiamo prevederle. Pertanto, la risposta alla seconda domanda è: "non c'è modo". Ciò non significa che non dovremmo cercare di separare le diverse ragioni che vediamo -- anzi. Semplicemente evitiamo di diventare troppo zelanti nel cercare di prevedere ogni possibile cambiamento.

Mi piace il confronto tra le responsabilità e il nostro utilizzo del tempo nella vita reale. Il tempo di infusione del tè nero è solitamente di circa tre-cinque minuti. Questo è quello che solitamente viene stampato sulla confezione che acquistiamo: "3 --- 5 minuti". Nessuno fornisce il tempo in secondi, perché tale granularità non è necessaria. Se i secondi facessero una notevole differenza nel processo di preparazione del tè, probabilmente ci verrebbe dato il tempo in secondi. Ma non lo fanno. Quando valutiamo le attività nell'ingegneria del software, utilizziamo anche una granularità temporale diversa a seconda della necessità[4] e la granularità diventa più fine quando raggiungiamo un punto in cui le differenze più piccole contano di più.

Allo stesso modo, un semplice programma software che stampa "hello world" sullo schermo potrebbe rientrare in un unico metodo "main" e probabilmente non lo vedremo come diverse responsabilità. Ma non appena riceviamo l'ordine di scrivere "hello world" nella lingua madre del sistema operativo attualmente in esecuzione, ottenere il testo diventa una responsabilità separata dal metterlo sullo schermo. Tutto dipende dalla granularità di cui abbiamo bisogno al momento (che, come accennato, può essere individuata dal codice o, in alcuni casi, nota in anticipo dalla nostra esperienza come sviluppatori o dalla conoscenza del dominio).

[4] A condizione che non utilizziamo una misura come gli story point.

31.1.3 Il rapporto reciproco tra "Principio di Responsabilità Unica" e componibilità

Il motivo per cui scrivo tutto questo è che le responsabilità[5] sono i veri granuli della componibilità. La componibilità degli oggetti di cui ho già parlato molto è un mezzo per raggiungere la componibilità delle responsabilità. Quindi questo è il nostro vero obiettivo. Se abbiamo due oggetti che collaborano, ciascuno con una singola responsabilità, possiamo facilmente sostituire il modo in cui la nostra applicazione raggiunge una di queste responsabilità senza toccare l'altra. Pertanto, gli oggetti conformi all'SRP [*Principio di Responsabilità Unica*] sono i più comodamente componibili e hanno le giuste dimensioni.[6].

Un buon esempio da un altro terreno di gioco in cui la singola responsabilità va di pari passo con la componibilità è UNIX. UNIX è famoso per la sua raccolta di tool a riga di comando monouso, come `ls`, `grep`, `ps`, `sed` ecc. Lo scopo unico di queste utilità insieme alla capacità della riga di comando UNIX di passare un flusso di output di un comando al flusso di input di un altro utilizzando l'operatore "|" (pipe). Ad esempio, possiamo combinare tre comandi: `ls` (elenca il contenuto di una directory), `sort` (ordina l'input passato) e `more` (visualizza comodamente sullo schermo l'input che occupa più di uno schermo) in una pipeline:

```
ls | sort | more
```

Che mostra il contenuto ordinato della directory corrente per una visualizzazione più comoda. Questa filosofia di comporre un insieme di strumenti con un unico scopo in un insieme più complesso e più utile ed è ciò che cerchiamo, solo che nello sviluppo di software object-oriented utilizziamo oggetti invece di eseguibili. Ne parleremo meglio nel prossimo capitolo.

31.2 Destinatari statici

Sebbene i campi statici nel corpo di una classe a volte possano sembrare una buona idea di "condividere" i riferimenti dei destinatari tra le sue istanze e un modo intelligente per rendere il codice più "efficiente in termini di memoria", il più delle volte danneggiano la componibilità. Diamo un'occhiata a un semplice esempio per avere un'idea di come i campi statici vincolano il nostro design.

31.2.1 Server SMTP

Immaginiamo di dover implementare un server di posta elettronica che riceva e invii messaggi SMTP[7]. Abbiamo una classe `OutboundSmtpMessage` che simboleggia i messaggi SMTP che inviamo ad altre parti. Per inviare il messaggio, dobbiamo codificarlo. Per ora, utilizziamo sempre una codifica chiamata *Quoted-Printable*, che è dichiarata in una classe separata chiamata `QuotedPrintableEncoding` e la classe `OutboundSmtpMessage` dichiara un campo privato di questo tipo:

```
public class OutboundSmtpMessage
{
  //... other code

  private Encoding _encoding = new QuotedPrintableEncoding();

  //... other code
}
```

Si noti che ogni messaggio possiede il proprio oggetto di codifica, quindi quando abbiamo, diciamo, 1.000.000 di messaggi in memoria, abbiamo anche la stessa quantità di oggetti di codifica.

[5] Si noti che sto scrivendo sulla responsabilità in termini di "Principio di Responsabilità Unica". Nella "*Responsibility-Driven Design*", responsabilità significa qualcosa di diverso. Vedere il chiarimento di Rebecca Wirfs-Brock.

[6] Si noti che sto parlando di responsabilità nel modo in cui ne parla SRP, non nel modo in cui vengono intese, ad es. Responsibility-Driven Design. Pertanto, sto parlando delle responsabilità di una classe, non delle responsabilità della sua API.

[7] SMTP sta per Simple Mail Transfer Protocol ed è un protocollo standard per l'invio e la ricezione di posta elettronica. Ulteriori informazioni su Wikipedia.

31.2.2 Ottimizzazione prematura

Un giorno notiamo che è uno spreco per ogni messaggio definire il proprio oggetto di codifica poiché una codifica è un puro algoritmo e ogni utilizzo di questa codifica non influisce in alcun modo su ulteriori usi -- quindi possiamo anche avere una singola istanza e usarla in tutti i messaggi -- non causerà alcun conflitto. Inoltre, potrebbe farci risparmiare alcuni cicli della CPU, poiché creare una codifica ogni volta che creiamo un nuovo messaggio ha il suo costo in scenari ad alta produttività.

Ma come facciamo a condividere la codifica tra tutte le istanze? Il primo pensiero -- campi statici! Un campo statico sembra adatto allo scopo poiché ci fornisce esattamente ciò che vogliamo -- un singolo oggetto condiviso tra molte istanze della sua classe dichiarante. Spinti dalla nostra (presumibilmente) eccellente idea, modifichiamo la nostra classe di messaggi `OutboundSmtpMessage` per contenere l'istanza `QuotedPrintableEncoding` come campo statico:

```
public class OutboundSmtpMessage
{
  //... other code

  private static Encoding _encoding = new QuotedPrintableEncoding();

  //... other code
}
```

Ecco, abbiamo risolto il problema! Ma le nostre mamme non ci hanno detto di non ottimizzare prematuramente? Vabbè...

31.2.3 Benvenuto, cambiamento!

Un giorno si scopre che nei nostri messaggi dobbiamo supportare non solo la codifica Quoted-Printable ma anche un'altra, chiamata *Base64*. Col nostro nostro design attuale, non possiamo farlo perché, come risultato dell'utilizzo di un campo statico, una singola codifica è condivisa tra tutti i messaggi. Pertanto, se modifichiamo la codifica per un messaggio che richiede la codifica Base64, cambierà anche la codifica per i messaggi che richiedono Quoted-Printable. In questo modo, limitiamo la componibilità con questa ottimizzazione prematura -- non possiamo comporre ciascun messaggio con la codifica che desideriamo. Tutti i messaggi utilizzano una codifica o un'altra. Una conclusione logica è che nessuna istanza di tale classe è indipendente dal contesto -- non può ottenere il proprio contesto, ma piuttosto il contesto le viene imposto.

31.2.4 E per quanto riguarda le ottimizzazioni?

Siamo condannati a tornare alla soluzione precedente per avere una codifica per ciascun messaggio? E se questo diventasse davvero un problema di prestazioni o di memoria? La nostra osservazione che non è necessario creare la stessa codifica molte volte è inutile?

Niente affatto. Possiamo ancora utilizzare questa osservazione e ottenere molti (anche se non tutti) dei vantaggi di un campo statico. Come lo facciamo? Come possiamo ottenere la condivisione degli encoding senza i vincoli di un campo statico? Bene, abbiamo già risposto a questa domanda qualche capitolo fa -- assegnare a ciascun messaggio una codifica tramite il suo costruttore. In questo modo, possiamo passare la stessa codifica a moltissime istanze `OutboundSmtpMessage`, ma se vogliamo, possiamo sempre creare un messaggio a cui è passata un'altra codifica. Utilizzando questa idea, proveremo a ottenere la condivisione delle codifiche creando una singola istanza di ciascuna codifica nella *composition root* e facendola passare a un messaggio tramite il suo costruttore.

Esaminiamo questa soluzione. Innanzitutto, dobbiamo creare una codifica per ciascuna codifica nella "composition root", in questo modo:

```
// We are in a composition root!

//...some initialization

var base64Encoding = new Base64Encoding();
var quotedPrintableEncoding = new QuotedPrintableEncoding();

//...some more initialization
```

Ok, le codifiche vengono create, ma dobbiamo ancora passarle ai messaggi. Nel nostro caso, dobbiamo creare un nuovo oggetto `OutboundSmtpMessage` nel momento in cui dobbiamo inviare un nuovo messaggio, ovvero su richiesta, quindi abbiamo bisogno di una factory per produrre gli oggetti del messaggio. Questa factory può (e deve) essere creata nella "composition root". Quando creiamo la factory, possiamo passare entrambe le codifiche al suo costruttore come contesto globale (ricordate che le factory incapsulano il contesto globale?):

```
// We are in a composition root!

//...some initialization

var messageFactory
  = new StmpMessageFactory(base64Encoding, quotedPrintableEncoding);

//...some more initialization
```

La factory stessa può essere utilizzata per la creazione di messaggi on-demand di cui abbiamo parlato. Poiché la factory riceve entrambe le codifiche tramite il suo costruttore, può memorizzarle come campi privati e passare quella appropriata all'oggetto messaggio che crea:

```
public class SmtpMessageFactory : MessageFactory
{
  private Encoding _quotedPrintable;
  private Encoding _base64;

  public SmtpMessageFactory(
    Encoding quotedPrintable,
    Encoding base64)
  {
    _quotedPrintable = quotedPrintable;
    _base64 = base64;
  }

  public Message CreateFrom(string content, MessageLanguage language)
  {
    if(language.IsLatinBased)
    {
      //each message gets the same instance of encoding:
      return new StmpMessage(content, _quotedPrintable);
    }
    else
    {
      //each message gets the same instance of encoding:
      return new StmpMessage(content, _base64);
    }
  }
}
```

Le prestazioni e il risparmio di memoria non sono esattamente così grandi come quando si utilizza un campo statico (ad esempio, ciascuna istanza `OutboundSmtpMessage` deve memorizzare un riferimento separato alla codifica ricevuta), ma rappresenta comunque un enorme miglioramento rispetto alla creazione di un oggetto di codifica separato per ogni messaggio.

31.2.5 Dove funziona la statica?

Ciò che ho scritto non significa che le cose static non abbiano la loro utilità. Lo fanno, ma questi usi sono molto specifici. Vi mostrerò uno di questi usi nei prossimi capitoli dopo aver introdotto gli *oggetti valore*.

31.3 Riepilogo

In questo capitolo ho provato a darvi qualche consiglio sulla progettazione di classi che non emergano in modo così naturale dal concetto di componibilità e interazioni come quelli descritti nei capitoli precedenti. Tuttavia, come spero di essere riuscito a dimostrare, migliorano la componibilità e sono preziosi.

CAPITOLO 32

Composizione di Oggetti come Linguaggio

Mentre la maggior parte dei capitoli precedenti parlava molto della visualizzazione della composizione degli oggetti come una rete, questo avrà una visione diversa -- quella di un linguaggio. Questi due punti di vista sono notevolmente simili e si completano a vicenda nel guidare la progettazione.

Potrebbe sorprendere il fatto che io stia paragonando la composizione degli oggetti a un linguaggio, ma, come spero si vedrà, ci sono molte somiglianze. Ci arriveremo passo dopo passo, il primo passo sarà dare una seconda occhiata alla *composition root*.

32.1 Una *composition root* più leggibile

Nel descrivere la composizione dell'oggetto e la *composition root* in particolare, ho promesso di tornare sull'argomento per rendere il codice di composizione più pulito e leggibile.

Prima di farlo, tuttavia, dobbiamo trovare una risposta a una domanda importante...

32.1.1 Perché preoccuparsi?

A questo punto si sarà stufi di come sottolineo l'importanza della componibilità. Lo faccio, però, perché credo che sia uno degli aspetti più importanti di classi ben progettate. Inoltre, ho detto che per raggiungere un'elevata componibilità di una classe, essa deve essere indipendente dal contesto. Per spiegare come raggiungere questa indipendenza, ho introdotto il principio di separare l'uso dell'oggetto dalla costruzione, relegando la parte di costruzione in luoghi specializzati nel codice. Ho anche detto che si può contribuire molto a questa qualità rendendo astratte le interfacce e i protocolli e facendo in modo che espongano il minor numero possibile di dettagli dell'implementazione.

Tutto ciò ha però il suo costo. La ricerca di un'elevata indipendenza dal contesto ci toglie la capacità di osservare una singola classe e determinarne il contesto semplicemente leggendone il codice. Questa classe conosce ben poco del contesto in cui opera. Ad esempio, qualche capitolo fa abbiamo trattato delle sessioni di dumping e ho mostrato che tale metodo di dump può essere implementato in questo modo:

```
public class RealSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    destination.AcceptOwner(this.owner);
    destination.AcceptTarget(this.target);
```

(continues on next page)

(continua dalla pagina precedente)

```
    destination.AcceptExpiryTime(this.expiryTime);
    destination.Done();
  }

  //...
}
```

Qui, la sessione sa che qualunque sia la destinazione, `Destination` accetta proprietario, i target e la scadenza e deve essere informata quando tutte le informazioni le vengono passate. Tuttavia, leggendo questo codice, non possiamo dire dove porta la destinazione, poiché `Destination` è un'interfaccia che astrae i dettagli. È un ruolo che può essere svolto da un file, una connessione di rete, uno schermo della console o un widget della GUI. L'indipendenza dal contesto consente la componibilità.

D'altra parte, per quanto le classi e le interfacce indipendenti dal contesto siano importanti, lo è anche il comportamento dell'applicazione nel suo insieme. Non ho detto che l'obiettivo della componibilità è poter modificare più facilmente il comportamento dell'applicazione? Ma come possiamo decidere consapevolmente di modificare il comportamento dell'applicazione se non lo comprendiamo? E non oltre lultimo paragrafo abbiamo concluso che limitarsi a leggere una classe dopo laltra non è sufficiente. Dobbiamo avere una visione di come queste classi lavorano insieme come un sistema. Allora, dov'è il contesto generale che definisce il comportamento dell'applicazione?

Il contesto è nel codice di composizione -- il codice che collega gli oggetti, passando veri collaboratori a ciascuno di essi e mostrando come le parti connesse formano un tutto.

32.1.2 Esempio

Presumo che ci si ricordi a malapena l'esempio degli allarmi che ho fornito in uno dei primi capitoli di questa parte del libro per spiegare il cambiamento del comportamento modificando la composizione dell'oggetto. Comunque, solo per ricordarlo, abbiamo terminato con un codice simile a questo:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      new SilentAlarm("919"), //call police
      new LoudAlarm()
    )
  )
);
```

Quindi avevamo tre edifici tutti dotati di allarmi. La caratteristica interessante di questo codice era che ci potevamo leggere le impostazioni dell'allarme, ad es. la seguente parte della composizione:

```
new OfficeBuilding(
  new DayNightSwitchedAlarm(
    new SilentAlarm("222-333-444"),
    new LoudAlarm()
```

(continues on next page)

(continua dalla pagina precedente)

```
  )
),
```

significava che stavamo organizzando un edificio di uffici con un allarme che chiama il numero "222-333-444" quando viene attivato durante il giorno, ma ci sono sirene assordanti viene attivato di notte. Potremmo leggerlo direttamente dal codice di composizione, a patto di sapere cosa ha aggiunto ciascun oggetto al comportamento composito complessivo. Quindi, ancora una volta, la composizione delle parti descrive il comportamento dellinsieme. C'è, tuttavia, un'altra cosa da notare su questo pezzo di codice: descrive il comportamento senza dichiarare esplicitamente il suo flusso di controllo (`if`, `else`, `for`, ecc.). Tale descrizione è spesso chiamata *dichiarativa* -- componendo oggetti, scriviamo *cosa* vogliamo ottenere senza scrivere *come* ottenerlo -- il flusso di controllo stesso è nascosto all'interno degli oggetti.

Riassumiamo queste due conclusioni con la seguente affermazione:

{{keyInfo}} Il codice di composizione è una descrizione dichiarativa del comportamento generale della nostra applicazione.

Wow, questa è una bella affermazione, vero? Ma, come abbiamo già notato, è vero. Esiste tuttavia un problema nel trattare il codice di composizione come una descrizione complessiva dell'applicazione: la leggibilità. Anche se la composizione *è* la descrizione del sistema, non si legge in modo naturale. Vogliamo vedere la descrizione del comportamento, ma la maggior parte di ciò che vediamo è: `new`, `new`, `new`, `new`, `new`... C'è molto rumore sintattico coinvolto, specialmente nei sistemi reali, dove il codice di composizione è molto più lungo di questo piccolo esempio. Non possiamo fare qualcosa al riguardo?

32.2 Il refactoring per la leggibilità

La dichiaratività del codice di composizione va di pari passo con un approccio di definizione delle cosiddette *interfacce fluenti*. Una *interfaccia fluente* è un'API realizzata pensando alla leggibilità e alla lettura scorrevole. Di solito è dichiarativa e mirata a un dominio specifico, quindi un altro nome: *internal domain-specific languages* [*linguaggi interni specifici del dominio*], in breve: DSL.

Esistono alcuni semplici pattern per creare tali linguaggi specifici del dominio. Uno di questi che può essere applicato alla nostra situazione è chiamato *nested function*[1], che, nel nostro contesto, significa racchiudere una chiamata a `new` con un metodo più descrittivo. Potrebbe risultare confuso, ma vedremo come funziona nella pratica tra un secondo. Lo faremo passo dopo passo, quindi ci sarà molto codice ripetuto, ma si spera che si possa osservare da vicino il processo di miglioramento della leggibilità del codice di composizione.

Ok, rivediamo il codice prima di apportare qualsiasi modifica:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      new SilentAlarm("919"), //call police
      new LoudAlarm()
    )
```

(continues on next page)

[1] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

(continua dalla pagina precedente)

```
  )
);
```

Notare che abbiamo diversi posti in cui creiamo `SilentAlarm`. Spostiamo la creazione di questi oggetti in un metodo separato:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(number);
}
```

Questo passaggio può sembrare sciocco (dopo tutto, stiamo introducendo un metodo che racchiude una singola riga di codice), ma ha molto senso. Innanzitutto, ci permette di ridurre il rumore della sintassi -- quando dobbiamo creare un allarme silenzioso, non dobbiamo più dire `new`. Un altro vantaggio è che possiamo descrivere il ruolo che gioca un'istanza `SilentAlarm` nella nostra composizione (spiegherò più avanti perché lo stiamo facendo utilizzando la voce passiva).

Dopo aver sostituito ogni invocazione del costruttore `SilentAlarm` con una chiamata a questo metodo, otteniamo:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      Calls("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      Calls("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      Calls("919"), //police number
      new LoudAlarm()
    )
  )
);
```

Successivamente, facciamo lo stesso con `LoudAlarm`, racchiudendo la sua creazione in un metodo:

```
public Alarm MakesLoudNoise()
{
  return new LoudAlarm();
}
```

e il codice di composizione dopo aver applicato questo metodo assomiglia a questo:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
```

(continues on next page)

(continua dalla pagina precedente)

```
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      Calls("919"), //police number
      MakesLoudNoise()
    )
  )
);
```

Si noti che abbiamo rimosso alcuni `new` in favore di qualcosa di più leggibile. Questo è esattamente ciò che intendevo con "ridurre il rumore della sintassi".

Ora concentriamoci un po' su questa parte:

```
  new GuardsBuilding(
    new HybridAlarm(
      Calls("919"), //police number
      MakesLoudNoise()
    )
  )
```

e proviamo ad applicare lo stesso trucco introducendo un metodo factory alla creazione di `HybridAlarm`. Ci viene sempre detto che i nomi delle classi dovrebbero essere sostantivi ed è per questo che `HybridAlarm` si chiama così. Ma non funziona bene come descrizione di ciò che fa il sistema. La sua vera funzione è quella di attivare entrambi gli allarmi quando viene attivato. Dobbiamo quindi trovare un nome migliore. Dovremmo chiamare il metodo `TriggersBothAlarms()`? No, è troppo rumore -- sappiamo già che stiamo attivando degli allarmi, quindi possiamo tralasciare la parte "alarms". E "triggers"? Dice cosa fa l'allarme ibrido, il che potrebbe sembrare buono, ma quando guardiamo la composizione, `Calls()` e `MakesLoudNoise()` dicono già cosa viene fatto. `HybridAlarm` dice solo che entrambe le cose accadono contemporaneamente. Potremmo lasciare `Trigger` se cambiassimo i nomi degli altri metodi nella composizione in questo modo:

```
  new GuardsBuilding(
    TriggersBoth(
      Calling("919"), //police number
      LoudNoise()
    )
  )
```

Ma ciò renderebbe fuori posto i nomi `Calling()` e `LoudNoise()` ovunque non siano annidati come argomenti di `TriggersBoth()`. Ad esempio, se volessimo realizzare un altro edificio che utilizzi solo un allarme sonoro, la composizione sarebbe simile a questa:

```
new OtherBuilding(LoudNoise());
```

o se volessimo usare quello silenzioso:

```
new OtherBuilding(Calling("919"));
```

Invece, proviamo a chiamare il metodo che racchiude la costruzione di `HybridAlarm` semplicemente `Both()` -- è semplice e comunica bene il ruolo svolto dagli allarmi ibridi -- dopo tutto, sono solo una sorta di operatori combinati, non allarmi reali. In questo modo, il nostro codice di composizione ora è:

```
new GuardsBuilding(
  Both(
    Calls("919"), //police number
```

(continues on next page)

(continua dalla pagina precedente)

```
    MakesLoudNoise()
  )
)
```

e, a proposito, il metodo `Both()` è definito come:

```
public Alarm Both(Alarm alarm1, Alarm alarm2)
{
  return new HybridAlarm(alarm1, alarm2);
}
```

Da ricordare che `HybridAlarm` è stato utilizzato anche nella composizione dell'istanza `StorageBuilding`:

```
new StorageBuilding(
  new HybridAlarm(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

che ora diventa:

```
new StorageBuilding(
  Both(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

Ora la parte più difficile -- trovare un modo per rendere leggibile la seguente parte di codice:

```
new OfficeBuilding(
  new DayNightSwitchedAlarm(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

La difficoltà qui è che `DayNightSwitchedAlarm` accetta due allarmi che vengono utilizzati alternativamente. Dobbiamo inventare un termine che:

1. Dica che è un'alternativa.
2. Dica di che tipo di alternativa si tratta (cioè che una avviene di giorno e l'altra di notte).
3. Dica quale allarme è collegato a quale condizione (l'allarme silenzioso viene utilizzato durante il giorno e quello sonoro durante la notte).

Se introduciamo un singolo nome, ad es. `FirstDuringDayAndSecondAtNight()`, sembrerà imbarazzante e perderemo il flusso. Uno sguardo:

```
new OfficeBuilding(
  FirstDuringDayAndSecondAtNight(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

Semplicemente non suona bene... Dobbiamo trovare un altro approccio a questa situazione. Ci sono due approcci che possiamo considerare:

32.2.1 Approccio 1: utilizzare i [*named parameter*]

I [*named parameter*] sono una funzionalità di linguaggi come Python o C#. In breve, quando abbiamo un metodo come questo:

```
public void DoSomething(int first, int second)
{
  //...
}
```

possiamo chiamarlo con i nomi dei suoi argomenti dichiarati esplicitamente, in questo modo:

```
DoSomething(first: 12, second: 33);
```

Possiamo utilizzare questa tecnica per rifattorizzare la creazione di `DayNightSwitchedAlarm` nel seguente metodo:

```
public Alarm DependingOnTimeOfDay(
    Alarm duringDay, Alarm atNight)
{
  return new DayNightSwitchedAlarm(duringDay, atNight);
}
```

Questo ci permette di scrivere il codice di composizione in questo modo:

```
new OfficeBuilding(
  DependingOnTimeOfDay(
    duringDay: Calls("222-333-444"),
    atNight: MakesLoudNoise()
  )
),
```

che è abbastanza leggibile. L'uso dei [*named parameter* ha questo piccolo vantaggio aggiuntivo che ci consente di passare gli argomenti in un ordine diverso in cui sono stati dichiarati, grazie ai loro nomi dichiarati esplicitamente. Ciò rende valide entrambe le seguenti invocazioni:

```
//this is valid:
DependingOnTimeOfDay(
  duringDay: Calls("222-333-444"),
  atNight: MakesLoudNoise()
)

//arguments in different order,
//but this is valid as well:
DependingOnTimeOfDay(
  atNight: MakesLoudNoise(),
  duringDay: Calls("222-333-444")
)
```

Passiamo ora al secondo approccio.

32.2.2 Approccio 2: utilizzare il concatenamento dei metodi

Questo approccio è meglio traducibile in diversi linguaggi e può essere utilizzato ad es. in Java e in C++. Questa volta, prima di mostrarvi l'implementazione, diamo un'occhiata al risultato finale che vogliamo ottenere:

```
new OfficeBuilding(
  DependingOnTimeOfDay
    .DuringDay(Calls("222-333-444"))
    .AtNight(MakesLoudNoise())
```

(continues on next page)

(continua dalla pagina precedente)

```
  )
),
```

Quindi, è molto simile nella lettura, la differenza principale è che richiede più lavoro. Potrebbe non essere ovvio fin dall'inizio come funziona questo tipo di passaggio di parametri:

```
DependingOnTimeOfDay
  .DuringDay(...)
  .AtNight(...)
```

quindi decodifichiamolo. Innanzitutto, `DependingOnTimeOfDay`. Questa è solo una classe:

```
public class DependingOnTimeOfDay
{
}
```

che ha un metodo statico chiamato `DuringDay()`:

```
//note: this method is static
public static
DependingOnTimeOfDay DuringDay(Alarm alarm)
{
  return new DependingOnTimeOfDay(alarm);
}

//The constructor is private:
private DependingOnTimeOfDay(Alarm dayAlarm)
{
  _dayAlarm = dayAlarm;
}
```

Ora, questo metodo sembra strano, non è vero? È un metodo statico che restituisce un'istanza della classe che lo racchiude (non un vero allarme!). Inoltre, il costruttore privato memorizza l'allarme trasmesso per poterlo utilizzare in seguito... perché?

Il mistero si risolve quando guardiamo un altro metodo definito nella classe `DependingOnTimeOfDay`:

```
//note: this method is NOT static
public Alarm AtNight(Alarm nightAlarm)
{
  return new DayNightSwitchedAlarm(_dayAlarm, nightAlarm);
}
```

Questo metodo non è statico e restituisce l'allarme che stavamo cercando di creare. Per fare ciò, utilizza il primo allarme passato attraverso il costruttore e il secondo passato come parametro. Quindi se dovessimo prendere questo costrutto:

```
DependingOnTimeOfDay //class
  .DuringDay(dayAlarm) //static method
  .AtNight(nightAlarm) //non-static method
```

e assegnare il risultato di ciascuna operazione a una variabile separata, sarebbe simile a questo:

```
DependingOnTimeOfDay firstPart = DependingOnTimeOfDay.DuringDay(dayAlarm);
Alarm alarm = firstPart.AtNight(nightAlarm);
```

Ora possiamo semplicemente concatenare queste chiamate e ottenere il risultato che volevamo:

```
new OfficeBuilding(
  DependingOnTimeOfDay
    .DuringDay(Calls("222-333-444"))
    .AtNight(MakesLoudNoise())
  )
),
```

Il vantaggio di questa soluzione è che non richiede un determinato linguaggio di programmazione per supportare i [*named parameter*]. Lo svantaggio è che l'ordine delle chiamate è rigorosamente definito. `DuringDay` restituisce un oggetto su cui viene invocato `AtNight`, quindi deve venire per primo.

32.2.3 La discussione è continua

Per ora presumo che abbiamo scelto l'approccio 1 perché è più semplice.

Il nostro codice di composizione finora è simile a questo:

```
new SecureArea(
  new OfficeBuilding(
    DependingOnTimeOfDay(
      duringDay: Calls("222-333-444"),
      atNight: MakesLoudNoise()
    )
  ),
  new StorageBuilding(
    Both(
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new GuardsBuilding(
    Both(
      Calls("919"), //police number
      MakesLoudNoise()
    )
  )
);
```

Ci sono ancora alcuni ritocchi finali che dobbiamo apportare. Prima di tutto, proviamo a estrarre questi numeri da comporre come `222-333-444` in costanti. Quando lo facciamo, quindi, ad esempio, questo codice:

```
Both(
  Calls("919"), //police number
  MakesLoudNoise()
)
```

diventa

```
Both(
  Calls(Police),
  MakesLoudNoise()
)
```

E l'ultima cosa è nascondere la creazione delle seguenti classi: `SecureArea`, `OfficeBuilding`, `StorageBuilding`, `GuardsBuilding` e abbiamo questo:

```
SecureAreaContaining(
  OfficeBuildingWithAlarmThat(
```

(continues on next page)

(continua dalla pagina precedente)

```
    DependingOnTimeOfDay(
      duringDay: Calls(Guards),
      atNight: MakesLoudNoise()
    )
  ),
  StorageBuildingWithAlarmThat(
    Both(
      Calls(Guards),
      MakesLoudNoise()
    )
  ),
  GuardsBuildingWithAlarmThat(
    Both(
      Calls(Police),
      MakesLoudNoise()
    )
  )
);
```

Ed eccola qui -- la descrizione reale e dichiarativa della nostra applicazione! La composizione si legge meglio di quando abbiamo iniziato, vero?

32.3 La composizione come linguaggio

Scritta in questo modo, la composizione degli oggetti ha un'altra proprietà importante -- è estensibile utilizzando gli stessi termini già usati (ovviamente possiamo anche aggiungerne di nuovi). Ad esempio, utilizzando i metodi che abbiamo inventato per rendere la composizione più leggibile, potremmo scrivere qualcosa del genere:

```
Both(
  Calls(Police),
  MakesLoudNoise()
)
```

ma, usando gli stessi termini, possiamo anche scrivere questo:

```
Both(
  Both(
    Calls(Police),
    Calls(Security)),
  Both(
    Calls(Boss),
    MakesLoudNoise()))
)
```

per ottenere un comportamento diverso. Notare che abbiamo inventato qualcosa che ha queste proprietà:

1. Definisce una sorta di *vocabolario* -- nel nostro caso, le seguenti "parole" fanno parte del vocabolario: `Both`, `Calls`, `MakesLoudNoise`, `DependingOnTimeOfDay`, `atNight`, `duringDay`, `SecureAreaContaining`, `GuardsBuildingWithAlarmThat`, `OfficeBuildingWithAlarmThat`.
2. Permette di combinare le parole del vocabolario. Queste combinazioni hanno un significato, che si basa esclusivamente sul significato delle parole usate e sul modo in cui sono combinate. Ad esempio: `Both(Calls(Police), Calls(Guards))` ha il significato di "chiama sia la polizia che le guardie quando attivato" -- quindi, ci permette di combinare le parole in *frasi*.
3. Anche se siamo abbastanza liberali nel definire i comportamenti per gli allarmi, ci sono alcune regole su cosa può essere composto con cosa (ad esempio, non possiamo comporre l'edificio delle guardie con un ufficio, ma ognuno di

essi può essere composto solo con gli allarmi). Pertanto, possiamo dire che le *frasi* che scriviamo devono obbedire a determinate regole che assomigliano molto a *una grammatica*.

4. Il vocabolario è *vincolato al dominio* degli allarmi. D'altra parte, *è più potente ed espressivo* come descrizione del dominio rispetto a una combinazione di istruzioni `if`, cicli `for`, assegnazioni di variabili e altri elementi di un linguaggio generico. È sintonizzato sulla descrizione delle regole di un dominio su un *livello di astrazione più elevato*.
5. Le frasi scritte definiscono il comportamento dell'applicazione -- quindi scrivendo frasi come questa continuiamo a scrivere software! Pertanto, ciò che facciamo combinando *parole* in *frasi* vincolate da una *grammatica* è ancora *programmazione*!

Tutti questi punti suggeriscono che abbiamo creato un *Domain-Specific Language*[Pag. 211, 1], che, tra l'altro, è un *linguaggio di livello superiore*, nel senso che descriviamo il nostro software a un livello di astrazione più elevato.

32.4 Il significato di un linguaggio di livello superiore

Quindi... perché abbiamo bisogno di un linguaggio di livello superiore per descrivere il comportamento della nostra applicazione? Dopotutto, espressioni, affermazioni, cicli e condizioni (e oggetti e polimorfismo) sono il nostro pane quotidiano. Perché inventare qualcosa che ci allontani da questo tipo di programmazione per indirizzarci verso qualcosa di "domain-specific"?

La mia risposta principale è: affrontare la complessità in modo più efficace.

Cos'è la complessità? Per il nostro scopo, possiamo definirla approssimativamente come il numero di decisioni diverse che la nostra applicazione deve prendere. Man mano che aggiungiamo nuove funzionalità, correggiamo errori o implementiamo requisiti non rispettati, la complessità del nostro software aumenta. Cosa possiamo fare quando diventa più grande di quanto possiamo gestire? Abbiamo le seguenti scelte:

1. Rimuovere alcune decisioni -- ovvero rimuovere funzionalità dalla nostra applicazione. È molto interessante quando *possiamo* farlo, ma ci sono momenti in cui ciò potrebbe essere inaccettabile dal punto di vista aziendale.
2. Ottimizzare le decisioni ridondanti -- si tratta di far sì che ogni decisione venga presa una sola volta nella base di codice. Ho già mostrato alcuni esempi di come il polimorfismo possa aiutare in questo.
3. Utilizzare componenti di terze parti o una libreria per farci gestire alcune delle decisioni -- mentre questo è abbastanza facile per il codice e le utilità dell'"infrastruttura", è molto, molto difficile (impossibile?) trovare una libreria che ci descriva le "regole del nostro dominio". Quindi, se queste regole sono la vera complessità (e spesso lo sono), rimaniamo ancora soli col nostro problema.
4. Nascondere le decisioni programmando a un livello di astrazione più elevato -- questo è ciò che abbiamo fatto finora in questo capitolo. Il vantaggio è che ci consente di ridurre la complessità del nostro dominio, creando blocchi più grandi da cui è possibile creare una descrizione del comportamento.

Quindi, solo lultimo dei punti precedenti aiuta davvero a ridurre la complessità del dominio. È qui che rientra lidea di linguaggi specifici del dominio (domain-specific languages). Se creiamo attentamente la composizione del nostro oggetto in un insieme di linguaggi specifici del dominio (uno è spesso troppo poco per tutto tranne che nei casi più semplici), un giorno potremmo scoprire che stiamo aggiungendo nuove funzionalità scrivendo nuove frasi in questi linguaggi in modo dichiarativo piuttosto che aggiungere un nuovo codice imperativo. Pertanto, se possediamo un buon linguaggio e una solida conoscenza del suo vocabolario e della grammatica, possiamo programmare a un livello di astrazione più elevato, più espressivo e meno complesso.

Questo è molto difficile da realizzare poiché richiede, tra le altre cose:

1. Una ferrea disciplina in un team di sviluppo.
2. Un senso dell'orientamento su come strutturare la composizione e dove condurre i progetti dei linguaggi mentre si evolvono.
3. Un refactoring spietato.
4. Una conoscenza minima della progettazione del linguaggio ed esperienza nel farlo.
5. Conoscenza di alcune tecniche (come quelle usate nel nostro esempio) che fanno sembrare i costrutti scritti in un linguaggio generico come un altro linguaggio.

Naturalmente, non tutte le parti della composizione costituiscono un buon materiale per essere strutturate come un linguaggio. Nonostante queste difficoltà, penso che ne valga la pena. Programmare a un livello di astrazione più elevato con codice dichiarativo anziché imperativo è il luogo in cui ripongo la mia speranza di scrivere sistemi gestibili e comprensibili.

32.5 Qualche consiglio

Allora, si è ansiosi di provare questo approccio? Permettimi alcuni consigli:

32.5.1 Far evolvere il linguaggio man mano che evolve il codice

All'inizio di questo capitolo, abbiamo raggiunto il nostro linguaggio di livello superiore effettuando il refactoring della composizione di oggetti già esistente. Ciò non significa affatto che nei progetti reali dobbiamo aspettare che appaia molto codice di composizione e poi provare a racchiuderlo tutto. In effetti ho fatto proprio questo nell'esempio dell'allarme, ma questo era solo un esempio e il suo scopo era principalmente didattico.

In realtà, è meglio che il linguaggio evolva insieme alla composizione che descrive. Uno dei motivi è che ci sono molti feedback sulla componibilità del progetto ottenuti provando a metterci sopra un linguaggio. Come ho detto nel capitolo sulla responsabilità singola, se gli oggetti non sono comodamente componibili, probabilmente c'è qualcosa che non va nella distribuzione delle responsabilità tra di loro (per un confronto delle responsabilità erroneamente assegnate, immaginiamo un linguaggio generico che non abbia costrutti `if` e `for` separati ma solo una combinazione di essi chiamata `forif` :-)). Non perdersi questo feedback!

La seconda ragione è che anche se si può tranquillamente rifattorizzare tutto il codice perché si ha una specifica eseguibile che ci protegge dal commettere errori, sono semplicemente troppe decisioni da gestire contemporaneamente (in più ci vuole molto tempo e i colleghi continuano ad aggiungere nuovo codice, vero?). Un buon linguaggio cresce e matura in modo organico anziché essere creato in uno sforzo enorme. Alcune decisioni richiedono tempo e molta riflessione.

32.5.2 La composizione non è un singolo DSL, ma una serie di mini DSL[2]

L'ho già brevemente notato. Anche se può essere forte la tentazione di inventare un unico DSL per descrivere l'intera applicazione, in pratica è difficilmente possibile, perché le nostre applicazioni hanno sottodomini diversi che spesso utilizzano insiemi di termini diversi. Piuttosto, conviene cercare tali sottodomini e creare per essi linguaggi più piccoli. L'esempio dell'allarme mostrato sopra sarebbe probabilmente solo una piccola parte della composizione reale. Non tutte le parti si presterebbero a modellarsi in questo modo, almeno non immediatamente. Ciò che inizia come una singola classe potrebbe, ad un certo punto, diventare un sottodominio con un proprio vocabolario. Dobbiamo prestare attenzione. Quindi, vogliamo ancora applicare alcune delle tecniche DSL anche a quelle parti della composizione che non possono essere facilmente trasformate in DSL e cercare un'occasione in cui possiamo farlo.

Come dice Nat Pryce, si tratta di:

> (...) esprimere chiaramente le dipendenze tra gli oggetti nel codice che li compone, in modo che la struttura del sistema possa essere facilmente rifattorizzata, ed eseguire aggressivamente il refactoring di quel codice per rimuovere la duplicazione ed esprimere l'intento, e quindi aumentare il livello di astrazione in cui possiamo programmare (...). L'obiettivo finale è avere bisogno di sempre meno codice per scrivere sempre più funzionalità man mano che il sistema cresce.

Ad esempio, un mini-DSL per impostare la gestione degli aggiornamenti della configurazione di un'applicazione potrebbe assomigliare a questo:

```
return ConfigurationUpdates(
  Of(log),
  Of(localSettings),
  OfResource(Departments()),
  OfResource(Projects()));
```

Leggere questo codice non dovrebbe essere difficile, soprattutto quando sappiamo cosa significa ogni termine nella frase. Questo codice restituisce un oggetto che gestisce gli aggiornamenti della configurazione di quattro cose: il log dell'applicazione, le impostazioni locali e due risorse (in questo sottodominio, risorse significa cose che possono essere aggiunte,

[2] Un lettore ha notato che le idee in questa sezione sono notevolmente simili alla nozione di "*Bounded Contexts*" in un libro: E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Prentice Hall 2003.

cancellate e modificate). Queste due risorse sono: dipartimenti e progetti (ad esempio possiamo aggiungere un nuovo progetto o eliminarne uno esistente).

Si noti che i costrutti di questo linguaggio hanno senso solo in un contesto di creazione di gestori di aggiornamento della configurazione. Pertanto, dovrebbero essere limitati a questa parte della composizione. Altre parti che non hanno nulla a che fare con gli aggiornamenti della configurazione non dovrebbero aver bisogno di conoscere questi costrutti.

32.5.3 Non utilizzare una quantità eccessiva di trucchi DSL

Nella creazione di DSL interni, si possono usare molti trucchi, alcuni dei quali sono molto "hacky" e distorcono il linguaggio generale in molti modi per ottenere una sintassi "fluente". C'è da ricordare che il codice di composizione deve essere mantenuto dal team. A meno che ogni membro del team non sia un esperto nella creazione di tali DSL, non esporre trucchi troppo sofisticati. Scegliere alcuni di quelli collaudati che sono semplici da usare e funzionano, come quelli che ho usato nell'esempio dell'allarme.

Martin Fowler[Pag. 211, 1] descrive molti trucchi per creare tali DSL e allo stesso tempo mette in guardia dall'usarne troppi nello stesso linguaggio.

32.5.4 Il [*Factory method nesting*] è il migliore

Una delle tecniche DSL, quella che ho utilizzato di più, è il [*factory method nesting*]. Fondamentalmente, significa avvolgere l'invocazione di un costruttore (o costruttori -- nessuno ha detto che ogni metodo factory debba racchiudere esattamente un `new`) con un metodo che abbia un nome più adatto al contesto in cui viene utilizzato (e che nasconde l'oscurità della parola chiave `new`). Questa tecnica è quello che rende questo:

```
new HybridAlarm(
  new SilentAlarm("222-333-444"),
  new LoudAlarm()
)
```

e lo fa assomigliare a questo:

```
Both(
  Calls("222-333-444"),
  MakesLoudNoise()
)
```

Come si ricorderà, in questo caso, ogni metodo racchiude un costruttore, ad es. `Calls()` è definito come:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(number);
}
```

Questa tecnica è ottima per descrivere qualsiasi tipo di struttura ad albero e simile a un grafo poiché ciascun metodo fornisce uno scopo naturale per i suoi argomenti:

```
Method1( //beginning of scope
  NestedMethod1(),
  NestedMethod2()
);       //end of scope
```

Pertanto, è un adattamento naturale per la composizione di oggetti, che *è* una struttura a grafo.

Questo approccio sembra ottimo sulla carta, ma non è che tutto si adatti sempre. Ci sono due problemi con i metodi factory da affrontare.

Dove mettere questi metodi?

Nel caso normale, vogliamo essere in grado di invocare questi metodi senza alcun qualificatore prima di essi, cioè vogliamo chiamare `MakesLoudNoise()` invece di `alarmsFactory.MakesLoudNoise()` o di `this.MakesLoudNoise()` o altro.

Se sì, dove mettiamo tali metodi?

Sono disponibili due opzioni[3]:

1. Inserire i metodi nella classe che esegue la composizione.
2. Mettere i metodi in una superclasse.

Oltre a ciò, possiamo scegliere tra:

1. Rendere statici i metodi factory.
2. Rendere i metodi factory non statici.

Innanzitutto, consideriamo il dilemma tra l'inserimento di una classe di composizione e l'avere una superclasse da cui ereditare. Questa scelta è determinata principalmente dalle esigenze del riutilizzo. I metodi che usiamo soltanto in una composizione e che non vogliamo riutilizzare è meglio che siano metodi privati nella classe di composizione. D'altra parte, i metodi che vogliamo riutilizzare (ad esempio in altre applicazioni o servizi appartenenti allo stesso sistema), è meglio inserirli in una superclasse da cui possiamo ereditare. Inoltre, è possibile una combinazione dei due approcci, in cui la superclasse contiene un metodo più generale mentre la classe di composizione lo racchiude con un altro metodo che adatta la creazione al contesto corrente. A proposito, ricordare che in molti linguaggi possiamo ereditare solo da una singola classe -- quindi mettere i metodi per ogni linguaggio in una superclasse separata ci costringe a distribuire il codice di composizione su più classi, ciascuna eredita il proprio insieme di metodi e restituisce uno o più oggetti. Questo non è affatto male -- al contrario, è qualcosa che vorremmo avere perché ci permette di far evolvere una lingua e le frasi scritte in tale lingua in un contesto isolato.

La seconda scelta tra statico e non statico è quella di avere accesso ai campi dell'istanza -- i metodi dell'istanza hanno questo accesso, mentre i metodi statici no. Pertanto, se quanto segue è un metodo di istanza di una classe chiamata `AlarmComposition`:

```
public class AlarmComposition
{
  //...

  public Alarm Calls(string number)
  {
    return new SilentAlarm(number);
  }

  //...
}
```

e si deve passare una dipendenza aggiuntiva a `SilentAlarm` che non si vuole mostrare nel codice di composizione principale, si è liberi di cambiare il metodo `Calls` in:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(
    number,
    _hiddenDependency) //field
}
```

e questa nuova dipendenza può essere passata a `AlarmComposition` tramite il suo costruttore:

[3] In alcuni linguaggi esiste una terza via: Java ci consente di utilizzare importazioni statiche che fanno parte anche di C# a partire dalla versione 6.0. Il C++ ha sempre supportato le funzioni, quindi non è un argomento lì.

```
public AlarmComposition(
  HiddenDependency hiddenDependency)
{
  _hiddenDependency = hiddenDependency;
}
```

In questo modo posso nasconderlo dal codice di composizione principale. Questa è la libertà che non ho con i metodi statici.

Utilizzare collection implicite anziché esplicite

a maggior parte dei linguaggi object-oriented supporta il passaggio di elenchi variabili di argomenti (ad esempio in C# ciò si ottiene con la parola chiave `params`, mentre Java ha l'operatore `...`). Questo è utile nella composizione, perché spesso vogliamo essere in grado di passare un numero arbitrario di oggetti in alcuni posti. Ancora una volta, tornando a questa composizione:

```
return ConfigurationUpdates(
  Of(log),
  Of(localSettings),
  OfResource(Departments()),
  OfResource(Projects()));
```

il metodo `ConfigurationUpdates()` tilizza un elenco di argomenti variabili:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  return new MyAppConfigurationUpdates(updates);
}
```

Notare che potremmo, ovviamente, passare l'array di istanze `ConfigurationUpdate` utilizzando la definizione esplicita: `new ConfigurationUpdate[] {...}`, ma ciò ostacolerebbe notevolmente la leggibilità e il flusso di questa composizione. Guardate voi stessi:

```
return ConfigurationUpdates(
  new [] { //explicit definition brings noise
    Of(log),
    Of(localSettings),
    OfResource(Departments()),
    OfResource(Projects())
  }
);
```

Non tanto bello, eh? Questo è il motivo per cui ci piace la possibilità di passare elenchi di argomenti variabili poiché migliora la leggibilità.

Un singolo metodo può creare più di un oggetto

Nessuno ha detto che ogni metodo factory debba creare uno e un solo oggetto. Ad esempio, si dia un'altra occhiata a questo metodo per creare aggiornamenti della configurazione:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  return new MyAppConfigurationUpdates(updates);
}
```

Ora, supponiamo di dover tracciare ogni invocazione sull'istanza della classe `ConfigurationUpdates` e di voler ottenere questo risultato racchiudendo l'istanza `MyAppConfigurationUpdates` con un proxy di tracing (un oggetto che passa le

chiamate a un oggetto reale che racchiude, ma scrive alcuni messaggi di trace prima e dopo). A questo scopo possiamo riutilizzare il metodo che già abbiamo, semplicemente aggiungendovi la creazione di oggetti aggiuntivi:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  //now two objects created instead of one:
  return new TracedConfigurationUpdates(
    new MyAppConfigurationUpdates(updates)
  );
}
```

Notare che `TracedConfigurationUpdates` non è importante dal punto di vista della composizione -- si tratta di puro codice infrastrutturale, non di una nuova regola di dominio. Per questo motivo, potrebbe essere una buona idea nasconderlo all'interno del metodo factory.

32.6 Riepilogo

In questo capitolo ho cercato di trasmettere una visione della composizione di oggetti come un linguaggio, con un proprio vocabolario, una propria grammatica, con parole chiave e argomenti. Possiamo comporre le parole del vocabolario in frasi diverse per creare nuovi comportamenti a un livello di astrazione più elevato.

Quest'area della progettazione orientata agli oggetti è qualcosa con cui sto ancora sperimentando, cercando di mettermi al passo con ciò che condividono altri autori su questo argomento. Pertanto, non lo parlo così fluentemente come in altri argomenti trattati in questo libro. Aspettatevi che questo capitolo cresca (magari in più capitoli) o che venga chiarito in futuro. Per ora, se si ritiene di aver bisogno di maggiori informazioni, dare un'occhiata al video di Steve Freeman e Nat Pryce intitolato "Building on SOLID foundations".

CAPITOLO 33

Gli *Oggetti Valore*

In diversi capitoli ho parlato della composizione di oggetti in una rete in cui la vera implementazione era nascosta esponendo solo le interfacce. Questi oggetti si scambiavano messaggi e modellavano ruoli nel nostro dominio.

Tuttavia, questa è solo una parte dell'approccio object-oriented agli oggetti che sto cercando di spiegare. Un'altra parte del mondo object-oriented , complementare a ciò di cui abbiamo parlato, sono gli *oggetti valore*. Hanno una propria serie di vincoli e idee di progettazione, quindi la maggior parte dei concetti dei capitoli precedenti non si applicano a loro o si applicano in modo diverso.

33.1 Cos'è un *oggetto valore*?

In breve, i valori sono solitamente visti come quantità immutabili, misure[1] o altri oggetti che vengono confrontati in base al loro contenuto, non alla loro identità. Sono presenti alcuni esempi di valori nelle librerie dei nostri linguaggi di programmazione. Ad esempio, la classe `String` in Java o in C# è un oggetto valore perché è immutabile e ogni due stringhe sono considerate uguali quando contengono gli stessi dati. Altri esempi sono i tipi primitivi nativi nella maggior parte dei linguaggi di programmazione, come numeri o caratteri.

La maggior parte dei valori forniti con le librerie general-purpose sono piuttosto primitivi o generali. Ci sono molte volte, tuttavia, in cui vogliamo modellare un'astrazione di dominio come un *oggetto valore*. Alcuni esempi includono data e ora (che oggigiorno di solito fanno parte della libreria standard, perché è utilizzabile in tanti domini), denaro, temperatura, ma anche cose come path di file o identificatori di risorse.

Come già notato leggendo questo libro, sono davvero pessimo nello spiegare le cose senza esempi, quindi eccone uno:

33.2 Esempio: soldi e nomi

Immaginiamo di sviluppare un negozio web per un cliente. Esistono diversi tipi di prodotti in vendita e il cliente desidera avere la possibilità di aggiungere di nuovi.

Ogni prodotto ha almeno due attributi importanti: nome e prezzo (ce ne sono altri come la quantità, ma per ora lasciamoli stare).

Ora, immaginiamo come si modellerebbero queste due cose: il nome sarebbe modellato come una semplice `string` e il prezzo sarebbe di tipo `double` o `decimal`?

Diciamo che abbiamo effettivamente deciso di utilizzare un `decimal` per contenere un prezzo e una `string` per contenere un nome. Notare che entrambi sono tipi di libreria generici, non collegati ad alcun dominio. È una buona scelta utilizzare i "tipi della libreria" per le astrazioni del dominio? Lo scopriremo presto...

[1] S. Freeman, N. Pryce, Growing Object-Oriented Software Guided by Tests, Addison-Wesley Professional, 2009

33.2.1 Il tempo passa...

Un giorno si scopre che questi valori devono essere condivisi tra diversi sottodomini del sistema. Per esempio:

1. Il sito web deve visualizzarli
2. Sono utilizzati nei calcoli dei ricavi
3. Vengono presi in considerazione nella definizione e nella verifica delle regole di sconto (ad esempio "compra tre, paghi due")
4. Devono essere forniti durante la stampa delle fatture

ecc.

Il codice diventa sempre più grande e, poiché i concetti di nome del prodotto e prezzo sono tra quelli principali dell'applicazione, tendono ad arrivare in molti posti.

33.2.2 Richiesta di modifica

Ora, immaginiamo che uno dei seguenti cambiamenti debba farsi strada nel sistema:

1. Il nome del prodotto deve essere confrontato senza distinzione tra maiuscole e minuscole poiché i nomi dei prodotti sono sempre stampati in maiuscolo sulla fattura. Pertanto, creando due prodotti che differiscono solo per la lettera maiuscola (es. "laptop" e "LAPTOP") confonderebbero i clienti poiché entrambi i prodotti appaiono uguali sulla fattura. Inoltre, l'unico modo per creare due prodotti che differiscono solo per le lettere è per errore e vogliamo evitarlo.
2. Il nome del prodotto non è sufficiente per differenziare un prodotto. Ad esempio, i produttori di PC hanno gli stessi modelli di notebook in diverse configurazioni (ad esempio diversa quantità di RAM o diversi modelli di processore all'interno). Quindi ogni prodotto riceverà un identificativo aggiuntivo di cui si dovrà tenere conto durante i confronti.
3. Per supportare i clienti di paesi diversi, è necessario supportare nuove valute.

Nella situazione attuale, questi cambiamenti sono laboriosi da apportare. Perché? È perché abbiamo utilizzato i tipi primitivi per rappresentare le cose che avrebbero dovuto cambiare, il che significa che siamo accoppiati in più punti a una particolare implementazione del nome del prodotto (`string`) e a una particolare implementazione del denaro (ad esempio `decimal`) . Non sarebbe poi così male, se non fosse per il fatto che siamo vincolati all'implementazione e non possiamo cambiare!

Siamo condannati a convivere con problemi del genere nel nostro codice e non possiamo farci nulla? Scopriamolo esplorando le opzioni a nostra disposizione.

D'ora in poi mettiamo da parte il concetto di denaro e concentriamoci solo sul nome del prodotto, poiché sia nome che prezzo sono casi simili con soluzioni simili, quindi ci basta considerarne solo uno.

33.2.3 Quali opzioni abbiamo per affrontare le modifiche al nome del prodotto?

Per supportare i nuovi requisiti, dobbiamo trovare tutti i posti in cui utilizziamo il nome del prodotto (a proposito, un IDE non ci aiuterà molto in questa ricerca, perché cercheremmo tutte le occorrenze del tipo `string`) e fare la stessa modifica. Ogni volta che dobbiamo fare qualcosa del genere (ovvero dobbiamo apportare la stessa modifica in più posti e c'è una possibilità diversa da zero di perdere almeno uno di quei posti), significa che abbiamo introdotto la ridondanza. Ricordate? Abbiamo parlato di ridondanza parlando di factory e abbiamo accennato al fatto che la ridondanza riguarda una duplicazione concettuale che ci costringe a apportare lo stesso cambiamento (non letteralmente, ma concettualmente) in diversi luoghi.

Al Shalloway ha coniato una "legge" divertente sulla ridondanza, chiamata *La Legge di Shalloway*, che dice:

> Ogni volta che la stessa modifica deve essere applicata in N posti e N > 1, Shalloway troverà al massimo N-1 di tali posti.

Un esempio di applicazione di questa legge potrebbe essere:

> Ogni volta che è necessario applicare la stessa modifica in 4 posti, Shalloway ne troverà al massimo 3.

Prendendosi in giro, Al ha descritto qualcosa che vedo comune a me e ad altri programmatori: che la duplicazione concettuale ci rende vulnerabili e quando ci abbiamo a che fare, non abbiamo strumenti avanzati per aiutarci, solo la nostra memoria e pazienza.

Per fortuna, ci sono diversi modi per affrontare questa ridondanza. Alcuni di essi sono migliori e altri peggiori[2].

Opzione uno: modificare semplicemente l'implementazione in tutti i punti

Questa opzione consiste nel lasciare la ridondanza dov'è e apportare semplicemente la modifica in tutti i punti, sperando che questa sia l'ultima volta che cambiamo qualcosa riguardo al nome del prodotto.

Quindi diciamo che vogliamo aggiungere il confronto ignorando le maiuscole. Questa opzione ci porterebbe a trovare tutti i posti in cui facciamo qualcosa del genere:

```
if(productName == productName2)
{
..
```

o

```
if(String.Equals(productName, productName2))
{
..
```

Cambiandoli in un confronto che ignori maiuscole e minuscole, ad esempio:

```
if(String.Equals(productName, productName2,
   StringComparison.OrdinalIgnoreCase))
{
..
```

Questo risolve il problema, almeno per ora, ma a lungo termine può causare qualche problema:

1. Sarà molto difficile trovare tutti questi posti e le possibilità sono che se ne perderà almeno uno. Questo è un modo semplice per far sì che un bug possa insinuarsi.
2. Anche se questa volta si sarà in grado di trovare e correggere tutti i posti, ogni volta che cambia la logica del dominio per i confronti dei nomi dei prodotti (ad esempio, dovremo utilizzare l'opzione `InvariantIgnoreCase` invece di `OrdinalIgnoreCase`per qualche motivo, o gestire il caso menzionato prima in cui il confronto include un identificatore di un prodotto), lo si dovrà rifare. E la legge di Shalloway si applica ogni volta allo stesso modo. In altre parole, le cose non stanno migliorando.
3. Chiunque in futuro aggiunga nuova logica per confrontare i nomi dei prodotti dovrà ricordare che in tali confronti maiuscole e minuscole vengono ignorate. Pertanto, dovranno tenere presente che dovrebbero utilizzare l'opzione `OrdinalIgnoreCase` ogni volta che aggiungono nuovi confronti da qualche parte nel codice. La mia opinione è che, la violazione accidentale di questa convenzione in un team che ha dimensioni adeguate o un tasso di turnover del personale superiore al minimo è solo questione di tempo.
4. Inoltre, ci sono altri cambiamenti che saranno legati al concetto di uguaglianza dei nomi dei prodotti in modo diverso (ad esempio, i set di hash e le tabelle hash utilizzano codici hash per aiutare a trovare gli oggetti giusti) e si dovranno trovare quei posti e apportare modifiche anche lì.

Quindi, come si vede, questo approccio non migliora le cose. È proprio questo approccio che ci ha portato al problema che stiamo cercando di risolvere.

Opzione due - utilizzare una classe helper

Possiamo affrontare i problemi #1 e #2 dell'elenco precedente (ovvero la necessità di cambiare più posizioni quando cambia la logica di confronto dei nomi dei prodotti) spostando questo confronto in un metodo helper statico di una classe helper (chiamiamolo semplicemente `ProductNameComparison`) e rendere questo metodo un unico posto in grado di confrontare

[2] Tutte le decisioni ingegneristiche sono comunque dei compromessi, quindi dovrei dire "alcune di esse apportano compromessi migliori nel nostro contesto, mentre altre li peggiorano".

i nomi dei prodotti. Ciò farebbe sì che ciascuno dei punti del codice in cui è necessario effettuare il confronto assomigli a questo:

```
if(ProductNameComparison.AreEqual(productName, productName2))
{
..
```

Si noti che i dettagli su cosa significa confrontare due nomi di prodotto sono ora nascosti all'interno del metodo statico `AreEqual()` appena creato. Questo metodo è diventato l'unico posto che conosce questi dettagli e ogni volta che è necessario cambiare il confronto, dobbiamo modificare solo questo metodo. Il resto del codice chiama semplicemente questo metodo senza sapere cosa fa, quindi non sarà necessario modificarlo. Questo ci libera dalla necessità di cercare e modificare questo codice ogni volta che cambia la logica di confronto.

Tuttavia, anche se ci protegge dal cambiamento della logica di confronto, non è ancora sufficiente. Perché? Perché il concetto di nome di prodotto non è ancora incapsulato - un nome di prodotto è ancora una `string` e ci consente di farci tutto ciò che possiamo fare con qualsiasi altra `string`, anche quando non ha senso per i nomi dei prodotti. Questo perché, nel dominio del problema, i nomi dei prodotti non sono sequenze di caratteri (come lo sono le `string`), ma un'astrazione con uno speciale insieme di regole ad essa applicabili. Non riuscendo a modellare questa astrazione in modo appropriato, possiamo imbatterci in una situazione in cui un altro sviluppatore che inizia ad aggiungere del nuovo codice potrebbe non notare nemmeno che i nomi dei prodotti devono essere confrontati in modo diverso rispetto ad altre stringhe e utilizzare semplicemente il confronto predefinito di un tipo `string`.

Si applicano anche altre carenze dell'approccio precedente (come menzionato, ad eccezione delle questioni #1 e #2).

Opzione tre - incapsulare il concetto di dominio e creare un "value object"

Penso che sia più che chiaro ora che il nome di un prodotto non è "solo una stringa", ma un concetto di dominio e come tale merita una classe a parte. Introduciamo quindi una classe di questo tipo e chiamiamola `ProductName`. Le istanze di questa classe avranno il metodo `Equals()` sovrascritto[3] con la logica specifica dei nomi di prodotto. Detto questo, lo snippet per il confronto ora è:

```
// productName and productName2
// are both instances of ProductName
if(productName.Equals(productName2))
{
..
```

In cosa differisce dall'approccio precedente in cui avevamo una classe helper, chiamata `ProductNameComparison`? In precedenza i dati del nome di un prodotto erano visibili pubblicamente (come una stringa) e utilizzavamo la classe helper solo per memorizzare una funzione che operava su questi dati (e chiunque poteva creare le proprie funzioni da qualche altra parte senza accorgersi di quelle che avevamo già aggiunto). Questa volta i dati del nome del prodotto sono nascosti[4] al mondo esterno. L'unico modo disponibile per operare su questi dati è attraverso l'interfaccia pubblica di `ProductName` (che espone solo i metodi che riteniamo abbiano senso per i nomi dei prodotti e nient'altro). In altre parole, mentre prima avevamo a che fare con un tipo generico che non potevamo cambiare, ora abbiamo un tipo specifico del dominio che è completamente sotto il nostro controllo. Ciò significa che possiamo cambiare liberamente il significato di due nomi uguali e questo cambiamento non si diffonderà in tutto il codice.

Nei capitoli seguenti esplorerò ulteriormente questo esempio di nomi di prodotto per mostrare alcune proprietà degli *oggetti valore*.

---

[3] e, per C#, anche sovrascrivere gli operatori di uguaglianza (== e !=) è probabilmente una buona idea, per non parlare di `GetHashCode()` (vedere https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/equality-operators)

[4] In realtà, questo è vero solo in parte. Ad esempio, dovremo comunque sovrascrivere `ToString()` da qualche parte per garantire l'interoperabilità con librerie di terze parti che non conoscono il nostro tipo `ProductName`, ma accetteranno argomenti di tipo stringa. Inoltre, è sempre possibile utilizzare la reflection per ottenere dati privati. Spero che sia chiaro il punto però :-).

CAPITOLO 34

Anatomia del *oggetto valore*

Nel capitolo precedente abbiamo visto un *oggetto valore* - `ProductName` in azione. In questo capitolo ne studieremo l'anatomia - riga per riga, campo per campo, metodo dopo metodo. Dopo aver fatto ciò, si spera che si abbia una idea migliore di alcune delle proprietà più generali degli *oggetti valore*.

Iniziamo la nostra disamina dando un'occhiata alla definizione del tipo `ProductName` del capitolo precedente (il codice che vi mostrerò non è legale C# - ho omesso i corpi dei metodi, inserendo un `;` dopo ogni dichiarazione di metodo. L'ho fatto perché altrimenti ci sarebbe molto codice da comprendere e non voglio necessariamente approfondire il codice di ciascun metodo). Ogni sezione della definizione della classe `ProductName` è contrassegnata da un commento. Tali commenti contrassegnano gli argomenti che discuteremo in questo capitolo.

Quindi ecco la definizione promessa di `ProductName`:

```
//This is the class we created and used
//in the previous chapter

// class signature
public sealed class ProductName
  : IEquatable<ProductName>
{
  // Hidden data:
  private string _value;

  // Constructor - hidden as well:
  private ProductName(string value);

  // Static method for creating new instances:
  public static ProductName For(string value);

  // Overridden version of ToString()
  // from Object class
  public override string ToString();

  // Non-standard version of ToString().
  // I will explain its purpose later
  public string ToString(Format format);

  // For value types, we need to implement all the equality
```

(continues on next page)

(continua dalla pagina precedente)

```
  // methods and operators, plus GetHashCode():
  public override bool Equals(Object other);
  public bool Equals(ProductName other);
  public override int GetHashCode();
  public static bool operator ==(ProductName a, ProductName b);
  public static bool operator !=(ProductName a, ProductName b);
}
```

Usando i commenti, ho diviso la lezione in sezioni e le descriverò nell'ordine.

34.1 Class [*firma*] della classe

Ci sono due cose da notare sulla firma della classe. La prima è che la classe è `sealed` [*sigillata*] (in Java sarebbe `final`), ovvero non consento l'ereditarietà da essa. Questo perché voglio che gli oggetti di questa classe siano immutabili. A prima vista, definire "sealed" la classe non ha nulla a che fare con limmutabilità. Lo spiegherò nel prossimo capitolo quando discuterò gli aspetti della progettazione degli *oggetti valore*.

La seconda cosa da notare è che la classe implementa un'interfaccia `IEquatable` che aggiunge versioni più fortemente tipizzate del metodo `Equals(T object)`. Questo non è strettamente richiesto come in C#, ogni oggetto ha un metodo `Equals(Object o)` predefinito, ma è generalmente considerato una buona pratica poiché consente ad es. l'uso più efficiente degli *oggetti valore* con collection C# come i `Dictionary`[1].

34.2 Dati nascosti

I dati effettivi sono privati:

```
private string _value;
```

Solo i metodi che pubblichiamo possono essere utilizzati per operare sullo stato. Questo è utile per tre cose:

1. Limitare le operazioni consentite a ciò che riteniamo abbia senso fare con il nome di un prodotto. Tutto il resto (ovvero ciò che riteniamo non abbia senso fare) non è consentito.
2. Per ottenere l'immutabilità delle istanze di `ProductName` (più avanti spiegheremo perché vogliamo che il tipo sia immutabile), il che significa che quando creiamo un'istanza, non possiamo modificarla. Se il campo `_value` fosse pubblico, chiunque potrebbe modificare lo stato dell'istanza di `ProductName` scrivendo qualcosa del tipo:

```
productName.data = "something different";
```

1. Per proteggersi dalla creazione di un nome di prodotto con uno stato non valido. Quando creiamo il nome di un prodotto, dobbiamo passare una stringa contenente un nome tramite un metodo statico `For()` che possa eseguire la validazione (ne parleremo più avanti). Se non ci sono altri modi in cui possiamo impostare il nome, possiamo essere certi che la convalida avverrà ogni volta che qualcuno vorrà creare un `ProductName`.

34.3 Costruttore nascosto

Notare che anche il costruttore è reso privato:

```
private ProductName(string value)
{
  _value = value;
}
```

e probabilmente ci si chiederà perché. Vorrei scomporre ulteriormente la domanda in altre due:

1. Come dovremmo allora creare nuove istanze?

[1] https://stackoverflow.com/questions/2734914/whats-the-difference-between-iequatable-and-just-overriding-object-equals

2. Perché *private* e non *public*?

Rispondiamo una per una.

34.3.1 Come dovremmo creare nuove istanze?

La classe `ProductName` contiene uno speciale metodo factory statico, chiamato `For()`. Richiama il costruttore e gestisce tutte le validazioni dei parametri di input[2]. Un esempio di implementazione potrebbe essere:

```
public static ProductName For(string value)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    //validation failed
    throw new ArgumentException(
      "Product names must be human readable!");
  }
  else
  {
    //here we call the constructor
    return new ProductName(value);
  }
}
```

Esistono diversi motivi per non esporre direttamente un costruttore ma utilizzare invece un metodo factory statico. Di seguito ne descrivo brevemente alcuni.

Spiegare l'intenzione

Proprio come le factory, i metodi delle factory statici aiutano a spiegare l'intenzione, perché, a differenza dei costruttori, possono avere nomi, mentre i costruttori hanno il vincolo di prendere il nome dalla loro classe[3]. Si può dedurre che quanto segue:

```
ProductName.For("super laptop X112")
```

non è più leggibile di:

```
new ProductName("super laptop X112");
```

ma si noti che nel nostro esempio abbiamo un unico e semplice metodo factory. Il vantaggio sarebbe più visibile quando avessimo bisogno di supportare un ulteriore modo di creare un nome di prodotto. Supponiamo che nell'esempio sopra del "super laptop X112", il "super laptop" sia un modello e "X112" sia una configurazione specifica (poiché gli stessi modelli di laptop sono spesso venduti in diverse configurazioni, con più o meno RAM, diversi sistemi operativi, ecc.) e troviamo comodo passare queste due informazioni come argomenti separati in alcuni punti (ad esempio perché potremmo ottenerle da fonti diverse) e lasciare che `ProductName` le metta assieme. Se utilizzassimo un costruttore per questo, scriveremmo:

```
// assume model is "super laptop"
// and configuration is "X112"
new ProductName(model, configuration)
```

D'altra parte, possiamo creare un metodo factory e dire:

```
ProductName.CombinedOf(model, configuration)
```

che si legge più fluentemente. Oppure, se vogliamo essere super espliciti:

[2] A proposito, il codice contiene una chiamata a `IsNullOrEmpty()`. Esistono diversi argomenti validi contro l'utilizzo di questo metodo, ad es. quello di Mark Seemann (http://blog.ploeh.dk/2014/11/18/the-isnullorwhitespace-trap/), ma in questo caso l'ho inserito per rendere il codice più breve poiché la logica di convalida in sé non è importante in questo momento.

[3] Questo è vero per linguaggi come Java, C# o C++. Esistono altri linguaggi (come Ruby), con regole diverse per quanto riguarda la costruzione degli oggetti. Tuttavia, l'argomentazione originale - che la denominazione dei metodi responsabili della creazione degli oggetti è vincolata - resta valida. Still, the original argument - that the naming of methods responsible for object creation is constrained - holds.

```
ProductName.FromModelAndConfig(model, configuration)
```

che non è il mio modo preferito di scrivere codice, perché non mi piace ripetere le stesse informazioni nel nome del metodo e nei nomi degli argomenti. Volevo dimostrare che possiamo farlo volendo.

Ho incontrato molti sviluppatori che trovano poco familiare l'uso dei metodi factory, ma la buona notizia è che i metodi factory per gli *oggetti valore* stanno diventando sempre più diffusi. Giusto per dare due esempi, il tipo `TimeSpan` in C# li usa (ad esempio possiamo scrivere `TimeSpan.FromSeconds(12)` e il tipo `Period` in Java (ad esempio `Period.ofNanos(2222)`).

Garantire l'inizializzazione coerente degli oggetti

Nel caso in cui abbiamo diversi modi di inizializzare un oggetto che condividono tutti una parte comune (ovvero qualunque modo scegliamo, parte dell'inizializzazione deve essere sempre eseguita allo stesso modo), avere diversi costruttori che delegano a uno comune sembra una buona idea. Ad esempio, possiamo avere due costruttori, uno che delega all'altro, che mantiene una logica di inizializzazione comune:

```
// common initialization logic
public ProductName(string value)
{
  _value = value;
}

//another constructor that uses the common initialization
public ProductName(string model, string configuration)
 : this(model + " " + configuration) //delegation to "common" constructor
{
}
```

Grazie a ciò, il campo `_value` viene inizializzato in un unico posto e non abbiamo duplicazioni.

Il problema con questo approccio è che questo legame tra costruttori non viene obbligato - possiamo usarlo se vogliamo, altrimenti possiamo saltarlo. Ad esempio, possiamo anche utilizzare un set separato di campi in ciascun costruttore:

```
public ProductName(string value)
{
  _value = value;
}

public ProductName(string model, string configuration)
 //oops, no delegation to the other constructor
{
}
```

il che lascia spazio a errori - potremmo dimenticare di inizializzare sempre tutti i campi e consentire la creazione di oggetti con uno stato non valido.

Ritengo che l'utilizzo di diversi metodi factory statici lasciando un solo costruttore è più sicuro in quanto impone a ogni creazione di oggetto di passare attraverso questo singolo costruttore. Questo costruttore può quindi garantire che tutti i campi dell'oggetto siano inizializzati correttamente. In tal caso non è possibile ignorare questo costruttore in nessuno dei metodi factory statici, ad esempio:

```
public ProductName CombinedOf(string model, string configuration)
{
  // no way to bypass the constructor here,
  // and to avoid initializing the _value field
  return new ProductName(model + " " + configuration);
}
```

Ciò che ho scritto sopra potrebbe sembrare una complicazione inutile in quanto l'esempio dei nomi dei prodotti è banale e difficilmente commetteremo un errore come quello descritto sopra, tuttavia:

1. Ci sono casi più complessi in cui possiamo effettivamente dimenticare di inizializzare alcuni campi in più costruttori.
2. È sempre meglio essere protetti dal compilatore piuttosto che non esserlo quando il prezzo per la protezione è considerevolmente basso. Per lo meno, quando succede qualcosa, avremo un posto in meno dove cercare i bug.

Il posto migliore per la convalida dell'input

Diamo ancora un'occhiata al metodo factory `For()`:

```
public static ProductName For(string value)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    //validation failed
    throw new ArgumentException(
      "Product names must be human readable!");
  }
  else
  {
    //here we call the constructor
    return new ProductName(value);
  }
}
```

e notare che contiene una certa convalida dell'input, mentre il costruttore no. È una decisione saggia spostare la convalida in tale metodo e lasciare il costruttore solo per l'assegnazione dei campi? La risposta a questa domanda dipende dalla risposta a un'altra: ci sono casi in cui non vogliamo convalidare gli argomenti del costruttore? In caso negativo, la convalida dovrebbe andare nel costruttore, poiché il suo scopo è garantire che un oggetto sia inizializzato correttamente.

Apparentemente, ci sono casi in cui vogliamo mantenere le convalide fuori dal costruttore. Consideriamo il caso seguente: vogliamo creare pacchetti di due nomi di prodotto come uno solo. A questo scopo, introduciamo un nuovo metodo su `ProductName`, chiamato `BundleWith()`, che prende un altro nome di prodotto:

```
public ProductName BundleWith(ProductName other)
{
  return new ProductName(
    "Bundle: " + _value + other._value);
}
```

Notare che il metodo `BundleWith()` non contiene alcuna validazione ma chiama semplicemente il costruttore. È sicuro farlo in questo caso perché sappiamo che:

1. La stringa non sarà né nulla né vuota poiché stiamo aggiungendo i valori di entrambi i nomi di prodotto al valore costante di `"Bundle: "`. Il risultato di tale operazione di accodamento non ci darà mai una stringa vuota o un `null`.
2. I campi `_value` di entrambi i componenti del nome di prodotto `this` e `other` devono essere validi perché se non lo fossero, i due nomi di prodotto che contengono tali valori non potrebbero essere creati in primo luogo.

Questo era un caso in cui non avevamo bisogno della convalida perché eravamo sicuri che l'input fosse valido. Potrebbe esserci un altro caso, quando è più conveniente per un metodo factory statico fornire la convalida da solo. Tale validazione può essere più dettagliata e utile poiché si trova in un metodo factory creato per un caso specifico e sa di più su cosa sia questo caso. Consideriamo ad esempio il metodo già visto per combinare il modello e la configurazione nel nome di un prodotto. Se lo guardiamo di nuovo (non contiene ancora alcuna validazione):

```
public ProductName CombinedOf(string model, string configuration)
{
  return ProductName.For(model + " " + configuration);
}
```

Potremmo sostenere che questo metodo trarrebbe vantaggio da un insieme specializzato di convalide perché probabilmente sia il modello che la configurazione devono essere convalidati separatamente (a proposito, a volte può essere una buona idea creare *oggetti valore* anche per il modello e la configurazione - dipende dove li otteniamo e come li usiamo). Potremmo quindi arrivare al punto di lanciare un'eccezione diversa per ciascun caso, ad esempio:

```
public ProductName CombinedOf(string model, string configuration)
{
  if(!IsValidModel(model))
  {
    throw new InvalidModelException(model);
  }

  if(!IsValidConfiguration(configuration))
  {
    throw new InvalidConfigurationException(configuration);
  }

  return ProductName.For(model + " " + configuration);
}
```

Cosa succede se in alcuni casi abbiamo bisogno della convalida di default? Possiamo comunque inserirli in un metodo factory comune e invocarlo da altri metodi factory. Sembra un po' come tornare al problema costruttori multipli, ma direi che questo problema non è così serio - nella mia mente, il problema delle validazioni è più facile da individuare che mancare erroneamente un'assegnazione di campo come nel caso dei costruttori. Si potrebero avere preferenze diverse però.

Da ricordare che sono state poste due domande e ho risposto solo a una di esse. Per fortuna, ora è molto più facile rispondere all'altra domanda, ovvero perché il costruttore è privato e non pubblico.

34.3.2 Perché *private* e non *public*?

Le mie ragioni sono: validazione e separazione dell'uso dalla costruzione.

Validazione

Osservando il costruttore di `ProductName` - abbiamo già discusso del fatto che non convalida il suo input. Questo va bene quando il costruttore viene utilizzato internamente all'interno di `ProductName` (come ho appena dimostrato nella sezione precedente), perché può essere chiamato solo dal codice di cui noi, come creatori della classe `ProductName`, possiamo fidarci. D'altra parte, probabilmente c'è molto codice che creerà istanze di `ProductName`. Parte di questo codice non è nemmeno stato ancora scritto, la maggior parte non la conosciamo, quindi non possiamo fidarci. Per tale codice, vogliamo utilizzare solo i metodi "sicuri" che convalidano l'input e generano errori, non il costruttore.

Separare l'uso dalla costruzione[?]

Ho già detto che la maggior parte delle volte non vogliamo usare il polimorfismo per i valori, poiché non svolgono alcun ruolo che altri oggetti possano ricoprire. Anche se ritengo saggio riservarci un certo grado di flessibilità per poter cambiare più facilmente la nostra decisione in futuro, soprattutto quando il costo della flessibilità è molto basso.

I metodi di factory statici offrono maggiore flessibilità rispetto ai costruttori. Ad esempio, quando abbiamo un metodo factory statico come questo:

```
public static ProductName For(string value)
{
  //validations skipped for brevity
  return new ProductName(value);
}
```

e tutto il nostro codice dipende da esso per la creazione dei nomi dei prodotti piuttosto che dal costruttore, siamo liberi di rendere astratta la classe `ProductName` ad un certo punto e fare in modo che il metodo `For()` restituisca un'istanza di una sottoclasse di `ProductName`. Questa modifica avrebbe un impatto solo su questo metodo statico, poiché il costruttore è

[4] A. Shalloway et al., Essential Skills For The Agile Developer.

nascosto e accessibile solo dall'interno della classe `ProductName`. Ancora una volta, questo è qualcosa che non consiglio di fare per default, a meno che non ci sia una ragione molto forte. Ma se esiste, la capacità di farlo è qui.

34.4 Metodi di conversione delle stringhe

La versione sovrascritta (overridden) di `ToString()` solitamente restituisce il valore mantenuto internamente o la sua rappresentazione in stringa. È utilizzabile per interagire con API di terze parti o altro codice che non riconosce il nostro tipo `ProductName`. Ad esempio, se vogliamo salvare il nome del prodotto all'interno del database, l'API del database non ha idea di `ProductName`, ma accetta piuttosto tipi di libreria come stringhe, numeri, ecc. In tal caso, possiamo utilizzare `ToString()` per rendere possibile il passaggio del nome del prodotto:

```
// let's assume that we have a variable
// productName of type ProductName.

var dataRecord = new DataRecord();
dataRecord["Product Name"] = productName.ToString();

//...

database.Save(dataRecord);
```

Le cose diventano più complicate quando un oggetto valore ha più campi o quando racchiude un altro tipo come `DateTime` o un `int` - potremmo dover implementare altri metodi di accesso per ottenere questi dati. `ToString()` può quindi essere utilizzato per scopi diagnostici per consentire la stampa di dump di dati di facile utilizzo.

A parte il `ToString()` sovrascritto (overridden), il nostro tipo `ProductName` ha un sovraccarico con la firma `ToString(Format format)`. Questa versione di `ToString()` non è ereditata da nessun'altra classe, quindi è un metodo che abbiamo creato per soddisfare i nostri obiettivi. Il nome `ToString()` viene utilizzato solo per comodità, poiché il nome è abbastanza buono per descrivere ciò che fa il metodo e sembra familiare. Il suo scopo è quello di poter formattare il nome del prodotto in modo diverso per output diversi, ad es. report e stampa su schermo. È vero, potremmo introdurre un metodo speciale per ciascuno dei casi (ad esempio `ToStringForScreen()` e `ToStringForReport()`), ma ciò potrebbe far sì che `ProductName` sappia troppo su come viene utilizzato - dovremmo estendere il tipo con nuovi metodi ogni volta che lo si doveva stampare in modo diverso. Invece, `ToString()` accetta un `Format` (che è un'interfaccia, tra l'altro) che ci dà un po' più di flessibilità.

Quando dobbiamo stampare il nome del prodotto sullo schermo, possiamo dire:

```
var name = productName.ToString(new ScreenFormat());
```

e per i report, possiamo dire:

```
var name = productName.ToString(new ReportingFormat());
```

Niente ci obbliga a chiamare questo metodo `ToString()` - possiamo usare un altro nome se lo desideriamo.

34.5 Membri di uguaglianza

Per valori come `ProductName`, dobbiamo implementare tutte le operazioni di uguaglianza più `GetHashCode()`. Lo scopo delle operazioni di uguaglianza è quello di conferire valore semantico ai nomi dei prodotti e consentire le seguenti espressioni:

```
ProductName.For("a").Equals(ProductName.For("a"));
ProductName.For("a") == ProductName.For("a");
```

per restituire `true`, poiché lo stato degli oggetti confrontati è lo stesso nonostante siano istanze separate in termini di riferimenti. In Java, ovviamente, possiamo solo sovrascrivere il metodo `equals()` - non siamo in grado di sovrascrivere gli operatori di uguaglianza poiché il loro comportamento è fissato al confronto dei riferimenti (eccetto i tipi primitivi), ma i programmatori Java sono così abituati a questo che raramente ne fanno un problema.

Una cosa da notare sull'implementazione di `ProductName` è che implementa l'interfaccia `IEquatable<ProductName>`. In C#, eseguire l'override di questa interfaccia per avere una semantica del valore è considerata una buona pratica. L'interfaccia `IEquatable<T>` è ciò che ci obbliga a creare un metodo `Equals()` fortemente tipizzato:

```
public bool Equals(ProductName other);
```

mentre quello ereditato da `object` accetta un `object` come parametro. L'uso e l'esistenza dell'interfaccia `IEquatable<T>` sono per lo più specifici di C#, quindi non entrerò nei dettagli qui, ma si può sempre cercarlo nella documentazione.

Quando sovrascriviamo (override) `Equals()`, anche il metodo `GetHashCode()` deve essere sovrascritto. La regola è che tutti gli oggetti considerati uguali dovrebbero restituire lo stesso codice hash e tutti gli oggetti considerati non uguali dovrebbero restituire codici hash diversi. Il motivo è che i codici hash vengono utilizzati per identificare oggetti nelle tabelle hash o nei set hash - queste strutture dati non funzioneranno correttamente con i valori se `GetHashCode()` non è implementato correttamente. Sarebbe un peccato perché i valori vengono spesso usati come chiavi in vari dizionari basati su hash.

34.6 Il ritorno dell'investimento

Ci sono altri aspetti dei valori che non sono visibili nell'esempio `ProductName`, ma prima di spiegarli nel prossimo capitolo, vorrei considerare un'altra cosa.

Esaminando l'anatomia di `ProductName`, potrebbe sembrare che ci sia molto codice solo per racchiudere una singola stringa. Ne vale la pena? Dovè il ritorno dellinvestimento?

Per rispondere, vorrei tornare al nostro problema originale con i nomi dei prodotti e ricordare che ho introdotto un *oggetto valore* per limitare l'impatto di alcune modifiche che potrebbero verificarsi nel codice in cui vengono utilizzati i nomi dei prodotti. Dato che è passato molto tempo, ecco le modifiche che volevamo influissero il meno possibile sul nostro codice:

1. Modificare il confronto dei nomi dei prodotti senza distinzione tra maiuscole e minuscole
2. Modificare il confronto per prendere in considerazione non solo il nome di un prodotto ma anche la configurazione con cui un prodotto viene venduto.

Andiamo a scoprire se introdurre un *oggetto valore* potrebbe ripagare in questi casi.

34.6.1 Prima modifica - indipendenza dalle maiuscole

Questo è facile da eseguire - dobbiamo solo modificare gli operatori di uguaglianza, le operazioni `Equals()` e `GetHashCode()`, in modo che trattino uguali i nomi con lo stesso contenuto in lettere con altezze diverse. Non esaminerò il codice ora perché non è troppo interessante, spero che si possa immaginare come sarebbe l'implementazione. Dovremmo modificare tutti i confronti tra stringhe per utilizzare un'opzione che ignori maiuscole e minuscole, ad es. `OrdinalIgnoreCase`. Ciò dovrebbe accadere solo all'interno della classe `ProductName` poiché è l'unica che sa cosa significa che due nomi di prodotto siano uguali. Ciò significa che l'incapsulamento che abbiamo introdotto senza la classe `ProductName` ha dato i suoi frutti.

34.6.2 Seconda modifica - identificatore aggiuntivo

Questa è più complessa, ma avere un *oggetto valore* sul posto lo rende comunque molto più semplice rispetto all'approccio basato sulla semplice stringa. Per apportare questa modifica, dobbiamo modificare la creazione della classe `ProductName` affinché accetti un parametro aggiuntivo, chiamato `config`:

```
private ProductName(string value, string config)
{
  _value = value;
  _config = config;
}
```

Notare che questo è un esempio che abbiamo menzionato prima. C'è una differenza, però. Mentre in precedenza presumevamo che non fosse necessario conservare valore e configurazione separatamente all'interno di un'istanza `ProductName` e concatenarli in un'unica stringa durante la creazione di un oggetto, questa volta assumiamo che avremo in seguito bisogno di questa separazione tra nome e configurazione.

Dopo aver modificato il costruttore, il passo successivo è aggiungere ulteriori convalide al metodo factory:

```
public static ProductName CombinedOf(string value, string config)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    throw new ArgumentException(
          "Product names must be human readable!");
  }
  else if(string.IsNullOrWhiteSpace(config))
  {
    throw new ArgumentException(
          "Configs must be human readable!");
  }
  else
  {
    return new ProductName(value, config);
  }
}
```

Da notare che questa modifica richiede cambiamenti in tutto il codice base (perché sono necessari argomenti aggiuntivi per creare un oggetto), tuttavia, questo non è il tipo di cambiamento di cui abbiamo troppa paura. Questo perché la modifica della signature [*firma*] del metodo attiverà gli errori del compilatore. Ciascuno di questi errori dovrà essere corretto prima che la compilazione possa passare (possiamo dire che il compilatore ci crea un simpatico elenco di cose da fare e si assicura che affrontiamo ogni singolo elemento di quell'elenco). Ciò significa che non corriamo il rischio di dimenticare di trattare uno dei posti in cui è necessario apportare una modifica. Ciò riduce notevolmente il rischio di violare la legge di Shalloway.

L'ultima parte di questa modifica consiste nel modificare gli operatori di uguaglianza, `Equals()` e `GetHashCode()`, per confrontare le istanze non solo per nome, ma anche per configurazione. E ancora, lascerò il codice di questi metodi come esercizio al lettore. Noterò solo brevemente non sarà richiesta alcuna modifica al di fuori della classe `ProductName`.

34.7 Riepilogo

Finora abbiamo parlato di *oggetti valore* utilizzando un esempio specifico di nomi di prodotti. Spero che ora si abbia un'idea di come tali oggetti possano essere utili. Il prossimo capitolo completerà la descrizione degli *oggetti valore* spiegando alcune delle loro proprietà generali.

CAPITOLO 35

Aspetti del design degli *oggetti valore*

Nell'ultimo capitolo abbiamo esaminato l'anatomia di un *oggetto valore*. Tuttavia, ci sono molti altri aspetti del design degli *oggetti valore* che devo ancora menzionare per darvi un quadro completo.

35.1 Immutabilità

Ho menzionato prima che gli *oggetti valore* sono generalmente immutabili. Alcuni dicono che l'immutabilità è la parte fondamentale del fatto che qualcosa sia un valore (ad esempio Kent Beck arriva addirittura a dire che 1 è sempre 1 e non diventerà mai 2), mentre altri non lo considerano un vincolo rigido. In un modo o nell'altro, progettare *oggetti valore* come immutabili mi è servito eccezionalmente bene al punto che non prendo nemmeno in considerazione la possibilità di scrivere classi di *oggetti valore* che siano mutabili. Consentitemi di descrivere tre dei motivi per cui considero l'immutabilità un vincolo chiave per gli *oggetti valore*.

35.1.1 Modifica accidentale del codice hash

Molte volte, i valori vengono utilizzati come chiavi nelle mappe hash (ad esempio il `Dictionary<K,V>` di .NET è essenzialmente una mappa hash). Immaginiamo di avere un dizionario indicizzato con istanze di un tipo chiamato `KeyObject`:

```
Dictionary<KeyObject, AnObject> _objects;
```

Quando utilizziamo un `KeyObject` per inserire un valore in un dizionario:

```
KeyObject key = ...
_objects[key] = anObject;
```

il suo codice hash viene calcolato e memorizzato separatamente dalla chiave originale.

Quando leggiamo dal dizionario usando la stessa chiave:

```
AnObject anObject = _objects[key];
```

viene ricalcolato il suo codice hash e solo quando questi corrispondono gli oggetti risultano uguali.

Pertanto, per recuperare con successo un oggetto da un dizionario con una chiave, questo oggetto chiave deve soddisfare le seguenti condizioni relative alla chiave utilizzata in precedenza per inserire l'oggetto:

1. Il metodo `GetHashCode()` della chiave utilizzata per recuperare l'oggetto deve restituire lo stesso codice hash della chiave utilizzata per inserire l'oggetto durante l'inserimento,

2. Il metodo `Equals()` deve indicare che sia la chiave utilizzata per inserire l'oggetto sia la chiave utilizzata per recuperarlo sono uguali.

La conclusione è: se una qualsiasi delle due condizioni non è soddisfatta, non possiamo aspettarci di ottenere l'oggetto che abbiamo inserito.

Nel capitolo precedente ho accennato al fatto che il codice hash di un *oggetto valore* viene calcolato in base al suo stato. Una conclusione da ciò è che ogni volta che cambiamo lo stato di un *oggetto valore*, cambia anche il suo codice hash. Quindi, supponiamo che il nostro `KeyObject` consenta di modificare il suo stato, ad es. utilizzando un metodo `SetName()`. Pertanto, possiamo fare quanto segue:

```
KeyObject key = KeyObject.With("name");
_objects[key] = new AnObject();

// we mutate the state:
key.SetName("name2");

//do we get the inserted object or not?
var objectIInsertedTwoLinesAgo = _objects[key];
```

Ciò genererà una `KeyNotFoundException` (questo è il comportamento del dizionario quando è indicizzato con una chiave che non contiene), poiché il codice hash quando si recupera l'elemento è diverso da quando lo si inserisce. Modificando lo stato della `key` con l'istruzione: `key.SetName("name2");`, ne ho cambiato anche il codice hash calcolato, quindi quando ho chiesto l'oggetto precedentemente inserito con `_objects[val]`, ho provato ad accedere a una posizione completamente diversa nel dizionario rispetto a quella in cui è archiviato il mio oggetto.

Poiché trovo che sia una situazione abbastanza comune che gli *oggetti valore* finiscano come chiavi all'interno dei dizionari, preferisco lasciarli immutabili per evitare brutte sorprese.

35.1.2 Modifica accidentale da codice esterno

Scommetto che molti di coloro che programmano o codificano in Java conoscono la sua classe `Date`. `Date` si comporta come un valore (ha overloaded [*sovraccaricato*] l'uguaglianza e la generazione del codice hash), ma è mutabile (con metodi come `setMonth()`, `setTime()`, `setHours()` ecc.).

In genere, gli *oggetti valore* tendono ad essere passati molto in un'applicazione e utilizzati nei calcoli. Molti programmatori Java hanno esposto almeno una volta un valore `Date` utilizzando un getter:

```
public class ObjectWithDate {

  private final Date date = new Date();

  //...

  public Date getDate() {
    //oops...
    return this.date;
  }
}
```

Il metodo `getDate()` consente agli utenti della classe `ObjectWithDate` di accedere alla data. Ma ricordare, un oggetto data è mutabile e un getter restituisce un riferimento! Chiunque chiami il getter ottiene l'accesso all'istanza memorizzata internamente di `Date` e può modificarla in questo modo:

```
ObjectWithDate o = new ObjectWithDate();

o.getDate().setTime(date.getTime() + 10000); //oops!

return date;
```

Naturalmente, quasi nessuno lo farebbe nella stessa riga come nello snippet sopra, ma di solito si accede a questa data, viene assegnata a una variabile e quindi passata attraverso diversi metodi, uno dei quali fa qualcosa del genere:

```
public void doSomething(Date date) {
  date.setTime(date.getTime() + 10000); //oops!
  this.nextUpdateTime = date;
}
```

Ciò ha portato a situazioni impreviste poiché gli oggetti data sono stati modificati accidentalmente molto, molto lontano dal luogo in cui sono stati prelevati[1].

Poiché la maggior parte delle volte non era questa l'intenzione, il problema della mutabilità della data ci costringeva a creare manualmente una copia ogni volta che il codice restituiva una data:

```
public Date getDate() {
  return (Date)this.date.clone();
}
```

che molti di noi tendono a dimenticare. Questo approccio della clonazione, tra l'altro, potrebbe aver introdotto una penalizzazione delle prestazioni perché gli oggetti venivano clonati ogni volta, anche quando il codice che chiamava `getDate()` non aveva intenzione di modificare la data[2].

Anche se seguiamo il suggerimento di evitare i getter, lo stesso capita quando la nostra classe supera la data da qualche parte. Osserviamo il corpo di un metodo, chiamato `dumpInto()`:

```
public void dumpInto(Destination destination) {
  destination.write(this.date); //passing reference to mutable object
}
```

In questo caso, la `destination` può modificare la data che riceve in qualsiasi modo, il che, ancora una volta, di solito va contro le intenzioni degli sviluppatori.

Ho riscontrato moltissimi problemi nel codice di produzione causati dalla mutabilità del solo tipo Java `Date`. Questo è uno dei motivi per cui la nuova libreria time in Java 8 (`java.time`) contiene tipi immutabili per ora e data. Quando un tipo è immutabile, se ne può tranquillamente restituire l'istanza o passarla da qualche parte senza doversi preoccupare che qualcuno sovrascriva lo stato locale contro la nostra volontà.

35.1.3 Thread safety

Quando i valori mutabili vengono condivisi tra thread, esiste il rischio che vengano modificati da più thread contemporaneamente o modificati da un thread mentre vengono letti da un altro. Ciò può corrompere i dati. Come ho già detto, gli *oggetti valore* tendono a essere creati molte volte in molti posti e passati all'interno di metodi o restituiti spesso come risultati - questa sembra essere la loro natura. Pertanto, aumenta questo rischio di corruzione o incoerenza dei dati.

Si immagini che il codice abbia preso possesso di un *oggetto valore* di tipo `Credentials`, contenente nome utente e password. Inoltre, supponiamo che gli oggetti `Credentials` siano modificabili. In tal caso, un thread potrebbe modificare accidentalmente l'oggetto mentre viene utilizzato da un altro thread, causando un'incoerenza dei dati. Pertanto, a condizione che sia necessario trasmettere login e password separatamente a un meccanismo di sicurezza di terze parti, potremmo riscontrare quanto segue:

```
public void LogIn(Credentials credentials)
{
  thirdPartySecuritySystem.LogIn(
    credentials.GetLogin(),
    //imagine password is modified before the next line
    //from a different thread
    credentials.GetPassword())
}
```

[1] Questo a volte viene chiamato "bug di aliasing": https://martinfowler.com/bliki/AliasingBug.html
[2] A meno che Java non lo ottimizzi in qualche modo, ad es. utilizzando l'approccio copy-on-write.

D'altra parte, quando un oggetto è immutabile, non ci sono problemi di multithreading. Se un dato è di sola lettura, può essere letto in sicurezza da tutti i thread che desideriamo. Dopotutto, nessuno può modificare lo stato di un oggetto, quindi non c'è possibilità di incoerenza[3].

35.1.4 Se non la mutabilità, allora cosa?

Per le ragioni che ho descritto, considero limmutabilità un aspetto cruciale della progettazione degli *oggetti valore* e in questo libro, quando parlo di *oggetti valore*, presumo che siano immutabili.

Tuttavia, c'è una domanda che rimane senza risposta: che dire di una situazione in cui ho bisogno di:

- sostituire tutte le occorrenze della lettera 'r' in una stringa con la lettera 'l'?
- spostare una data in avanti di cinque giorni?
- aggiungere un nomefile a un path assoluto di directory per formare un percorso assoluto del file (ad esempio "C:" + "myFile.txt" = "C:\myFile.txt")?

Se non mi è consentito modificare un valore esistente, come posso raggiungere tali obiettivi?

La risposta è semplice - gli *oggetti valore* hanno operazioni che, invece di modificare l'oggetto esistente, ne restituiscono uno nuovo, con lo stato che ci aspettiamo. Il vecchio valore rimane invariato. Questo è il modo ad es. di come le stringhe si comportano in Java e C#.

Giusto per affrontare i tre esempi che ho citato

- quando ho una `string` esistente e voglio sostituire ogni occorrenza della lettera `r` con la lettera `l`:

```
string oldString = "rrrr";
string newString = oldString.Replace('r', 'l');
//oldString is still "rrrr", newString is "llll"
```

- Quando voglio spostare una data in avanti di cinque giorni:

```
DateTime oldDate = DateTime.Now;
DateTime newString = oldDate + TimeSpan.FromDays(5);
//oldDate is unchanged, newDate is later by 5 days
```

- Quando voglio aggiungere un filename a un path di directory per formare un percorso assoluto del file[4]:

```
AbsoluteDirectoryPath oldPath
  = AbsoluteDirectoryPath.Value(@"C:\Directory");
AbsoluteFilePath newPath = oldPath + FileName.Value("file.txt");
//oldPath is "C:\Directory", newPath is "C:\Directory\file.txt"
```

Quindi, ancora, ogni volta che vogliamo avere un valore basato su un valore precedente, invece di modificare l'oggetto precedente, creiamo un nuovo oggetto con lo stato desiderato.

35.1.5 I trucchi dellimmutabilità

Attenzione al costruttore!

Tornando all'esempio `Date` di Java - si potrebbe pensare che sia abbastanza facile abituarsi a casi come quello ed evitarli semplicemente stando più attenti, ma lo trovo difficile a causa di molti trabocchetti associati all'immutabilità in linguaggi come come C# o Java. Ad esempio, un'altra variante del caso `Date` da Java potrebbe essere qualcosa del genere: Immaginiamo di avere un tipo `Money`, che è definito come::

```
public sealed class Money
{
  private readonly int _amount;
```

(continues on next page)

[3] Questo è uno dei motivi per cui i linguaggi funzionali, in cui i dati sono immutabili per default, ottengono molta attenzione nei settori in cui è necessario fare molte cose in parallelo.

[4] questo esempio utilizza una libreria chiamata Atma Filesystem: https://www.nuget.org/packages/AtmaFilesystem/

(continua dalla pagina precedente)

```
  private readonly Currencies _currency;
  private readonly List<ExchangeRate> _exchangeRates;

  public Money(
    int amount,
    Currencies currency,
    List<ExchangeRate> exchangeRates)
  {
    _amount = amount;
    _currency = currency;
    _exchangeRates = exchangeRates;
  }

  //... other methods
}
```

Notare che questa classe ha un campo di tipo `List<>`, che è esso stesso mutabile. Ma immaginiamo anche di aver rivisto attentamente tutti i metodi di questa classe in modo che questi dati mutabili non vengano esposti. Significa che siamo al sicuro?

La risposta è: finché il nostro costruttore resta così com'è, no. Da notare che il costruttore prende una lista mutabile e la assegna semplicemente a un campo privato. Quindi, qualcuno potrebbe fare qualcosa del genere:

```
List<ExchangeRate> rates = GetExchangeRates();

Money dollars = new Money(100, Currencies.USD, rates);

//modify the list that was passed to dollars object
rates.Add(GetAnotherExchangeRate());
```

Nell'esempio sopra, l'oggetto `dollars` è stato modificato cambiando la lista che gli era stata passata al suo interno. Per ottenere l'immutabilità, si dovrebbe utilizzare una libreria di collection immutabili o modificare la seguente riga:

```
_exchangeRates = exchangeRates;
```

in:

```
_exchangeRates = new List<ExchangeRate>(exchangeRates);
```

Le dipendenze ereditabili possono sorprendere!

Un altro problema ha a che fare con oggetti di tipo che possono essere sotto-classati (cioè non sono `sealed` [*sigillati*]). Diamo un'occhiata all'esempio di una classe chiamata `DateWithZone`, che rappresenta una data con un fuso orario. Diciamo che questa classe ha una dipendenza da un'altra classe chiamata `ZoneId` ed è definita come tale:

```
public sealed class DateWithZone : IEquatable<DateWithZone>
{
  private readonly ZoneId _zoneId;

  public DateWithZone(ZoneId zoneId)
  {
    _zoneId = zoneId;
  }

  //... some equality methods and operators...

  public override int GetHashCode()
```

(continues on next page)

(continua dalla pagina precedente)

```
  {
    return (_zoneId != null ? _zoneId.GetHashCode() : 0);
  }

}
```

Notare che per semplicità ho fatto in modo che il tipo `DateWithZone` sia composto *solo* dall'id della zona, il che ovviamente in realtà non ha alcun senso. Lo faccio solo perché voglio che questo esempio sia ridotto all'osso. Questo è anche il motivo per cui, ai fini di questo esempio, il tipo `ZoneId` è definito semplicemente come:

```
public class ZoneId
{

}
```

Ci sono due cose da notare su questa classe. Innanzitutto, ha un corpo vuoto, quindi nessun campo e metodo definito. La seconda cosa è che questo tipo non è `sealed` (OK, la terza cosa è che questo tipo non ha una semantica di valore, poiché le sue operazioni di uguaglianza sono ereditate come basate su riferimenti dalla classe `Object` ma, sempre per semplificare, ignoriamolo).

Ho appena detto che `ZoneId` non ha campi e metodi, vero? Beh, ho mentito. Una classe in C# eredita da `Object`, il che significa che eredita implicitamente alcuni campi e metodi. Uno di questi metodi è `GetHashCode()`, il che significa che viene compilato il seguente codice:

```
var zoneId = new ZoneId();
Console.WriteLine(zoneId.GetHashCode());
```

L'ultima informazione di cui abbiamo bisogno per vedere il quadro più ampio è che metodi come `Equals()` e `GetHashCode()` possono essere sovrascritti. Questo, combinato con il fatto che il nostro `ZoneId` non è `sealed`, significa che qualcuno può fare qualcosa del genere:

```
public class EvilZoneId : ZoneId
{
  private int _i = 0;

  public override GetHashCode()
  {
    _i++;
    return i;
  }
}
```

Quando si chiama `GetHashCode()` su un'istanza di questa classe più volte, restituirà 1,2,3,4,5,6,7... e così via. Questo perché il campo `_i` è un pezzo di stato mutabile e viene modificato ogni volta che richiediamo un codice hash. Ora, presumo che nessuna persona sana di mente scriverebbe un codice come questo, ma d'altra parte, il linguaggio non lo limita. Quindi, supponendo che una classe così malvagia venga alla luce in un codice che utilizza `DateWithZone`, vediamo quali potrebbero essere le conseguenze.

Innanzitutto, immaginiamo che qualcuno faccia quanto segue:

```
var date = new DateWithZone(new EvilZoneId());

//...

DoSomething(date.GetHashCode());
DoSomething(date.GetHashCode());
DoSomething(date.GetHashCode());
```

Notare che l'utente dell'istanza `DateWithZone` utilizza il suo codice hash, ma l'operazione `GetHashCode()` di questa classe è implementata come:

```
public override int GetHashCode()
{
  return (_zoneId != null ? _zoneId.GetHashCode() : 0);
}
```

Quindi utilizza il codice hash dell'ID della zona, che, nel nostro esempio, è della classe `EvilZoneId` che è mutabile. Di conseguenza, anche la nostra istanza di `DateWithZone` finisce per essere modificabile.

Questo esempio mostra un caso banale e non troppo credibile di `GetHashCode()` perché volevo mostrare che anche le classi vuote hanno alcuni metodi che possono essere sovrascritti per rendere gli oggetti mutabili. Per assicurarci che la classe non possa essere una sottoclasse di una classe mutabile, dovremmo rendere tutti i metodi `sealed` (compresi quelli ereditati da `Object`) o, meglio, rendere la classe `sealed`. Un'altra osservazione che si può fare è che se il nostro `ZoneId` fosse una classe astratta con almeno un metodo astratto, non avremmo alcuna possibilità di garantire l'immutabilità delle sue implementazioni, poiché per definizione esistono metodi astratti da implementare in sottoclassi, quindi non è possibile creare un metodo o una classe astratta `sealed`.

Un altro modo per prevenire la mutabilità da parte delle sottoclassi è rendere il costruttore della classe `private`. Le classi con costruttori privati possono ancora essere sottoclassi, ma solo da classi nidificate, quindi c'è un modo per l'autore della classe originale di controllare l'intera gerarchia e assicurarsi che nessuna operazione muti alcuno stato.

Ci sono altri trucchi (ad esempio uno simile applicato ai tipi generici), ma li lascerò per un'altra volta.

35.2 Gestione della variabilità

Come negli oggetti comuni, anche nel mondo dei *valori* può esserci una certa variabilità. Ad esempio, il denaro può essere dollari, sterline, zloty (moneta polacca), euro, ecc. Un altro esempio di qualcosa che può essere modellato come *valore* sono i valori dei path (ad esempio, `C:\Directory\file.txt` o `/usr/bin/sh`) -- possono esserci percorsi assoluti, percorsi relativi, percorsi per file e percorsi che puntano a directory, possiamo avere path Unix e path Windows.

A differenza degli oggetti ordinari, tuttavia, dove abbiamo risolto la variabilità utilizzando interfacce e implementazioni diverse (ad esempio avevamo un'interfaccia `Alarm` con classi di implementazione come `LoudAlarm` o `SilentAlarm`), nei valori del mondo lo facciamo in modo diverso. Prendendo come esempio gli allarmi che ho appena citato, possiamo dire che le diverse tipologie di allarme variavano nel modo in cui assolvevano alla responsabilità di segnalare l'avvenuta attivazione (abbiamo detto che rispondevano allo stesso messaggio -- con risposte a volte del tutto diverse -- e comportamenti diversi). La variabilità nel mondo dei *valori* tipicamente non è comportamentale come nel caso degli oggetti. Consideriamo i seguenti esempi:

1. Il denaro può essere dollari, sterline, zloty, ecc., e i diversi tipi di valute differiscono per i tassi di cambio applicati (ad esempio "quanti dollari ottengo da 10 euro e quanti da 10 sterline?"), che non è una distinzione comportamentale. Pertanto, il polimorfismo non si adatta a questo caso.
2. I path possono essere assoluti e relativi e puntano a file e directory. Differiscono nelle operazioni che possono essere applicate loro. Per esempio. possiamo immaginare che per i path che puntano a file, possiamo avere un'operazione chiamata `GetFileName()`, che non ha senso per un path che punta a una directory. Sebbene si tratti di una distinzione comportamentale, non possiamo dire che il "path della directory" e il "path del file" siano varianti della stessa astrazione, piuttosto che si tratta di due astrazioni diverse. Pertanto, anche in questo caso il polimorfismo non sembra essere la risposta.
3. A volte, potremmo voler avere una distinzione comportamentale, come nell'esempio seguente. Abbiamo una *classe valore* che rappresenta i nomi dei prodotti e vogliamo scrivere in diversi formati a seconda della situazione.

Come modelliamo questa variabilità? Solitamente considero tre approcci fondamentali, ciascuno applicabile in contesti diversi:

- implicito - che si applicherebbe all'esempio del denaro,
- esplicito - che si adatterebbe bene al caso dei path,
- delegato - che si adatterebbe al caso dei nomi di prodotto.

Permettetemi di darvi una descrizione più ampia di ciascuno di questi approcci.

35.2.1 Variabilità implicita

Torniamo all'esempio della modellazione del denaro utilizzando *oggetti valore*[5]. Il denaro può avere valute diverse, ma non vogliamo trattare ciascuna valuta in un modo speciale. Le uniche cose che sono influenzate dalla valuta sono i tassi con cui li scambiamo con altre valute. Vogliamo che il resto del nostro programma non sappia con quale valuta ha a che fare in quel momento (potrebbe anche funzionare con più valori, ciascuno di valuta diversa, contemporaneamente, durante un calcolo o un'altra operazione commerciale).

Questo ci porta a rendere implicite le differenze tra valute, ovvero avremo un unico tipo chiamato `Money`, che non esporrà affatto la propria valuta. Dobbiamo solo dire qual è la valuta quando creiamo un'istanza:

```
Money tenPounds = Money.Pounds(10);
Money tenBucks = Money.Dollars(10);
Money tenYens = Money.Yens(10);
```

e quando vogliamo conoscere l'importo concreto in una determinata valuta:

```
//doesn't matter which currency it is, we want dollars.
decimal amountOfDollarsOnMyAccount = mySavings.AmountOfDollars();
```

a parte questo, possiamo mescolare diverse valute quando e dove vogliamo[6]:

```
Money mySavings =
  Money.Dollars(100) +
  Money.Euros(200) +
  Money.Zlotys(1000);
```

Questo approccio funziona presupponendo che tutta la nostra logica sia comune per tutti i tipi di denaro e non abbiamo alcuna logica speciale solo per le sterline o solo per gli euro in cui non vogliamo trasferire altre valute per errore[7].

Per riassumere, abbiamo progettato il tipo `Money` in modo che la variabilità della valuta sia implicita - la maggior parte del codice semplicemente non ne è consapevole e viene gestita bene all'interno della classe `Money`.

35.2.2 Variabilità esplicita

Ci sono momenti, tuttavia, in cui vogliamo che la variabilità sia esplicita, cioè modellata utilizzando tipi diversi. I percorsi del filesystem sono un buon esempio.

Per cominciare, immaginiamo di avere il seguente metodo per creare archivi di backup che accetta un path di destinazione (per ora come una stringa - arriveremo agli oggetti path in seguito) come parametro di input:

```
void Backup(string destinationPath);
```

Questo metodo presenta un ovvio inconveniente - la sua firma non dice nulla sulle caratteristiche del path di destinazione, il che solleva alcune domande:

- Dovrebbe essere un percorso assoluto o relativo. Se relativo, allora rispetto a cosa?
- Il path dovrebbe contenere un nome file per il file di backup o dovrebbe essere solo un percorso di directory e un nome file verrà fornito in base a un qualche tipo di pattern (ad esempio una parola "backup" + il timestamp corrente)?
- O forse il nome del file nel percorso è facoltativo e se non ne viene fornito uno, viene utilizzato un nome di default?

Queste domande suggeriscono che il design attuale non trasmette l'intenzione in modo sufficientemente esplicito. Possiamo provare ad aggirare il problema cambiando il nome del parametro per suggerire i vincoli, in questo modo:

[5] Questo esempio è vagamente basato sul libro Test-Driven Development By Example di Kent Beck.

[6] Potrei usare metodi di estensione per rendere l'esempio ancora più idiomatico, ad es. essere in grado di scrivere `5.Dollars()`, ma non voglio andare troppo lontano nella terra degli idiomi specifici di qualsiasi linguaggio, perché il mio obiettivo è un pubblico più ampio dei soli programmatori C#.

[7] Sono consapevole che questo esempio sembra un po' ingenuo - dopo tutto, aggiungere denaro in diverse valute implicherebbe che queste debbano essere convertite in un'unica valuta e si applicherebbero i tassi di cambio, il che potrebbe farci perdere denaro. Kent Beck ha riconosciuto e risolto questo problema nel suo libro Test-Driven Development By Example - se si è interessati si deve dare un'occhiata alla sua soluzione.

```
void Backup(string absoluteFilePath);
```

ma l'efficacia di ciò si basa esclusivamente sul fatto che qualcuno legga il nome dell'argomento e inoltre, prima che un path (passato come stringa) raggiunga questo metodo, di solito viene passato più volte ed è molto difficile tenere traccia di cosa c'è dentro questa stringa, quindi diventa facile fare confusione e passare ad es. un percorso relativo laddove se ne prevede uno assoluto. Il compilatore non impone alcun vincolo. Inoltre, è possibile passare un argomento che non sia nemmeno un percorso, perché una `string` può contenere qualsiasi contenuto arbitrario.

Sembra una buona situazione introdurre un *oggetto valore*, ma di che tipo o tipi dovremmo introdurre? Sicuramente, potremmo creare un singolo tipo chiamato `Path`[8] che abbia metodi come `IsAbsolute()`, `IsRelative()`, `IsFilePath()` e `IsDirectoryPath()` (cioè gestirà la variabilità implicita), che risolverebbe (solo - lo vedremo tra poco) una parte del problema - la signature [*firma*] sarebbe:

```
void Backup(Path absoluteFilePath);
```

e non saremmo in grado di passare una stringa arbitraria, solo un'istanza di un `Path`, che potrebbe esporre un metodo factory che controlla se la stringa passata è nel formato corretto:

```
//the following could throw an exception
//because the argument is not in a proper format
Path path = Path.Value(@"C:\C:\C:\C:\//\/\/");
```

Un metodo factory di questo tipo potrebbe generare un'eccezione al momento della creazione dell'oggetto path. Questo è importante - in precedenza, quando non avevamo l'*oggetto valore*, potevamo assegnare spazzatura a una stringa, passarla tra diversi oggetti e ottenere un'eccezione dal metodo `Backup()`. Ora che abbiamo modellato i path come *oggetti valore*, c'è un'alta probabilità che il tipo `Path` venga utilizzato il prima possibile nella catena di chiamate. Grazie a questo e alla validazione all'interno del metodo factory, otterremo un'eccezione molto più vicino al luogo in cui è stato commesso l'errore, non alla fine della catena di chiamate.

Quindi sì, l'introduzione di un *oggetto valore* `Path` generale potrebbe risolvere alcuni problemi, ma non tutti. Tuttavia, la firma del metodo `Backup()` non segnala che il path previsto deve essere un percorso assoluto per un file, quindi è possibile passare un percorso relativo o uno per una directory, anche se è presente solo un tipo di path accettabile.

In questo caso, le diverse proprietà dei path non sono solo un ostacolo, un problema da risolvere, come nel caso del denaro. Sono il fattore chiave di differenziazione nella scelta se un comportamento è appropriato o meno per un valore. In tal caso, ha molto senso creare diversi *tipi valore* diversi, ciascuno dei quali rappresenta un diverso insieme di vincoli del percorso.

Pertanto, potremmo decidere di introdurre tipi come[9]:

- `AbsoluteFilePath` - rappresenta un percorso assoluto contenente un nome file, ad es. `C:\Dir\file.txt`
- `RelativeFilePath` - rappresenta un percorso relativo contenente un nome file, ad es. `Dir\file.txt`
- `AbsoluteDirPath` - rappresenta un percorso assoluto che non contiene un nome file, ad es. `C:\Dir\`
- `RelativeDirPath` - rappresenta un percorso relativo che non contiene un nome file, ad es. `Dir\`

Avendo tutti questi tipi, ora possiamo cambiare la firma del metodo `Backup()` in:

```
void Backup(AbsoluteFilePath path);
```

Notare che non dobbiamo spiegare i vincoli con il nome dell'argomento - possiamo semplicemente chiamarlo `path` perché il tipo dice già ciò che deve essere detto. E comunque nessuno potrà passare, ad es. un `RelativeDirPath` ora per sbaglio, per non parlare di una stringa arbitraria.

Rendere esplicita la variabilità tra i valori creando tipi separati di solito ci porta a introdurre alcuni metodi di conversione tra questi tipi dove tale conversione è legale. Ad esempio, quando tutto ciò che abbiamo è un `AbsoluteDirPath`, ma vogliamo comunque invocare il metodo `Backup()`, dobbiamo convertire il nostro percorso in un `AbsoluteFilePath` aggiungendo un nome file, che può essere rappresentato da un *oggetto valore* stesso (chiamiamo la classe `FileName`). In

[8] Questo è ciò che ha fatto Java. Non dico che i progettisti Java abbiano preso una decisione sbagliata - una singola classe `Path` è probabilmente molto più versatile. L'unica cosa che sto dicendo è che questo design non è ottimale per il nostro scenario particolare.

[9] per riferimento, consultare https://www.nuget.org/packages/AtmaFilesystem/

C#, possiamo utilizzare l'overload degli operatori per alcune conversioni, ad es. l'operatore + andrebbe bene per aggiungere un nome di file al path di una directory. Il codice che esegue la conversione sarebbe quindi simile a questo:

```
AbsoluteDirPath dirPath = ...
...
FileName fileName = ...
...
//'+' operator is overloaded to handle the conversion:
AbsoluteFilePath filePath = dirPath + fileName;
```

Naturalmente, creiamo metodi di conversione solo laddove abbiano senso nel dominio che stiamo modellando. Non inseriremo un metodo di conversione all'interno di `AbsoluteDirectoryPath` che lo combini con un altro `AbsoluteDirectoryPath`[10].

35.2.3 Variabilità delegata

Infine, possiamo ottenere la variabilità delegando il comportamento variabile a un'interfaccia e facendo in modo che l'*oggetto valore* accetti un'implementazione dell'interfaccia come parametro del metodo. Un esempio di ciò potrebbe essere la classe `Product` del capitolo precedente in cui era dichiarato il seguente metodo:

```
public string ToString(Format format);
```

dove `Format` era un'interfaccia e abbiamo passato diverse implementazioni di questa interfaccia a questo metodo, ad es. `ScreenFormat` o `ReportingFormat`. Da notare che avere il `Format` come parametro del metodo invece di ad es. un parametro costruttore ci consente di sostenere la semantica del valore perché `Format` non è parte dell'oggetto ma piuttosto un "helper guest". Grazie a questo siamo liberi da dilemmi con "il nome 'laptop' formattato per lo schermo è uguale a 'laptop' formattato per un report?"

35.2.4 Riassumendo la discussione implicita, esplicita o delegata

Notare che mentre nel primo esempio (quello con il denaro), rendere implicita la variabilità (in valuta) tra i valori ci ha aiutato a raggiungere i nostri obiettivi di design, nell'esempio del path aveva più senso fare esattamente il contrario: rendere la variabilità (sia in assoluto/relativo che verso file/directory) in modo esplicito come creare un tipo separato per ciascuna combinazione di vincoli.

Se scegliamo l'approccio implicito, possiamo trattare tutte le variazioni allo stesso modo, poiché sono tutte dello stesso tipo. Se decidiamo per l'approccio esplicito, ci ritroveremo con diversi tipi che solitamente sono incompatibili e consentiremo conversioni tra loro laddove tali conversioni abbiano senso. Ciò è utile quando vogliamo che alcune parti del nostro programma siano esplicitamente compatibili con solo una delle varianti.

Devo dire che trovo la variabilità delegata un caso raro (la formattazione della conversione in stringa è un tipico esempio) e durante tutta la mia carriera ho avuto forse una o due situazioni in cui ho dovuto ricorrere ad essa. Tuttavia, alcune librerie utilizzano questo approccio e in un particolare dominio o tipo di applicazione, tali casi potrebbero essere molto più tipici.

35.3 Valori speciali

Alcuni *tipi valore* hanno valori così specifici da avere nomi propri. Ad esempio, un valore stringa costituito da `""` è chiamata "stringa vuota". `2,147,483,647` è chiamata "un valore intero massimo a 32 bit". Questi valori speciali si fanno strada nel design degli *oggetti valore*. Ad esempio, in C#, abbiamo `Int32.MaxValue` e `Int32.MinValue` che sono costanti che rappresentano un valore massimo e minimo di un intero a 32 bit e `string.Empty` che rappresenta una stringa vuota. In Java, abbiamo cose come `Duration.ZERO` per rappresentare una durata zero o `DayOfWeek.MONDAY` per rappresentare un giorno specifico della settimana.

Per tali valori, la pratica comune che ho visto è renderli accessibili a livello globale dalle classi di *oggetti valore*, come avviene in tutti gli esempi precedenti di C# e Java. Questo perché i valori sono immutabili, quindi l'accessibilità globale non danneggia. Ad esempio, possiamo immaginare `string.Empty` implementato in questo modo:

[10] francamente, come nel caso del denaro, la visione dei path che ho qui descritto è un po' ingenua. Tuttavia, questa visione ingenua potrebbe essere tutto ciò di cui abbiamo bisogno nel nostro caso particolare.

```
public sealed class String //... some interfaces here
{
  //...
  public const string Empty = "";
  //...
}
```

Il modificatore `const` aggiuntivo garantisce che nessuno assegni un nuovo valore al campo `Empty`. A proposito, in C# possiamo usare `const` solo per i tipi che hanno valori letterali, come una string o un `int`. Per i nostri oggetti con valore personalizzato, dovremo utilizzare un modificatore `static readonly` (o `static final` nel caso di Java). Per dimostrarlo, torniamo all'esempio della moneta di questo capitolo e immaginiamo di voler avere un valore speciale chiamato `None` per simboleggiare l'assenza di denaro in qualsiasi valuta. Dato che il nostro tipo `Money` non ha valori letterali, non possiamo usare il modificatore `const`, quindi dobbiamo fare qualcosa del genere:

```
public class Money
{
  //...

  public static readonly
    Money None = new Money(0, Currencies.Whatever);

  //...
}
```

Questo modo di dire è l'unica eccezione che conosco rispetto alla regola che vi ho dato diversi capitoli fa riguardo al non utilizzare affatto campi statici. Ad ogni modo, ora che abbiamo questo valore `None`, possiamo usarlo in questo modo:

```
if(accountBalance == Money.None)
{
  //...
}
```

35.4 *Tipi valore* e Tell Don't Ask

Quando ho raccontato della metafora della "rete di oggetti", ho sottolineato che agli oggetti dovrebbe essere detto cosa fare, non chiedere informazioni. Ho anche scritto che se una responsabilità è troppo grande per essere gestita da un singolo oggetto, non dovrebbe cercare di realizzarla da solo, ma piuttosto delegare ulteriormente il lavoro ad altri oggetti inviando loro messaggi. Ho detto che preferibilmente vorremmo avere metodi per lo più `void` che accettino il loro contesto come argomenti.

E i valori? Questa metafora si applica a loro? E se sì, allora come? E che dire di "Tell Don't Ask" [*Dire non chiedere*]?

Innanzitutto, i valori non appaiono esplicitamente nella metafora della rete di oggetti, almeno non sono "nodi" in questa rete. Sebbene in quasi tutti i linguaggi object-oriented, i valori siano implementati utilizzando lo stesso meccanismo degli oggetti - classi[11], li tratto come un tipo di costrutto leggermente diverso con il proprio insieme di regole e vincoli. I valori possono essere passati tra oggetti nei messaggi, ma non parliamo di valori che inviano messaggi da soli.

Una conclusione da ciò potrebbe essere che i valori non dovrebbero essere composti da oggetti (intesi come nodi nella "rete"). I valori dovrebbero essere composti da altri valori (poiché il nostro tipo `Path` aveva una `string` al suo interno), che ne garantisce l'immutabilità. Inoltre, possono occasionalmente accettare oggetti come parametri dei loro metodi (come la classe `ProductName` del capitolo precedente che aveva un metodo `ToString()` che accettava un'interfaccia `Format`), ma questa è più un'eccezione che una regola. In rari casi, devo utilizzare una collection all'interno di un oggetto valore. Le collection in Java e in C# non vengono generalmente trattate come valori, quindi questa è una sorta di eccezione alla regola. Tuttavia, quando utilizzo collection all'interno di *oggetti valore*, tendo a utilizzare quelle immutabili, come ImmutableList.

[11] C# ha struct, che a volte possono tornare utili quando si implementano valori, anche se hanno alcuni vincoli (vedere https://stackoverflow.com/questions/333829/why-cant-i-define-a-default-constructor-for-a-struct-in-net). Inoltre, a partire dalle versioni più recenti, C# contiene dei record che possono essere talvolta utilizzati per eliminare parte del codice boilerplate.

Se le affermazioni di cui sopra sui valori sono vere, significa semplicemente che non ci si può aspettare che i valori siano conformi a "Tell Don't Ask". Certo, vogliamo che incapsulano i concetti del dominio, forniscano un'interfaccia di livello superiore, ecc., quindi lottiamo molto affinché gli *oggetti valore* non diventino semplici strutture dati come quelle che conosciamo dal C, ma la natura dei valori è piuttosto come "pezzi di dati intelligenti" piuttosto che "insiemi astratti di comportamenti".

Pertanto, ci aspettiamo che i valori contengano metodi di query (anche se, come ho detto, il più delle volte miriamo a qualcosa di più astratto e più utile dei semplici metodi "getter"). Ad esempio, potrebbe piacere l'idea di avere un insieme di classi sui path (come `AbsoluteFilePath`), ma alla fine si dovrà interagire in qualche modo con una serie di API di terze parti che non sanno nulla di quelle classi. Quindi, tornerà utile un metodo `ToString()` che restituisce semplicemente il valore mantenuto internamente.

35.5 Riepilogo

Questo conclude l'argomento sugli *oggetti di valore*. Non avrei mai pensato che ci sarebbe stato così tanto da discutere su come credo che dovrebbero essere progettati. Per i lettori interessati a vedere un caso di studio all'avanguardia sugli *oggetti valore*, consiglio di consultare le librerie Noda Time (per C#) e Joda Time (per Java) (o le nuove api time e date di Java 8).

CAPITOLO 36

Oggetti di Trasferimento Dati

Osservando le strutture dati iniziali, Johnny e Benjamin le chiamarono Data Transfer Objects [*Oggetti di Trasferimento Dati*]

Un *Data Transfer Object* è un pattern per descrivere gli oggetti responsabili dello scambio di informazioni tra i confini del processo (TODO: confermare con il libro). Quindi, possiamo avere DTO che rappresentano l'input che il nostro processo riceve e DTO che rappresentano l'output che il nostro processo invia.

Come visto, i DTO sono in genere solo strutture dati. Ciò potrebbe sorprendere, perché ormai da diversi capitoli ho ripetutamente sottolineato come dovremmo unire dati e comportamenti. Non è questo infrangere tutte le regole che ho menzionato?

Sì, lo è e lo è per una buona ragione. Ma prima di spiegarlo, diamo un'occhiata a due modi di disaccoppiamento.

CAPITOLO 37

Disaccoppiamento mediante l'astrazione del comportamento

TODO: questo vale non solo per le interfacce ma anche per le funzioni

Questo è il modo di disaccoppiamento che ho descritto parlando della rete di oggetti. Diamo un'occhiata ad una semplice classe che rappresenta un documento che può essere stampato su una stampante:

```
public class Document
{
    private string _content = string.Empty;

    public Add(string additionalText)
    {
        _content += additionalText;
    }

    public void PrintWith(PopularBrandPrinter printer)
    {
        printer.Print(_content);
    }
}
```

Notare che il metodo `PrintWith()` del documento richiede un'istanza di `PopularBrandPrinter` che, per il bene di questo esempio, è una classe concreta che a sua volta dipende da una libreria specifica di un fornitore di terze parti (supponiamo che sia chiamato `ThirdPartyLib`). Ciò rende anche la nostra classe `Document` indirettamente accoppiata a questa libreria. La direzione delle dipendenze è la seguente:

```
Document --> PopularBrandPrinter --> ThirdPartyLib
```

Perché tale dipendenza è un problema? Dopotutto, la classe `Document` stessa non è a conoscenza direttamente della classe `ThirdPartyLib`? La risposta arriva quando pensiamo ai componenti. La direzione delle dipendenze così com'è ora dice: "`Document` non funzionerà senza `PopularBrandPrinter` che non funzionerà senza `ThirdPartyLib`". Quindi, se volessimo rilasciare una classe `Document` in una libreria separata, ogni utente di questa libreria dovrebbe anche fare riferimento a `ThirdPartyLib`.

Possiamo risolvere questo problema utilizzando un'interfaccia. Tutto quello che dobbiamo fare è fare in modo che `PopularBrandPrinter` implementi un'interfaccia (ad esempio chiamata `Printer`):

```
public interface Printer
{
    void Print(string text);
}

public class PopularBrandPrinter : Printer
{
    //...
}
```

E fare in modo che il nostro `Document` dipenda da quell'interfaccia invece che da `PopularBrandPrinter`:

```
public class Document
{
    private string _content = string.Empty;

    public Add(string additionalText)
    {
        _content += additionalText;
    }

    public void PrintWith(Printer printer)
    {
        printer.Print(_content);
    }
}
```

Ciò modifica la direzione delle dipendenze in fase di compilazione[1] nel modo seguente:

```
Document --> Printer <| -- PopularBrandPrinter --> ThirdPartyLib
```

Da notare che questa volta il `Document` non dipende in alcun modo da `PopularBrandPrinter`. D'altra parte, la `PopularBrandPrinter` dipende dall'interfaccia, perché deve implementarla.

Considerato ciò che abbiamo ottenuto finora, proviamo di nuovo a svolgere il nostro esercizio di partizionamento di questo codice in librerie. Questa volta possiamo farlo mettendo insieme `Document` con l'interfaccia `Printer` in una libreria e `PopularBrandPrinter` in un'altra:

```
|| Document --> Printer || <|-- PopularBrandPrinter --> || ThirdPartyLib ||
```

Ora, la libreria che contiene `PopularBrandPrinter` diventa una dipendenza opzionale - ogni utente di `Document` può decidere se utilizzare `PopularBrandPrinter` o implementare l'interfaccia `Printer` in un altro modo. Altri fornitori possono anche rilasciare le proprie librerie che contengono le proprie implementazioni dell'interfaccia `Printer` tra le quali gli utenti della classe `Document` possono poi scegliere.

Ciò è possibile perché abbiamo reso la classe `Document` indipendente dai dettagli di implementazione di una particolare stampante e dipendente solo dalle firme astratte presenti nell'interfaccia. Pertanto, l'unica fonte di dipendenze aggiuntive per la classe `Document` risulta dalle firme dei metodi dell'interfaccia (a proposito, possiamo esporre i dettagli di implementazione e le dipendenze indesiderate anche in questo modo, di cui ho scritto nei capitoli precedenti).

[1] ed è, in effetti, un esempio di utilizzo del *Dependency Inversion Principle*.

CAPITOLO 38

Disaccoppiamento mediante l'astrazione dei dati

Un altro modo per disaccoppiare il `Document` dai dettagli di implementazione di una particolare stampante è definire un formato di scambio dati intermedio.

Prendiamo l'esempio del documento e della stampante e capovolgiamo la situazione. Diciamo che ora la classe `PopularBrandPrinter` dipende dal `Document`:

```
public class PopularBrandPrinter
{
    private ThirdPartyPrintingDriver _driver
      = new ThirdPartyPrintingDriver();

    public void Print(Document document)
    {
        _driver.Send(document.GetContent());
    }
}
```

e la classe `Document` è definita come:

```
public class Document
{
    private string _content = string.Empty;

    public Add(string additionalText)
    {
        _content += additionalText;
    }

    public string GetContent()
    {
        return _content;
    }
}
```

//TODO

La direzione delle dipendenze è così:

```
PopularBrandPrinter --> Document
                    --> ThirdPartyLibrary
```

Ciò che vorremmo è impacchettare `PopularBrandPrinter` in una libreria separata. Per fare ciò, dobbiamo renderlo indipendente da `Document`. Inoltre vorremmo che Document rimanesse indipendente dalla stampante. Cosa possiamo fare? Possiamo fare in modo che la stampante accetti non un documento, ma una stringa, che è un tipo di dati. Quindi il metodo `Print()` di `Printer` dovrebbe cambiare da:

```
public void Print(Document document)
{
    _driver.Send(document.GetContent());
}
```

in:

```
public void Print(string content)
{
    _driver.Send(content);
}
```

e il codice che utilizza la stampante non sarebbe più simile a questo:

```
var document = //... get the document
var printer = //... get the printer
printer.Print(document);
```

ma piuttosto:

```
var document = //... get the document
var printer = //... get the printer
printer.Print(document.GetContent());
```

Ora abbiamo una situazione in cui la classe `Document` è disaccoppiata da `PopularBrandPrinter` (ha solo un metodo che restituisce il suo contenuto come stringa) e anche `PopularBrandPrinter` è disaccoppiata da `Document` (ha solo un metodo che accetta una `string`). Utilizzando una `string` come formato di scambio dati intermedio, abbiamo disaccoppiato le due classi l'una dall'altra.

Notare che questa tecnica non riguarda l'uso di stringhe. Il disaccoppiamento può essere effettuato utilizzando la nostra struttura dati personalizzata. Il punto è che i dati non incapsulano alcun comportamento (neppure le loro definizioni), quindi non siamo accoppiati ad essi. Quindi, ad esempio, quando il `Document`, invece di aggregare il testo che gli è stato passato, inizia a leggere il contenuto dal database e la sua firma `GetContent()` cambia in un metodo asincrono:

```
var document = //... get the document
var printer = //... get the printer
printer.Print(await document.GetContent());
```

Non è necessario modificare la stampante poiché è accoppiata all'astrazione dei dati, non all'astrazione del comportamento.

Questo può sembrare un disaccoppiamento più forte e migliore rispetto allutilizzo di un comportamento astratto, ma ha anche un prezzo:

38.1 Problemi con il disaccoppiamento orientato ai dati

Innanzitutto il codice che coordina lo scambio tra `Document` e `PopularBrandPrinter` deve diventare più complesso. Come descritto in precedenza, quando una stampante è accoppiata a un documento, la riga di codice che dice alla stampante di stampare assomiglia a questa:

```
printer.Print(document);
```

mentre nel caso in cui la stampante accetti una stringa, il codice di coordinamento deve chiamare anche il metodo `GetContent()`.

Laltro motivo è che questo approccio incentrato sui dati limita anche la nostra capacità di utilizzare il polimorfismo. Per illustrare ciò, vorrei tornare allesempio che abbiamo utilizzato parlando del disaccoppiamento mediante lastrazione del comportamento. Lì avevamo la seguente riga responsabile della stampa:

```
document.PrintWith(printer);
```

Quando il documento è responsabile della decisione su come stampare, potrebbe decidere di non stamparlo affatto. Si immagini che la nostra classe `Document` implementi un'interfaccia come questa:

```
public interface Printable
{
  void Print(Printer printer);
}
```

Quindi potremmo avere un'altra classe che implementa questa interfaccia, ad es. chiamato `IgnoredPrintable` che avrebbe questa implementazione:

```
public class IgnoredPrintable : Printable
{
  public void Print(Printer printer)
  {
    //deliberately left empty
  }
}
```

Utilizzando questa classe, potremmo modificare il comportamento di questo codice:

```
document.PrintWith(printer);
```

senza doverlo cambiare.

Ora torniamo all'approccio incentrato sui dati:

```
printer.Print(document.GetContent());
```

Possiamo usare il polimorfismo in questo caso per ottenere lo stesso risultato? Bene, proviamo. Ancora una volta, immaginiamo di avere un'interfaccia chiamata `Printable`, ma questa volta è strutturata in questo modo:

```
public interface Printable
{
  string GetContent();
}
```

Ancora una volta, vogliamo creare un'implementazione che non stampi nulla, ma poi ci imbattiamo in un problema:

```
public class IgnoredPrintable : Printable
{
  public string GetContent()
  {
    return ????; //!!!
  }
}
```

Notare che quando implementiamo `GetContent()`, dobbiamo restituire *qualcosa*. Questo perché in questo caso le implementazioni di `Printable` non prendono la decisione se stampare o meno. Questa decisione deve essere presa dal codice che coordina la logica della stampa:

```
printer.Print(document.GetContent());
```

Può, ad es. controllare se è null:

```
if(document.GetContent() != null)
{
  printer.Print();
}
```

ma il punto è che non possiamo sfruttare il polimorfismo dei documenti per nascondere questa decisione.

Questo perché, disaccoppiando i documenti dal comportamento di stampa, abbiamo perso la capacità di nascondere l'implementazione di quel comportamento. Perdendo la capacità di nascondere limplementazione, abbiamo perso anche la capacità di nascondere la variabilità del comportamento. Pertanto, questa variabilità deve andare altrove. Ciò può portare allaggregazione di molta complessità nei luoghi che coordinano la logica del caso duso.

1. più lavoro per i coordinatori - TODO un esempio con un if
2. nessun polimorfismo
3. immaginare che la corda fosse mutabile

CAPITOLO 39

Usare entrambi

In realtà, mostrando due esempi ed etichettando il primo come disaccoppiamento centrato sul comportamento e laltro come disaccoppiamento centrato sui dati, ho mentito un po. Il primo esempio li conteneva entrambi. Da un lato, `Document` è stato disaccoppiato dall'implementazione della stampante reale tramite un'interfaccia astratta, dall'altro, `PopularBrandPrinter` è stato disaccoppiato dal documento perché accettava solo i suoi dati. Quindi la vera scelta è spesso tra la proporzione nell'utilizzo di ciascun tipo.

CAPITOLO 40

DTO come meccanismo di disaccoppiamento incentrato sui dati

Per il codice object-oriented, il disaccoppiamento incentrato sul comportamento è quello di default e più importante poiché senza una forte pressione su di esso, avremmo molte più difficoltà a sfruttare il polimorfismo.

Daltra parte, quando si scambiano informazioni tra i confini del processo, ci sono diversi motivi per cui preferiamo il disaccoppiamento incentrato sui dati:

1. le informazioni in genere vanno in formato binario o testuale.
2. Prestazione.
3. Contesti delimitati
4. Nessun pericolo di mutazione dei dati

TODO: l'interfaccia è la definizione di comportamento

In un certo senso lo è ed è a causa del disaccoppiamento. Esistono due modi principali per disaccoppiare.

TODO: ogni sistema è procedurale ai confini.

DTO:

1. Due tipi di disaccoppiamento
    1. con interfacce (descrizioni pure del comportamento, disaccoppiate dai dati)
    2. con i dati (disaccoppia dal comportamento)
2. Poiché è molto difficile trasmettere il comportamento e abbastanza facile trasmettere i dati, in genere i servizi scambiano dati, non comportamenti.
3. I DTO servono per lo scambio di dati tra processi
4. Rappresenta in genere un contratto dati esterno
5. I dati possono essere facilmente serializzati in testo o byte e deserializzati dall'altra parte.
6. I DTO possono contenere valori, sebbene ciò possa essere difficile perché questi valori devono essere serializzabili. Ciò potrebbe porre alcuni vincoli sui nostri tipi di valore a seconda del parser
7. differenze tra DTO e *oggetti valore* (gli *oggetti valore* possono avere un comportamento, i DTO no, sebbene la linea sia sfumata, ad esempio quando un *oggetto valore* fa parte di un DTO. String contiene molti comportamenti ma non sono specifici del dominio). Le DTO possono implementare luguaglianza?
8. Inoltre, per questo motivo, la logica applicativa dovrebbe essere tenuta lontana dai DTO per evitare di accoppiarli a contratti esterni e rendere difficile la serializzazione/deserializzazione.

9. Mappatura e Wrapping
10. Non ci sono mock dei DTO
11. i DTO di input sono di sola lettura e immutabili, ma possono avere i builder

CAPITOLO 41

Parte 3: TDD nel Mondo Object-Oriented

{{keyToDo}} **Stato:** in sviluppo. Sto scrivendo questa parte. Tuttavia, diversi capitoli sono già disponibili per la lettura e abbastanza stabili. Non vedo l'ora di ricevere il feedback!

Finora abbiamo parlato molto del mondo object-oriented, costituito da oggetti che presentano le seguenti proprietà:

1. Gli oggetti si scambiano messaggi utilizzando interfacce e secondo protocolli. Finché i destinatari dei messaggi rispettano queste interfacce e protocolli, gli oggetti mittente non hanno bisogno di sapere chi si trova esattamente dall'altra parte per gestire il messaggio. In altre parole, interfacce e protocolli consentono di disaccoppiare i mittenti dallidentità dei loro destinatari.
2. Gli oggetti sono costruiti tenendo presente l'euristica *Tell Don't Ask*, in modo che ogni oggetto abbia la propria responsabilità e la adempia quando gli viene detto di fare qualcosa, senza rivelare i dettagli di come gestisce questa responsabilità,
3. Gli oggetti, le loro interfacce e i loro protocolli, sono progettati pensando alla componibilità, che ci permette di comporli come comporremmo parti di frasi, creando piccoli linguaggi di livello superiore, in modo da poter riutilizzare gli oggetti che già abbiamo come nostro "vocabolario" e si aggiungono più funzionalità combinandole in nuove "frasi".
4. Gli oggetti vengono creati in luoghi ben separati quelli che utilizzano tali oggetti. Il luogo di creazione dell'oggetto dipende dal ciclo di vita dell'oggetto - potrebbe essere ad es. una factory o una *composition root*.

Il mondo degli oggetti è completato dal mondo dei valori che presentano le seguenti caratteristiche:

1. I valori rappresentano quantità, misure e altri dati discreti che vogliamo nominare, combinare, trasformare e trasmettere. Esempi sono date, stringhe, denaro, durate, path, numeri, ecc.
2. I valori vengono confrontati in base ai dati, non ai riferimenti. Due valori contenenti gli stessi dati sono considerati uguali.
3. I valori sono immutabili - quando vogliamo avere un valore come un altro, ma con un aspetto modificato, creiamo un nuovo valore contenente questa modifica in base al valore precedente e il valore precedente rimane invariato.
4. I valori non si basano (tipicamente) sul polimorfismo - se abbiamo diversi tipi di valore che devono essere utilizzati in modo intercambiabile, la strategia usuale è fornire metodi di conversione espliciti tra tali tipi.

Ci sono momenti in cui scegliere se qualcosa debba essere un oggetto o un valore pone un problema (mi sono imbattuto in situazioni in cui ho modellato lo stesso concetto come valore in un'applicazione e come oggetto in un'altra), quindi non esiste una regola rigida su come scegliere e, inoltre, persone diverse hanno preferenze diverse.

Questo mondo comune è il mondo in cui inseriremo oggetti mock e altre pratiche TDD nella parte successiva.

So che abbiamo messo da parte il TDD per così tanto tempo. Credetemi questo è dovuto al fatto che ritengo fondamentale comprendere i concetti della parte 2 per ottenere i mock giusti.

Gli oggetti mock non sono uno strumento nuovo, tuttavia ci sono ancora molti malintesi su quale sia la loro natura e dove e come si adattino meglio all'approccio TDD. Alcune opinioni sono arrivate al punto di dire che esistono due stili di TDD: uno che utilizza mock (chiamato "mockist TDD" o "London style TDD") e un altro senza di essi (chiamato "classic TDD" o "Chicago style TDD"). Non sostengo questa divisione. Mi piace molto quello che Nat Pryce ha scritto a riguardo:

> (...) Io sostengo che non esistono diversi tipi di TDD. Esistono diverse convenzioni di progettazione e si scelgono le tecniche e gli strumenti di test più appropriati per le convenzioni in cui sta lavorando.

La spiegazione delle "convenzioni di progettazione" da cui sono nati i mock richiedeva di passare attraverso così tante pagine su una visione specifica della progettazione object-oriented. Questa è la visione che si è scelto di sostenere con gli oggetti mock come strumento e come tecnica. Parlare di oggetti mock fuori dal contesto di questa visione mi farebbe sentire come se stessi dipingendo un quadro falso.

Dopo aver letto la parte 3, si capirà come i mock si inseriscono nel codice object-oriented test-driving, come rendere gestibili gli Statement che utilizzano i mock e come alcune delle pratiche introdotte nei capitoli della parte 1 si applicano ai mock. Si sarà anche in grado di testare semplici sistemi object-oriented.

CAPITOLO 42

Oggetti Mock come strumento di test

Ricordate uno dei primi capitoli di questo libro, in cui ho introdotto oggetti mock e ho detto che avevo mentito sul loro vero scopo e natura? Ora che abbiamo molta più conoscenza della progettazione object-oriented (almeno da un punto di vista specifico e supponente al riguardo), possiamo veramente capire da dove provengono i mock e a cosa servono.

In questo capitolo non dirò ancora nulla sul ruolo degli oggetti mock nel codice object-oriented test-driving. Per ora, voglio concentrarmi sulla giustificazione del loro posto nel contesto del test di oggetti scritti nello stile descritto nella parte 2.

42.1 Un esempio a sostegno

Per questo capitolo, userò un esempio semplice. Prima di descriverlo, si deve sapere che non considero questo esempio una vetrina per oggetti mock. I mock brillano dove ci sono interazioni domain-driven [*guidate dal dominio*] tra oggetti e questo esempio non è così - le interazioni qui sono più guidate dall'implementazione. Tuttavia, ho deciso di usarlo comunque perché lo considero qualcosa di facile da capire e abbastanza buono per discutere alcuni meccanismi degli oggetti mock. Nel prossimo capitolo userò lo stesso esempio come illustrazione, ma dopo lo abbandonerò e mi addentrerò in argomenti più interessanti.

L'esempio è una singola classe, chiamata `DataDispatch`, che è responsabile dell'invio dei dati ricevuti a un canale (rappresentato da un'interfaccia `Channel`). Il `Channel` deve essere aperto prima che i dati vengano inviati e chiuso dopo. `DataDispatch` implementa questo requisito. Ecco il codice completo per la classe `DataDispatch`:

```
public class DataDispatch
{
  private Channel _channel;

  public DataDispatch(Channel channel)
  {
    _channel = channel;
  }

  public void Dispatch(byte[] data)
  {
    _channel.Open();
    try
    {
      _channel.Send(data);
    }
```

(continues on next page)

(continua dalla pagina precedente)

```
    finally
    {
      _channel.Close();
    }
  }
}
```

Il resto di questo capitolo si concentrerà sull'analisi dei comportamenti di `DataDispatch` e del loro contesto.

Inizierò a descrivere questo contesto osservando l'interfaccia utilizzata da `DataDispatch`.

42.2 Interfacce

Come mostrato sopra, `DataDispatch` dipende da una singola interfaccia chiamata `Channel`. Ecco la definizione completa di questa interfaccia:

```
public interface Channel
{
  void Open();
  void Send(byte[] data);
  void Close();
}
```

Un'implementazione di `Channel` viene passata al costruttore di `DataDispatch`. In altre parole, `DataDispatch` può essere composto con qualsiasi cosa che implementi l'interfaccia `Channel`. Almeno dal punto di vista del compilatore. Questo perché, come ho accennato nell'ultima parte, affinché due oggetti composti possano lavorare insieme con successo, le interfacce non sono sufficienti. Devono anche stabilire e seguire un protocollo.

42.3 Protocolli

Notare che quando esaminiamo la classe `DataDispatch`, ci sono due protocolli che deve seguire. Li descriverò uno per uno.

42.3.1 Protocollo tra `DataDispatch` e il suo utente

Il primo protocollo è tra `DataDispatch` e il codice che lo utilizza, cioè quello che chiama il metodo `Dispatch()`. Qualcuno, da qualche parte, deve fare quanto segue:

```
dataDispatch.Send(messageInBytes);
```

oppure non ci sarebbe motivo per l'esistenza di `DataDispatch`. Analizzando più a fondo questo protocollo, possiamo notare che `DataDispatch` non richiede troppo ai suoi utenti -- non ha alcun tipo di valore di ritorno. L'unico feedback che fornisce al codice che lo utilizza è rilanciare qualsiasi eccezione sollevata da un canale, quindi il codice utente deve essere preparato a gestire l'eccezione. Notaree che `DataDispatch` non conosce né definisce i tipi di eccezioni che possono essere lanciate. Questa è la responsabilità di una particolare implementazione del canale. Lo stesso vale per decidere in quale condizione deve essere lanciata un'eccezione.

42.3.2 Protocollo tra `DataDispatch` e `Channel`

Il secondo protocollo è tra `DataDispatch` e `Channel`. In questo caso, `DataDispatch` funzionerà con qualsiasi implementazione di `Channel` e gli consente di invocare i metodi di `Channel` un numero specificato di volte in un ordine specificato:

1. Apre il canale -- una volta,
2. Inviare i dati -- una volta,
3. Chiudere il canale -- una volta.

Qualunque sia l'effettiva implementazione dell'interfaccia `Channel` passata a `DataDispatch`, funzionerà presupponendo che questo sia effettivamente il conteggio e l'ordine in cui verranno chiamati i metodi. Inoltre, `DataDispatch` presuppone che sia necessario chiudere il canale in caso di errore durante l'invio dei dati (da qui il blocco `finally` che racchiude l'invocazione del metodo `Close()`).

42.3.3 Due conversazioni

Riassumendo, ci sono due "conversazioni" in cui un oggetto `DataDispatch` è coinvolto quando adempie alle sue responsabilità -- una con il suo utente e una con una dipendenza passata dal suo creatore. Non possiamo specificare queste due conversazioni separatamente poiché l'esito di ciascuna di queste due conversazioni dipende dall'altra. Pertanto, dobbiamo specificare la classe `DataDispatch`, poiché è coinvolta in entrambe le conversazioni contemporaneamente.

42.4 Ruoli

La nostra conclusione dall'ultima sezione è che l'ambiente in cui hanno luogo i comportamenti di `DataDispatch` è composto da tre ruoli (le frecce mostrano la direzione delle dipendenze o "chi invia messaggi a chi"):

```
User -> DataDispatch -> Channel
```

Dove `DataDispatch` è la classe specificata e il resto è il suo contesto (`Channel` è la parte del contesto da cui dipende `DataDispatch`. Per quanto io adori l'indipendenza dal contesto, la maggior parte delle classi ha bisogno di dipendere da qualche tipo di contesto, anche se in misura minima).

Usiamo questo ambiente per definire i comportamenti di `DataDispatch` che dobbiamo specificare.

42.5 Comportamenti

I comportamenti di `DataDispatch` definiti in termini di questo contesto sono:

1. Invio dati validi:

```
GIVEN User wants to dispatch a piece of data
AND a DataDispatch instance is connected to a Channel
  that accepts such data
WHEN the User dispatches the data via the DataDispatch instance
THEN the DataDispatch object should
  open the channel,
  then send the User data through the channel,
  then close the channel
```

1. Invio dati non validi:

```
GIVEN User wants to dispatch a piece of data
AND a DataDispatch instance is connected to a Channel
  that rejects such data
WHEN the User dispatches the data via the DataDispatch instance
THEN the DataDispatch object should report to the User
  that data is invalid
AND close the connection anyway
```

Per il resto di questo capitolo, mi concentrerò sul primo comportamento poiché il nostro obiettivo per ora non è creare una Specifica completa della classe DataDispatch, ma piuttosto osservare alcuni meccanismi degli oggetti mock come strumento di test.

42.6 Riempimento dei ruoli

Come accennato in precedenza, lambiente in cui avviene il comportamento si presenta così:

```
User -> DataDispatch -> Channel
```

Ora bisogna dire chi ricoprirà questi ruoli. Quelli che non abbiamo ancora riempito ho contrassegnato con punti interrogativi (?):

```
User? -> DataDispatch? -> Channel?
```

Cominciamo con il ruolo di DataDispatch. Probabilmente non sorprende che sarà riempito dalla classe concreta `DataDispatch` -- dopo tutto, questa è la classe che specifichiamo.

Il nostro ambiente ora appare così:

```
User? -> DataDispatch (concrete class) -> Channel?
```

Successivamente, chi sarà l'utente della classe `DataDispatch`? Per questa domanda, ho una risposta semplice -- il corpo dello Statement sarà l'utenza, interagirà con `DataDispatch` per attivare i comportamenti specificati. Ciò significa che il nostro ambiente ora appare così:

```
Statement body -> DataDispatch (concrete class) -> Channel?
```

Ora, l'ultimo elemento è decidere chi svolgerà il ruolo di un canale. Possiamo esprimere questo problema col seguente Statement incompiuto (ho contrassegnato tutte le incognite attuali con un doppio punto interrogativo: ??):

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
{
  //GIVEN
  Channel channel = ??; //what is it going to be?
  var dispatch = new DataDispatch(channel);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  ?? //how to specify DataDispatch behavior?
}
```

Come si vede, dobbiamo passare un'implementazione di `Channel` a un `DataDispatch`, ma non sappiamo quale dovrebbe essere questo canale. Allo stesso modo, non abbiamo una buona idea di come specificare le chiamate previste e il loro ordine.

Dal punto di vista di `DataDispatch`, è progettato per funzionare con tutto ciò che implementa l'interfaccia `Channel` e segue il protocollo, quindi non esiste un'unica implementazione "privilegiata" che sia più appropriata di altre. Ciò significa che possiamo praticamente scegliere quello che ci piace di più. Quale ci piace di più? Quello che rende più semplice la scrittura delle specifiche, ovviamente. Idealmente, vorremmo passare un canale che soddisfi al meglio i seguenti requisiti:

1. Aggiunge il minor numero possibile di effetti collaterali. Se l'implementazione di un canale utilizzata in uno Statement aggiungesse effetti collaterali, non saremmo mai sicuri se il comportamento che osserviamo durante l'esecuzione della nostra Specifica sia il comportamento di `DataDispatch` o forse il comportamento della particolare implementazione di `Channel` utilizzata in questo Statement. Questo è un requisito di fiducia -- vogliamo avere fiducia che le nostre Specifiche specifichino cosa dicono di fare.
2. È facile da controllare -- in modo da poter facilmente attivare condizioni diverse nell'oggetto che stiamo specificando. Inoltre, vogliamo essere in grado di verificare facilmente come l'oggetto specificato interagisce con esso. Questo è un requisito di convenienza.

3. È veloce da creare e facile da mantenere -- perché vogliamo concentrarci sui comportamenti che specifichiamo, non sul mantenimento o sulla creazione di classi helper. Questo è un requisito di basso attrito.

Esiste uno strumento che soddisfa questi tre requisiti meglio di altri che conosco e si chiama oggetto mock. Ecco come soddisfa i requisiti menzionati:

1. I mock non aggiungono quasi alcun effetto collaterale. Sebbene abbiano alcuni comportamenti predefiniti codificati (ad esempio quando un metodo che restituisce `int` viene chiamato su un mock, restituisce `0` per default), ma questi comportamenti sono predefiniti e privi di significato quanto possono essere. Ciò ci consente di riporre maggiore fiducia nelle nostre Specifiche.
2. I mock sono facili da controllare - ogni libreria di mock viene fornita con un'API per definire i risultati delle chiamate ai metodi predefiniti e per la verifica delle chiamate ricevute. Avere tale API offre comodità, almeno dal mio punto di vista.
3. I mock possono essere banali da mantenere. Sebbene sia possibile scrivere i propri mock (ovvero la propria implementazione di un'interfaccia che consenta l'impostazione e la verifica delle chiamate), la maggior parte di noi utilizza librerie che li generano, in genere utilizzando una funzionalità di *reflection* di un linguaggio di programmazione (nel nostro caso, C#). In genere, le librerie mock ci liberano dalla necessità di mantenere implementazioni mock, riducendo l'attrito della scrittura e del mantenimento dei nostri Statement.

Quindi usiamo un mock al posto di `Channel`! Questo fa sì che il nostro ambiente del comportamento specificato assomigli a questo:

```
Statement body -> DataDispatch (concrete class) -> Mock Channel
```

Notare che l'unica parte di questo ambiente che proviene dal codice di produzione è `DataDispatch`, mentre il suo contesto è Statement-specific.

42.7 Uso di un canale mock

Spero che ricordiate la libreria NSubstitute per la creazione di oggetti mock introdotto all'inizio del libro. Ora possiamo usarlo per creare rapidamente un'implementazione di `Channel` che si comporti come vogliamo, consenta una facile verifica del protocollo e tra `Dispatch` e `Channel` e introduca il numero minimo di effetti collaterali.

Usando questo mock per colmare le lacune nel nostro Statement, questo è ciò che otterremo:

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
{
  //GIVEN
  var channel = Substitute.For<Channel>();
  var dispatch = new DataDispatch(channel);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  Received.InOrder(() =>
  {
    channel.Open();
    channel.Send(data);
    channel.Close();
  };
}
```

in precedenza, questo Statement era incompleto, perché mancava la risposta alle seguenti due domande:

1. Da dove prendere il canale?
2. Come verificare il comportamento di `DataDispatch`?

Ho risposto alla domanda "da dove prendere il canale?" creandolo come un mock:

```
var channel = Substitute.For<Channel>();
```

Poi la seconda domanda: "come verificare il comportamento di DataDispatch?" è stata data risposta utilizzando l'API NSubstitute per verificare che il mock abbia ricevuto tre chiamate (o tre messaggi) in un ordine specifico:

```
Received.InOrder(() =>
{
  channel.Open();
  channel.Send(data);
  channel.Close();
};
```

La conseguenza è che se riorganizzo l'ordine dei messaggi inviati a `Channel` nell'implementazione del metodo `ApplyTo()` da questo:

```
public void Dispatch(byte[] data)
{
  _channel.Open();
  _channel.Send(data);
  _channel.Close();
}
```

a questo (notare l'ordine della chiamata modificato):

```
public void Dispatch(byte[] data)
{
  _channel.Send(data); //before Open()!
  _channel.Open();
  _channel.Close();
}
```

Lo Statement diventerà falso (cioè fallirà).

42.8 I mock come ancora un altro contesto

Ciò che abbiamo fatto nell'esempio precedente è stato inserire il nostro `DataDispatch` in un contesto che fosse il più affidabile, conveniente e senza attriti da utilizzare nel nostro Statement.

Alcuni dicono che specificare le interazioni degli oggetti nel contesto dei mock significa "specificare in isolamento" e che fornire tali dipendenze mock significa "isolare" la classe dalle sue dipendenze "reali". Non mi identifico molto con questo punto di vista. Dal punto di vista di una classe specifica, i mock sono ancora un altro contesto -- non sono né migliori né peggiori, non sono né più né meno reali di altri contesti in cui vogliamo inserire il nostro `Dispatch`. Certo, questo non è il contesto in cui viene eseguito in produzione, ma potremmo avere altre situazioni oltre al mero lavoro di produzione -- ad es. potremmo avere un contesto speciale per le demo, in cui contiamo i pacchetti inviati e mostriamo il throughput su una schermata della GUI. Potremmo anche avere un contesto di debug che in ogni metodo, prima di passare il controllo al codice di produzione, scrive un messaggio di trace in un log. La classe `DataDispatch` può essere utilizzata nel codice di produzione in più contesti contemporaneamente. Possiamo inviare dati attraverso la rete, a un database e a un file contemporaneamente nella nostra applicazione e la classe `DataDispatch` è utilizzabile in tutti questi scenari, ogni volta connessa a una diversa implementazione di `Channel` e utilizzata da un pezzo di codice diverso.

42.9 Riepilogo

Lo scopo di questo capitolo era solo quello di mostrare come gli oggetti mock si inseriscono nel test del codice scritto in uno stile "tell don't ask", concentrandosi su ruoli, responsabilità, comportamenti, interfacce e protocolli degli oggetti. Questo esempio è stato pensato come qualcosa di facilmente comprensibile, non come una vetrina per TDD che utilizza

mock. Per un altro capitolo lavoreremo su questo esempio minimale e poi proverò a mostrare come applico gli oggetti mock in casi più interessanti.

CAPITOLO 43

Test-first [*Testare prima*] utilizzando oggetti mock

Ora che abbiamo visto i mock in azione e li abbiamo inseriti nel contesto di uno specifico approccio di progettazione, vorrei mostrare come vengono utilizzati gli oggetti mock quando si impiega l'approccio "test-first". Per fare ciò, ripeterò l'esempio dell'ultimo capitolo. Ho già menzionato come questo esempio non sia particolarmente forte in termini di dimostrazione del potere degli oggetti mock, quindi non mi ripeterò qui. Nel prossimo capitolo farò un esempio che ritengo più adatto.

43.1 Come iniziare? -- con gli oggetto mock

Probabilmente si ricorda il capitolo "Come iniziare?" dalla prima parte di questo libro. In quel capitolo, ho descritto i seguenti modi per iniziare a scrivere uno Statement prima che l'effettiva implementazione sia in atto:

1. Iniziare con un buon nome.
2. Iniziare riempiendo la struttura GIVEN-WHEN-THEN con l'ovvio.
3. Iniziare dalla fine.
4. Iniziare invocando un metodo se c'è.

Praticamente tutte queste strategie funzionano altrettanto bene con gli Statement che utilizzano oggetti mock, quindi non li descriverò nuovamente in dettaglio. In questo capitolo mi concentrerò su una delle strategie: "Iniziare invocando un metodo se c'è" poiché è quella che utilizzo più spesso. In this chapter, I will focus on one of the strategies: "Start by invoking a method if you have one" as it's the one I use most often. Ciò è dovuto non solo alla mia scelta di utilizzare oggetti mock, ma anche allo stile di sviluppo che tendo ad utilizzare. Questo stile si chiama outside-in e tutto quello che dobbiamo sapere al riguardo, per ora, è che seguirlo significa iniziare lo sviluppo dagli input del sistema e terminare sugli output. Alcuni potrebbero considerarlo controintuitivo in quanto significa che scriveremo classi collaborando con classi che ancora non esistono. Ne darò un piccolo assaggio (insieme a una tecnica chiamata "interface discovery") in questo capitolo e approfondirò queste idee nel prossimo.

43.2 Responsabilità e Responsabilità

In questo capitolo utilizzerò due concetti che, sfortunatamente, condividono lo stesso nome: "responsabilità". Un significato di responsabilità era coniato da Rebecca Wirfs-Brock nel senso di "un obbligo di svolgere un compito o di conoscere determinate informazioni", e l'altro da Robert C. Martin per indicare "un motivo per cambiare". Per evitare questa ambiguità, in questo capitolo cercherò di chiamare il primo "obbligo" e il secondo "scopo".

La relazione tra i due può essere descritta dalle seguenti frasi:

1. Una classe ha degli obblighi nei confronti dei propri clienti.

2. Gli obblighi sono ciò che la classe promette di fare per i propri clienti.
3. La classe non deve adempiere agli obblighi da sola. In genere, lo fa con l'aiuto di altri oggetti -- i suoi collaboratori. Tali collaboratori, a loro volta, hanno i loro obblighi e collaboratori.
4. Ciascuno dei collaboratori ha il suo scopo - un ruolo nell'adempimento dell'obbligo principale. La finalità risulta dalla scomposizione dell'obbligazione principale.

43.3 Channel e DataDispatch ancora una volta

Ricordate l'esempio dell'ultimo capitolo? Immaginiamo di trovarci in una situazione in cui abbiamo già la classe `DataDispatch`, ma la sua implementazione è vuota -- dopo tutto, questo è ciò che testeremo.

Quindi per ora la classe `DataDispatch` appare così

```
public class DataDispatch
{
 public void ApplyTo(byte[] data)
 {
  throw new NotImplementedException();
 }
}
```

Da dove ho preso questa lezione in questa forma? Bene, supponiamo per ora che io sia nel mezzo dello sviluppo e che questa lezione sia il risultato delle mie precedenti attività TDD (dopo aver letto questo e il prossimo capitolo, spero che avrete un'idea più chiara di come funziona).

43.4 Il primo comportamento

Un ciclo TDD inizia con uno Statement falso. Quale comportamento dovrebbe descrivere? Non ne sono ancora sicuro, ma, dato che conosco già la classe che avrà i comportamenti che voglio specificare, inoltre ha un solo metodo (`ApplyTo()`), posso scrivere quasi alla cieca una dichiarazione in cui crea un oggetto di questa classe e invoca il metodo:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var dispatch = new DataDispatch();

 //WHEN
 dispatch.ApplyTo(); //TODO doesn't compile

 //THEN
 Assert.True(false); //TODO state expectations
}
```

Notare diverse cose:

1. Attualmente sto usando un nome fittizio per l'istruzione e ho aggiunto un elemento TODO alla mia lista per correggerlo in seguito, quando definirò lo scopo e il comportamento di `DataDispatch`.
2. Secondo la sua firma, il metodo `ApplyTo()` accetta un argomento, ma non ne ho fornito alcuno nello Statement. Per ora, non voglio pensarci troppo, voglio solo scartare tutto quello che so.
3. La sezione `//THEN` è vuota per ora: contiene una sola asserzione progettata per fallire quando il flusso di esecuzione la raggiunge (in questo modo mi proteggo dal rendere erroneamente vera l'istruzione finché non dichiaro le mie reali aspettative). Definirò la sezione `//THEN` una volta capito qual è lo scopo che voglio dare a questa classe e il comportamento che voglio specificare.
4. Se si ricorda l'interfaccia `Channel` dell'ultimo capitolo, si immagini che non esista ancora e che non sappia nemmeno se ne avrò bisogno. Lo "scoprirò" più tardi.

43.4.1 Appoggiarsi al compilatore

Quindi ho fatto il "brain dump". Cosa faccio ora? Non voglio ancora pensarci troppo (verrà il momento per quello). Innanzitutto, chiedo il feedback al mio compilatore -- forse può darmi qualche suggerimento su cosa mi manca?

Attualmente, il compilatore si lamenta del fatto che invoco il metodo `ApplyTo()` senza passare alcun argomento. Qual è il nome dell'argomento? Mentre cerco la firma del metodo `ApplyTo()`, sembra che il nome sia `data`:

```
public void ApplyTo(byte[] data)
```

Hmm, se sono i dati che vuole, allora passiamo alcuni dati. Non voglio ancora decidere di cosa si tratta, quindi mi comporterò *come se avessi* una variabile chiamata `data` e scriverò semplicemente il suo nome dove è previsto l'argomento:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var dispatch = new DataDispatch();

 //WHEN
 dispatch.ApplyTo(data); //TODO still doesn't compile

 //THEN
 Assert.True(false); //TODO state expectations
}
```

Il compilatore mi fornisce ulteriori feedback -- dice che la mia variabile `data` non è definita. Potrebbe sembrare strano (come se non lo sapessi!), ma in questo modo faccio un ulteriore passo avanti. Ora so che devo definire questi `data`. Posso utilizzare una funzionalità di "soluzione rapida" del mio IDE per introdurre una variabile. Per esempio. negli IDE Jetbrains (IntelliJ IDEA, Resharper, Rider...) questo può essere fatto premendo `ALT` + `ENTER` quando il cursore si trova sul nome della variabile mancante. L'IDE creerà la seguente dichiarazione:

```
byte[] data;
```

Notare che l'IDE ha indovinato il tipo di variabile. Come faceva a saperlo? Perché la definizione del metodo a cui provo a passarlo ha già il tipo dichiarato:

```
public void ApplyTo(byte[] data)
```

Naturalmente, la dichiarazione di `data` che il mio IDE ha inserito nel codice non verrà comunque compilata perché C# richiede che le variabili siano inizializzate esplicitamente. Quindi il codice dovrebbe assomigliare a questo:

```
byte[] data = ... /* whatever initialization code*/;
```

43.4.2 Accendere il cervello -- che dire dei dati?

Sembra che non possa più continuare la mia parata da cervello morto. Per decidere come definire questi dati, devo attivare i miei processi mentali e decidere qual è esattamente l'obbligo del metodo `ApplyTo()` e a cosa servono i `data`. Dopo averci pensato un po', decido che l'applicazione dell'invio dei dati dovrebbe inviare i dati che riceve. Ma... dovrebbe fare tutto il lavoro, o magari delegare alcune parti? Esistono almeno due sotto-task associati all'invio dei dati:

1. La logica dell'invio non elaborato (disposizione dei dati, push ad esempio attraverso un socket web, ecc.)
2. Gestire la durata della connessione per tutto il ciclo di vita (decidere quando aprirla e quando chiuderla, smaltire tutte le risorse allocate, anche a fronte di un'eccezione che può verificarsi durante l'invio).

Decido di non inserire l'intera logica nella classe `DataDispatch` perché:

1. Avrebbe più di uno scopo (come descritto in precedenza) -- in altre parole, violerebbe il principio di responsabilità unica.

2. Sono mentalmente incapace di capire come scrivere una dichiarazione falsa per così tanta logica prima dell'implementazione. Lo considero sempre un segno che sto cercando di caricare troppo peso su una singola classe[1].

43.4.3 Introduzione di un collaboratore

Decido quindi di dividere e conquistare, cioè trovare a `DataDispatch` dei collaboratori che lo aiutino a raggiungere il suo obiettivo e delegare loro parti della logica. Dopo alcune considerazioni, concludo che lo scopo di `DataDispatch` dovrebbe essere la gestione della durata della connessione. Per il resto della logica decido di delegare ad un collaboratore il ruolo che chiamo `Channel`. Il processo di definizione dei ruoli del collaboratore e di delega a loro parte degli obblighi specificati della classe è chiamato *interface discovery*.

Ad ogni modo, poiché il mio `DataDispatch` delegherà una parte della logica al `Channel`, deve saperlo. Quindi collegherò questo nuovo collaboratore a `DataDispatch`. Un `DataDispatch` non funzionerà senza un `Channel`, il che significa che devo passare il canale a `DataDispatch` come parametro del costruttore. Sono tentato di dimenticare il TDD, basta andare all'implementazione di questo costruttore e aggiungere lì un parametro, ma resisto. Come al solito, inizio le mie modifiche dallo Statement. Pertanto, modifico il seguente codice:

```
//GIVEN
var dispatch = new DataDispatch();
```

in:

```
//GIVEN
var dispatch = new DataDispatch(channel); //doesn't compile
```

Utilizzo una variabile `channel` *come se* fosse già definita nel corpo dell'istruzione e *come se* il costruttore la richiedesse già. Naturalmente, nessuna di queste è ancora vera. Questo porta il mio compilatore a darmi più errori di compilazione. Per me, questa è una preziosa fonte di feedback di cui ho bisogno per progredire ulteriormente. La prima cosa che il compilatore mi dice di fare è introdurre una variabile `channel`. Ancora una volta, utilizzo il mio IDE per generarla. Questa volta, però, il risultato della generazione è:

```
object channel;
```

L'IDE non è riuscito a indovinare il tipo corretto di `channel` (che sarebbe `Channel`) e lo ha reso un `object`, perché non ho ancora creato il tipo `Channel`.

Innanzitutto, introdurrò l'interfaccia `Channel` modificando la dichiarazione `object channel;` in `Channel channel;`. Questo mi darà un altro errore di compilazione, poiché il tipo `Channel` non esiste. Per fortuna, crearlo è solo a un clic dall'IDE (ad esempio in Resharper, posiziono il cursore sul tipo inesistente, premo `ALT + ENTER` e scelgo un'opzione per crearlo come interfaccia). Fare questo mi darà:

```
public interface Channel
{

}
```

che è sufficiente per superare questo particolare errore del compilatore, ma poi ne ottengo un altro -- alla variabile `channel` non è assegnato nulla. Ancora una volta, devo riaccendere il cervello. Fortunatamente, questa volta posso appoggiarmi a una regola semplice: nel mio progetto, `Channel` è un ruolo e, come accennato nell'ultimo capitolo, utilizzo dei mock per interpretare i ruoli dei miei collaboratori. Quindi la conclusione è usare un mock. Applicando questa regola, cambio la seguente riga:

```
Channel channel;
```

in:

```
var channel = Substitute.For<Channel>();
```

[1] maggiori informazioni su questo argomento nei capitoli successivi.

L'ultimo errore del compilatore da risolvere per introdurre completamente il collaboratore `Channel` è fare in modo che il costruttore `DataDispatch` accetti il canale come argomento. Per ora `DataDispatch` utilizza un costruttore implicito senza parametri. Ne genero uno nuovo, ancora una volta, utilizzando il mio IDE. Vado nel punto in cui viene chiamato il costruttore con il canale come argomento e dico al mio IDE di correggere la firma del costruttore in base a questo utilizzo. In questo modo ottengo un codice costruttore all'interno della classe `DataDispatch`:

```
public DataDispatch(Channel channel)
{

}
```

Notare che il costruttore non fa ancora nulla con il canale. Potrei creare un nuovo campo e assegnargli il canale, ma non ho ancora bisogno di farlo, quindi decido che posso aspettare ancora un po'.

Analizzando il mio Statement, attualmente ho:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 byte[] data; // does not compile yet
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Assert.True(false); //TODO state expectations
}
```

In questo modo, ho definito un collaboratore `Channel` e l'ho introdotto prima nel mio Statement e poi nel codice di produzione.

43.4.4 Specificare le expectation [*aspettative*]

Il compilatore e la mia lista TODO sottolineano che ho ancora tre compiti da svolgere per l'attuale Statement:

- definire la variabile `data`,
- dare un nome allo Statement e
- dichiarare le mie aspettative (la sezione `THEN` dello Statement)

Posso eseguirli nell'ordine che ritengo opportuno, quindi scelgo l'ultima attività dall'elenco - indicando il comportamento previsto.

Per specificare cosa ci si aspetta da `DataDispatch`, devo rispondere a quattro domande:

1. Quali sono gli obblighi di `DataDispatch`?
2. Qual è lo scopo di `DataDispatch`?
3. Chi sono i collaboratori che hanno bisogno di ricevere messaggi da `DataDispatch`?
4. Qual è il comportamento di `DataDispatch` che devo specificare?

Le mie risposte a queste domande sono:

1. `DataDispatch` è obbligato a inviare i dati finché sono validi. In caso di dati non validi, genera un'eccezione. Sono due comportamenti. Poiché specifico un solo comportamento per Statement, devo sceglierne uno. Scelgo il primo (che d'ora in poi chiamerò "il sentiero felice"), aggiungendo il secondo alla mia lista delle cose da fare:

```
//TODO: specify a behavior where sending data
//      through a channel raises an exception
```

2. Lo scopo di `DataDispatch` è gestire la durata della connessione durante l'invio dei dati ricevuti tramite il metodo `ApplyTo()`. Mettendolo insieme alla risposta all'ultima domanda, quello che dovrei specificare è come `DataDispatch` gestisce questa vita durante lo scenario del "percorso felice". Il resto della logica di cui ho bisogno per adempiere all'obbligo di `DataDispatch` non rientra nell'ambito del presente Statement poiché ho deciso di spingerla ai collaboratori.
3. Ho già definito un collaboratore e l'ho chiamato `Channel`. Come accennato nell'ultimo capitolo, negli Statement a livello di unità, riempio i ruoli dei miei collaboratori con mock e specifico quali messaggi dovrebbero ricevere. Pertanto, so che la sezione `THEN` descriverà i messaggi che il ruolo `Channel` (interpretato da un oggetto mock) dovrebbe ricevere dal mio `DataDispatch`.
4. Ora che conosco lo scenario, lo scopo e i collaboratori, posso definire il mio comportamento atteso in termini di queste cose. La mia conclusione è che mi aspetto che `DataDispatch` gestisca correttamente (scopo) un `Canale` (collaboratore) in uno scenario di "percorso felice" in cui i dati vengono inviati senza errori (obbligo). Poiché i canali vengono generalmente aperti prima di essere utilizzati e vengono chiusi successivamente, ciò che ci si aspetta che faccia il mio `DataDispatch` è aprire il canale, inserire i dati attraverso di esso e poi chiuderlo.

Come implementare tali aspettative? Dal punto di vista dell'implementazione, quello che mi aspetto è che `DataDispatch`:

- effettui chiamate corrette sul collaboratore `Channel` (open, send, close)
- con argomenti corretti (i dati ricevuti)
- nell'ordine corretto (non è possibile, ad esempio, chiamare chiudendo prima di aprire)
- numero corretto di volte (ad esempio, non inviare i dati due volte)

Posso specificarlo utilizzando la sintassi `Received.InOrder()` di NSubstitute. Lo userò quindi per affermare che i tre metodi dovrebbero essere chiamati in un ordine specifico. Aspetta, quali metodi? Dopotutto, la nostra interfaccia `Channel` si presenta così:

```
public interface Channel
{

}
```

quindi non ci sono metodi di sorta qui. La risposta è -- proprio come ho scoperto la necessità dell'interfaccia `Channel` e poi l'ho portata in vita, ora ho scoperto che ho bisogno di tre metodi: `Open()`, `Send()` e `Close()`. Allo stesso modo di come ho fatto con l'interfaccia `Channel`, li userò nel mio Statement *come se* esistessero:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 byte[] data; // does not compile yet
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open(); //doesn't compile
  channel.Send(data); //doesn't compile
  channel.Close(); //doesn't compile
```

(continues on next page)

(continua dalla pagina precedente)

```
 });
}
```

per poi crearli utilizzando il mio IDE e la sua scorciatoia per generare classi e metodi mancanti. In questo modo ottengo:

```
public interface Channel
{
 void Open();
 void Send(byte[] data);
 void Close();
}
```

Ora mi restano solo due cose sulla mia lista -- dare un buon nome allo Statement e decidere cosa dovrebbe contenere la variabile `data`. Scelgo quest'ultimo poiché è l'ultima cosa che impedisce al compilatore di compilare ed eseguire il mio Statement e mi aspetto che mi fornisca un feedback più utile.

43.4.5 La variabile `data`

Cosa devo assegnare alla variabile `data`? È ora di pensare a quanto `DataDispatch` deve sapere sui dati che invia attraverso il canale. Decido che `DataDispatch` dovrebbe funzionare con qualsiasi dato -- dopotutto il suo scopo è gestire la connessione -- non è necessario leggere o manipolare i dati per farlo. Qualcuno, da qualche parte, probabilmente ha bisogno di convalidare questi dati, ma decido che se aggiungessi la logica di convalida a `DataDispatch`, ne verrebbe meno il mono-scopo. Quindi spingo ulteriormente la validazione all'interfaccia `Channel`, poiché la decisione di accettare o meno i dati dipende dall'effettiva implementazione della logica di invio. Pertanto, definisco la variabile `data` nel mio Statement semplicemente come `Any.Array<byte>()`:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var data = Any.Array<byte>();
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open();
  channel.Send(data);
  channel.Close();
 });
}
```

43.4.6 Bel nome

Lo Statement ora viene compilato ed eseguito (al momento è falso, ovviamente, ma ci arriverò), quindi ciò di cui ho bisogno è dare a questo Statement un nome migliore. Io andrò con `ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch` [*DovrebbeInviareIDatiTramiteIlCanaleApertoPoiChiuderloQuandoVieneRichiestoDiInviare*]. Questo è stato l'ultimo TODO sul lato Specifiche, quindi vediamo il codice completo dello Statement:

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
```

(continues on next page)

(continua dalla pagina precedente)

```
{
 //GIVEN
 var data = Any.Array<byte>();
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open();
  channel.Send(data);
  channel.Close();
 });
}
```

43.4.7 Fallire per la ragione giusta

Lo Statement che ho appena scritto può ora essere valutato e, come previsto, è falso. Questo perché l'attuale implementazione del metodo `ApplyTo` lancia una `NotImplementedException`:

```
public class DataDispatch
{
 public DataDispatch(Channel channel)
 {

 }

 public void ApplyTo(byte[] data)
 {
  throw new NotImplementedException();
 }
}
```

Quello che mi piacerebbe vedere prima di iniziare a implementare il comportamento corretto è che lo Statement è falso perché le asserzioni (in questo caso, verifiche fittizie) falliscono. Quindi la parte dello Statement che mi piacerebbe vedere lanciare un'eccezione è questa:

```
Received.InOrder(() =>
{
 channel.Open();
 channel.Send(data);
 channel.Close();
});
```

ma invece, ottengo un'eccezione già a partire da:

```
//WHEN
dispatch.ApplyTo(data);
```

Per progredire oltre la sezione `WHEN`, devo spingere il codice di produzione un po' più in là verso il comportamento corretto, ma solo quanto basta per vedere il fallimento previsto. Per fortuna, posso ottenerlo facilmente accedendo al metodo `ApplyTo()` e rimuovendo la clausola `throw`:

```
public void ApplyTo(byte[] data)
{

}
```

Questo da solo è sufficiente per vedere la verifica mock che rende falso il mio Statement. Ora che posso vedere che lo Statement è falso per il motivo corretto, il passo successivo è inserire l'implementazione corretta per rendere vera lo Statement.

43.4.8 Rendere vero lo Statement

Inizio con il costruttore `DataDispatch`, che attualmente accetta un `Channel` come parametro, ma non ci fa nulla:

```
public DataDispatch(Channel channel)
{

}
```

Voglio assegnare il canale a un campo appena creato (posso farlo utilizzando un singolo comando nella maggior parte degli IDE). Il codice diventa quindi:

```
private readonly Channel _channel;

public DataDispatch(Channel channel)
{
 _channel = channel;
}
```

Ciò mi consente di utilizzare `_channel` nel metodo `ApplyTo()` che sto cercando di implementare. Ricordando che il mio obiettivo è aprire il canale, inviare i dati e chiudere il canale, immetto:

```
public void ApplyTo(byte[] data)
{
 _channel.Open();
 _channel.Send(data);
 _channel.Close();
}
```

{{keyLock}} A dire il vero, a volte prima di scrivere l'implementazione corretta, gioco un po', sbagliando lo Statement in diversi modi, solo per vedere se riesco a indovinare correttamente il motivo per cui lo Statement diventerà falso e per assicurarmi che i messaggi di errore siano sufficientemente informativi. Ad esempio, all'inizio potrei solo implementare l'apertura del canale e osservare se l'affermazione è ancora falsa e se il motivo cambia come mi aspetto. Poi potrei aggiungere l'invio dei dati, ma passare qualcosa di diverso da `_data` al metodo `Send()` (ad esempio un `null`), ecc. In questo modo, "testo il mio test", non solo per verificarne la correttezza (se fallirà per il motivo giusto) ma anche per la diagnostica (mi fornirà informazioni sufficienti quando fallisce?). Infine, questo è anche un modo per apprendere come i miei strumenti di automazione dei test mi informano sui problemi in questi casi.

43.5 Secondo comportamento -- specificare un errore

Il primo Statement è implementato, quindi è tempo per il secondo -- ricordare che l'ho inserito nell'elenco delle cose da fare qualche tempo fa in modo da non dimenticarmene:

```
//TODO: specify a behavior where sending data
//      through a channel raises an exception
```

Questo comportamento fa sì che quando l'invio fallisce con un'eccezione, l'utente di `DataDispatch` dovrebbe ricevere un'eccezione e la connessione dovrebbe essere chiusa in modo sicuro. Notare che il significato di "chiusura della connessione" è delegato alle implementazioni di `Channel`, quindi quando si specificano i comportamenti di `DataDispatch` devo

solo preoccuparmi se il metodo `Close()` di `Channel` viene invocato correttamente. Lo stesso vale per il significato di "errori durante l'invio dei dati" -- questo è anche l'obbligo di `Channel`. Ciò che dobbiamo specificare su `DataDispatch` è come gestisce gli errori di invio riguardanti il suo utente e il suo `Channel`.

43.5.1 A cominciare da un bel nome

Questa volta scelgo la strategia di iniziare con un buon nome, perché sento di avere una comprensione molto migliore di quale comportamento devo specificare rispetto al mio Statement precedente. Scelgo il seguente nome per indicare il comportamento previsto:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //...
}
```

Prima di iniziare a scomporre il nome in codice utile, inizio affermando ciò che è ovvio (notare che sto mescolando due strategie per iniziare dallo Statement falso adesso -- non ho detto che non lo si può fare ora, vero?). Avendo imparato molto scrivendo e implementando lo Statement precedente, so per certo che:

1. Devo lavorare di nuovo con `DataDispatch`.
2. Devo passare un mock di `Channel` al costruttore `DataDispatch`.
3. Il ruolo `Channel` sarà svolto da un oggetto mock.
4. Devo invocare il metodo `ApplyTo()`.
5. Ho bisogno di qualche tipo di dati non validi (anche se non so ancora cosa fare per renderli "non validi").

Lo scrivo come codice:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //GIVEN
 var channel = Substitute.For<Channel>();
 var dataDispatch = new DataDispatch(channel);
 byte[] invalidData; //doesn't compile

 //WHEN
 dataDispatch.ApplyTo(invalidData);

 //THEN
 Assert.True(false); //no expectations yet
}
```

43.5.2 Mi aspetto che il canale sia chiuso

So anche che un aspetto del comportamento previsto è la chiusura del canale. So come scrivere questa expectation [*aspettativa*] -- posso usare il metodo `Received()` di NSubstitute sul mock del canale. Questo, ovviamente, andrà nella sezione `//THEN`:

```
 //THEN
 channel.Received(1).Close(); //added
 Assert.True(false); //not removing this yet
}
```

Ho usato `Received(1)` invece semplicemente di `Received()`, perché tentare di chiudere il canale più volte potrebbe causare problemi, quindi voglio essere esplicito sull'aspettativa che `DataDispatch` chiuda il canale esattamente una volta. Un'altra cosa -- non sto ancora rimuovendo `Assert.True(false)`, poiché l'attuale implementazione chiude già il

canale e quindi la dichiarazione potrebbe diventare vera se non fosse per questa affermazione (se compilata, ovviamente). Rimuoverò questa affermazione solo dopo aver definito completamente il comportamento.

43.5.3 In attesa dell'eccezione

Un'altra cosa che mi aspetto che `DataDispatch` faccia in questo comportamento è rilanciare eventuali errori di invio, che vengono segnalati come eccezioni lanciate da `Channel` dal metodo `Send()`.

{{keyLock}} In genere, scrivo raramente dichiarazioni sulle eccezioni rilanciate, ma qui non ho scelta -- se non rilevo l'eccezione nel mio Statement, non sarò in grado di valutare se il canale è stato chiuso o meno, poiché l'eccezione non rilevata interromperà l'esecuzione dello Statement.

Per specificare che mi aspetto un'eccezione nel mio Statement, devo utilizzare un'asserzione speciale chiamata `Assert.Throws<>()` e passare il codice che dovrebbe generare l'eccezione come lambda:

```
 //WHEN
Assert.Throws<Exception>(() =>
  dataDispatch.ApplyTo(invalidData));
```

43.5.4 Definizione di dati non validi

Il mio compilatore mi mostra che la variabile `data` non è definita. OK, ora è giunto il momento di definire i *dati non validi*.

Prima di tutto, ricordare che `DataDispatch` non può distinguere tra dati validi e non validi - questo è lo scopo del `Channel` poiché ciascuna implementazione di `Channel` potrebbe avere criteri diversi per la validazione dei dati. Nel mio Statement, utilizzo un mock per svolgere il ruolo di canale, quindi posso semplicemente dire al mio mock che dovrebbe considerare non validi i dati che definisco nel mio Statement. Pertanto, il valore stesso dei `data` è irrilevante finché configuro il mio `Channel` mock in modo che si comporti come se non fosse valido. Ciò significa che posso semplicemente definire i `data` come qualsiasi array di byte:

```
var invalidData = Any.Array<byte>();
```

Devo anche scrivere l'ipotesi di come si comporterà `channel` dati questi dati:

```
//GIVEN
...
var exceptionFromChannel = Any.Exception();
channel.When(c => c.Send(invalidData)).Throw(exceptionFromChannel);
```

Notare che il posto in cui configuro il mock per generare un'eccezione è la sezione `//GIVEN`. Questo perché qualsiasi comportamento mock predefinito è la mia assunzione. Pre-inscatolando il risultato del metodo, in questo caso, dico "dato che il canale per qualche motivo rifiuta questi dati".

Ora che ho il codice completo dello Statement, posso eliminare l'asserzione `Assert.True(false)`. Lo Statement completo si presenta così:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //GIVEN
 var channel = Substitute.For<Channel>();
 var dataDispatch = new DataDispatch(channel);
 var data = Any.Array<byte>();
 var exceptionFromChannel = Any.Exception();

 channel.When(c => c.Send(data)).Throw(exceptionFromChannel);

 //WHEN
 var exception = Assert.Throws<Exception>(() =>
  dataDispatch.ApplyTo(invalidData));
```

(continues on next page)

(continua dalla pagina precedente)

```
 //THEN
 Assert.Equal(exceptionFromChannel, exception);
 channel.Received(1).Close();
}
```

Ora, potrebbe sembrare un po' complicato, ma dato il mio set di strumenti, questo dovrà bastare. Lo Statement ora diventerà falso sulla seconda affermazione. Aspetta, il secondo? E il primo? Bene, la prima affermazione dice che un'eccezione dovrebbe essere lanciata nuovamente e che i metodi in C# lanciano nuovamente l'eccezione per default, senza richiedere alcuna implementazione da parte mia[2]. Dovrei semplicemente accettarlo e andare avanti? Beh, non voglio. Ricordate cosa ho scritto nella prima parte del libro -- dobbiamo vedere ogni affermazione fallire almeno una volta. Un'affermazione che passa subito è qualcosa di cui dovremmo essere sospettosi. Ciò che devo fare ora è guastare temporaneamente il comportamento in modo da poter vedere l'errore. Posso farlo in (almeno) due modi:

1. Andando allo Statement e commentando la riga che configura il mock `Channel` per lanciare un'eccezione.
2. Andando al codice di produzione e circondando l'istruzione `channel.Send(data)` con un blocco try-catch.

Andrebbe bene in entrambi i casi, ma in genere preferisco modificare il codice di produzione e non alterare i miei Statement, quindi ho scelto la seconda strada. Avvolgendo l'invocazione `Send()` con `try` e un `catch` vuoto, ora posso osservare il fallimento dell'asserzione, perché è prevista un'eccezione ma dall'invocazione di `dataDispatch.ApplyTo()` non esce nessuna eccezione. Ora sono pronto per annullare la mia ultima modifica, fiducioso che il mio Statement descriva bene questa parte del comportamento e posso concentrarmi sulla seconda asserzione, che è:

```
channel.Received(1).Close();
```

Questa affermazione fallisce perché la mia attuale implementazione del metodo `ApplyTo()` è:

```
_channel.Open();
_channel.Send(data);
_channel.Close();
```

e un'eccezione lanciata dal metodo `Send()` interrompe l'elaborazione, uscendo immediatamente dal metodo, quindi `Close()` non viene mai chiamato. Posso modificare questo comportamento utilizzando il blocco `try-finally` per racchiudere la chiamata a `Send()`[3]:

```
_channel.Open();
try
{
 _channel.Send(data);
}
finally
{
 _channel.Close();
}
```

Ciò rende vera la mia seconda affermazione e conclude questo esempio. Se dovessi andare avanti, il mio prossimo passo sarebbe implementare l'interfaccia `Channel` appena scoperta, poiché attualmente non ha alcuna implementazione.

43.6 Riepilogo

In questo capitolo ho approfondito la scrittura di Statement basati su mock e sullo sviluppo di classi in modalità "test-first". Non sto mostrando questo esempio come una prescrizione o come una sorta di "unico vero modo" per testare tale implementazione - alcune cose avrebbero potuto essere fatte diversamente. Ad esempio, ci sono state molte situazioni in cui

[2] ecco perché in genere non specifico che qualcosa debba rilanciare un'eccezione -- lo faccio questa volta perché altrimenti non mi permetterebbe di specificare come `DataDispatch` utilizza un `Channel`.

[3] ovviamente, il modo idiomatico per farlo in C# sarebbe quello di utilizzare l'interfaccia `IDisposable` e un blocco `using` (o `IAsyncDisposable` e `await using` nel caso di chiamate `async`).

ho ricevuto diversi elementi TODO segnalati dal mio compilatore o dal mio Statement falso. A seconda di molti fattori, avrei potuto affrontarli in un ordine diverso. Oppure nel secondo comportamento, avrei potuto definire `data` come `Any.Array<byte>()` fin dall'inizio (e lasciare un elemento TODO per controllarlo in seguito e confermare se può rimanere così) per portare lo Statement in uno stato di compilazione più rapidamente.

Un altro punto interessante è stato il momento in cui ho scoperto l'interfaccia `Channel` -- sono consapevole di esserci scivolato sopra dicendo qualcosa del tipo "possiamo vedere che la classe ha troppi scopi, poi accade la magia e poi abbiamo un'interfaccia a cui delegare parti della logica". Questa "magia accade" o, come ho già detto, "la scoperta dell'interfaccia", è qualcosa che approfondirò nei capitoli seguenti.

Si potrebbe aver notato che questo capitolo è più lungo del precedente, il che potrebbe portare alla conclusione che il TDD complica le cose anziché semplificarle. Cerano, tuttavia, diversi fattori che hanno reso questo capitolo più lungo:

1. In questo capitolo ho specificato due comportamenti (un "percorso felice" più la gestione degli errori), mentre nell'ultimo capitolo ne ho specificato solo uno (il "percorso felice").
2. In questo capitolo ho progettato e implementato la classe `DataDispatch` e ho scoperto l'interfaccia `Channel` mentre nell'ultimo capitolo mi sono state fornite fin dall'inizio.
3. Poiché presumo che il modo di scrivere gli Statement basato sul "test-first " potrebbe essere meno familiare, mi sono preso il tempo per spiegare il mio processo di pensiero in modo più dettagliato.

Quindi niente paura -- quando ci si abitua, il processo descritto in questo capitolo richiede in genere, nel peggiore dei casi, diversi minuti.

CAPITOLO 44

Il test-driving ai confini di input

In questo capitolo ci uniremo nuovamente a Johnny e Benjamin mentre provano a testare un sistema partendo dai suoi confini di input. Si spera che ciò dimostri come si ottengono astrazioni dai bisogni e come si inventano i ruoli ai confini di un sistema. I capitoli successivi esploreranno il modello del dominio. Questo esempio fa diverse ipotesi:

1. In questa storia, Johnny è un superprogrammatore, che non commette mai errori. Nel TDD nella vita reale, le persone commettono errori e li correggono, a volte vanno avanti e indietro pensando ai test e alla progettazione. Qui, Johnny fa tutto bene la prima volta. Anche se so che questo è un calo di realismo, spero che possa aiutare i miei lettori a osservare come funzionano alcuni meccanismi del TDD. Questo è anche il motivo per cui Johnny e Benjamin non avranno bisogno di effettuare il refactoring di nulla in questo capitolo.
2. Non ci saranno Statement scritti a un livello superiore a quello unitario. Ciò significa che Johnny e Benjamin eseguiranno il TDD utilizzando solo Statement a livello di unità. Questo è il motivo per cui dovranno fare alcune cose che potrebbero evitare se potessero scrivere uno Statement di livello superiore. Una parte separata di questo libro tratterà del lavoro con diversi livelli di Statement contemporaneamente.
3. Questo capitolo (e molti dei successivi) eviteranno l'argomento relativo al lavoro con qualsiasi I/O, casualità e altri elementi difficili da testare. Per ora, voglio concentrarmi sulla logica del test-driving basata su codice puro.

Detto questo, uniamoci a Johnny e Benjamin e vediamo che tipo di problema stanno affrontando e come cercano di risolverlo utilizzando il TDD.

44.1 Sistemazione della biglietteria

Johnny: Cosa pensi dei treni, Benjamin?

Benjamin: Me lo chiedi perché ieri stavo viaggiando in treno per arrivare qui? Beh, mi piace, soprattutto dopo alcuni cambiamenti avvenuti negli ultimi anni. Penso davvero che le ferrovie di oggi siano moderne e a misura di passeggero.

Johnny: E riguardo al processo di prenotazione del posto?

Benjamin: Oh, quello... Voglio dire, perché non hanno ancora automatizzato il processo? È davvero pensabile che nel 21ř secolo non sia possibile prenotare un posto tramite Internet?

Johnny: Speravo che lo dicessi perché il nostro prossimo compito è fare esattamente questo.

Benjamin: Intendi prenotare i posti tramite Internet?

Johnny: Sì, la compagnia ferroviaria ci ha assunti.

Benjamin: Mi stai prendendo in giro, vero?

Johnny: No, sto dicendo la verità.

Benjamin: Assurdo.

Johnny: Prendi il tuo smartphone e controlla la posta. Ti ho già inoltrato i dettagli.

Benjamin: Ehi, sembra vero. Perché non me l'hai detto prima?

Johnny: Te lo spiegherò strada facendo. Dai, andiamo.

44.2 Oggetti iniziali

Benjamin: abbiamo qualche tipo di requisiti, storie o altro su cui lavorare?

Johnny: Sì, te ne spiegherò alcuni mentre ti guido attraverso le strutture dei dati di input e output. Basterà a farci andare avanti.

44.2.1 Richiesta

Johnny: Qualcuno ha già scritto la parte che accetta una richiesta HTTP e la mappa alla seguente struttura:

```
public class ReservationRequestDto
{
  public readonly string TrainId;
  public readonly uint SeatCount;

  public ReservationRequestDto(string trainId, uint seatCount)
  {
    TrainId = trainId;
    SeatCount = seatCount;
  }
}
```

Benjamin: Capisco... Ehi, perché nel nome `ReservationRequestDto` c'è `Dto`? Che cos'è?

Johnny: Il suffisso `Dto` significa che questa classe rappresenta un Data Transfer Object (abbreviato, DTO)[1]. Il suo ruolo è semplicemente quello di trasferire i dati oltre i confini del processo.

Benjamin: Quindi intendi che è necessario solo per rappresentare una sorta di XML o JSON inviato all'applicazione?

Johnny: Sì, possiamo dire così. Il motivo per cui le persone in genere inseriscono `Dto` in questi nomi è per comunicare che queste strutture dati sono speciali - rappresentano un contratto esterno e non possono essere modificati come altri oggetti.

Benjamin: Significa che non posso toccarli?

Johnny: Significa che se li tocchi, dovresti assicurarti che siano ancora mappati correttamente dai dati esterni, come JSON o XML.

Benjamin: Quindi sarà meglio non scherzarci. Ottimo, e che mi dici dell'ID del treno?

Johnny: Rappresenta un treno. L'applicazione client che utilizza il backend che scriveremo capirà questi ID e li utilizzerà per comunicare con noi.

Benjamin: Fantastico, qual è il prossimo passo?

Johnny: L'applicazione client comunica al nostro servizio quanti posti deve prenotare, ma non dice dove. Questo è il motivo per cui esiste solo un parametro `seatCount`. Il nostro servizio deve determinare quali posti scegliere.

Benjamin: Quindi se una coppia vuole prenotare due posti, può stare in carrozze diverse?

Johnny: Sì, tuttavia, ci sono alcune regole di preferenza che dobbiamo codificare, ad esempio, se possibile, un'unica prenotazione dovrebbe avere tutti i posti nella stessa carrozza. Ti aggiornerò sulle regole più tardi.

[1] Patterns of enterprise application architecture, M. Fowler.

44.2.2 Risposta

Benjamin: Restituiamo qualcosa all'app client?

Johnny: Sì, dobbiamo restituire una risposta che, nessuna sorpresa, è anche modellata nel codice come DTO. Questa risposta rappresenta la prenotazione effettuata:

```
public class ReservationDto
{
  public readonly string TrainId;
  public readonly string ReservationId;
  public readonly List<TicketDto> PerSeatTickets;

  public ReservationDto(
    string trainId,
    List<TicketDto> perSeatTickets,
    string reservationId)
  {
    TrainId = trainId;
    PerSeatTickets = perSeatTickets;
    ReservationId = reservationId;
  }
}
```

Benjamin: Vedo che c'è un ID del treno, che è... lo stesso nella richiesta, suppongo?

Johnny: Giusto. Viene utilizzato per correlare la richiesta e la risposta.

Benjamin: ...e c'è un ID della prenotazione - il nostro servizio lo genera?

Johnny: Esatto.

Benjamin: ma il campo `PerSeatTickets`... è un elenco di `TicketDto`, che da quanto ho capito è uno dei nostri tipi custom. Dove si trova?

Johnny: Ho dimenticato di mostrartelo. `TicketDto` è definito come:

```
public class TicketDto
{
  public readonly string Coach;
  public readonly int SeatNumber;

  public TicketDto(string coach, int seatNumber)
  {
    Coach = coach;
    SeatNumber = seatNumber;
  }
}
```

quindi ha il nome della carrozza e il numero del posto, e ne abbiamo un elenco nella nostra prenotazione.

Benjamin: Quindi una singola prenotazione può contenere molti biglietti e ogni biglietto è per un posto singolo in una carrozza specifica, giusto?

Johnny: Sì.

44.2.3 La classe Ticket Office

Benjamin: Le classi che abbiamo appena visto - sono rappresentazioni di qualche tipo di JSON o XML?

Johnny: Sì, ma non dobbiamo scrivere il codice per farlo. Come ho detto prima, qualcuno se ne è già occupato.

Benjamin: Fortunati noi.

Johnny: Il nostro lavoro inizia dal punto in cui i dati deserializzati vengono passati all'applicazione come DTO. Il punto di ingresso della richiesta si trova in una classe chiamata `TicketOffice`:

```
[SomeKindOfController]
public class TicketOffice
{
    [SuperFrameworkMethod]
    public ReservationDto MakeReservation(ReservationRequestDto requestDto)
    {
        throw new NotImplementedException("Not implemented");
    }
}
```

Johnny: Vedo che la classe è decorata con attributi specifici di un framework web, quindi probabilmente non implementeremo il caso d'uso direttamente nel metodo `MakeReservation` per evitare di accoppiare la logica del nostro caso d'uso al codice che deve soddisfare i requisiti di un framework specifico.

Benjamin: Quindi quello che stai dicendo è che stai cercando di tenere la classe `TicketOffice` il più lontano possibile dalla logica dell'applicazione?

Johnny: Sì. Ne ho solo bisogno per racchiudere tutti i dati in astrazioni appropriate che progetto per soddisfare le mie preferenze, non il framework. Quindi, in queste astrazioni, posso codificare la mia soluzione nel modo che preferisco.

44.3 Bootstrap

Benjamin: Siamo pronti a partire?

Johnny: Solitamente, se fossi in te, mi piacerebbe vedere un posto in più nel codice.

Benjamin: Che è...?

Johnny: La "composition root", ovviamente.

Benjamin: Perché mi piacerebbe vedere la "composition root"?

Johnny: Il primo motivo è che è molto vicino al punto di ingresso dell'applicazione, quindi è un'opportunità per vedere come l'applicazione gestisce le sue dipendenze. Il secondo motivo è che ogni volta che aggiungeremo una nuova classe che abbia la stessa durata di vita dell'applicazione, dovremo andare alla "composition root " e modificarla. Quindi probabilmente mi piacerebbe sapere dov'è e come dovrei lavorarci.

Benjamin: Pensavo di poterlo trovare più tardi, ma già che ci siamo, puoi mostrarmi la "composition root"?

Johnny: Certo, è qui, nella classe `Application`:

```
public class Application
{
  public static void Main(string[] args)
  {
    new WebApp(
        new TicketOffice()
    ).Host();
  }
}
```

Benjamin: È bello vedere che non richiede l'uso di alcun meccanismo fantasioso reflection-based [*basato sulla riflessione*] per comporre gli oggetti.

Johnny: ì, siamo fortunati in questo. Possiamo semplicemente creare gli oggetti con l'operatore `new` e passarli al framework. Un'altra informazione importante è che abbiamo un'unica istanza di `TicketOffice` per gestire tutte le richieste. Ciò significa che non possiamo memorizzare alcuno stato relativo a una singola richiesta all'interno di questa istanza poiché entrerebbe in conflitto con altre richieste.

44.4 Scrivere il primo Statement

Johnny: Comunque, penso che siamo pronti per iniziare.

Benjamin: Ma da dove cominciamo? Dovremmo prima scrivere una sorta di classe chiamata "Reservation" o "Train"?

Johnny: No, quello che faremo è iniziare dagli input e procedere verso l'interno dell'applicazione. Poi, se necessario, agli output.

Benjamin: Non credo di capire di cosa stai parlando. Intendi questo approccio "outside-in" di cui hai parlato ieri?

Johnny: Sì, e non preoccuparti se non hai capito quello che ho detto, te lo spiegherò man mano che procediamo. Per ora, l'unica cosa che intendo con questo è che seguiremo il percorso della richiesta così come proviene dall'esterno della nostra applicazione e inizieremo a implementare in primo luogo laddove la richiesta non viene gestita come dovrebbe. Nello specifico, ciò significa che iniziamo da:

```
[SomeKindOfController]
public class TicketOffice
{
    [SuperFrameworkMethod]
    public ReservationDto MakeReservation(ReservationRequestDto requestDto)
    {
        throw new NotImplementedException("Not implemented");
    }
}
```

Benjamin: Perché?

Johnny: Perché questo è il luogo più vicino al punto di ingresso della richiesta in cui il comportamento è diverso da quello che ci aspettiamo. Non appena la richiesta raggiunge questo punto, la sua gestione verrà interrotta e verrà generata un'eccezione. Dobbiamo modificare questo codice se vogliamo che la richiesta vada oltre.

Benjamin: Capisco... quindi... se non avessimo già il codice di deserializzazione della richiesta, inizieremmo da lì perché sarebbe il primo posto in cui la richiesta rimarrebbe bloccata nel suo percorso verso il suo obiettivo, giusto?

Johnny: Sì, hai capito.

Benjamin: E... iniziamo con uno Statement falso, no?

Johnny: Sì, facciamolo.

44.4.1 Primo scheletro dello Statement

Benjamin: Non dirmi niente, proverò a farlo da solo.

Johnny: Certo, come vuoi.

Benjamin: La prima cosa che devo fare è aggiungere una Specifica vuota per la classe `TicketOffice`:

```
public class TicketOfficeSpecification
{
    //TODO add a Statement
}
```

Quindi, devo aggiungere il mio primo Statement. So che in questo Statement, devo creare un'istanza della classe `TicketOffice` e chiamare il metodo `MakeReservation`, poiché è l'unico metodo in questa classe e non è implementato.

Johnny: quindi quale strategia usi per iniziare con uno Statement falso?

Benjamin: "richiama un metodo quando ne hai uno", per quanto ricordo.

Johnny: Allora, come sarà il codice?

Benjamin: per cominciare, farò un brain dump proprio come mi hai insegnato. Dopo aver affermato tutti i fatti dannatamente ovvi, ottengo:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var ticketOffice = new TicketOffice();

  //WHEN
  ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.True(false);
}
```

Johnny: Bene... il mio... apprendista...

Benjamin: Cosa?

Johnny: Oh, non importa... comunque il codice non viene compilato adesso, dato che questa riga:

```
ticketOffice.MakeReservation(requestDto);
```

utilizza una variabile `requestDto` che non esiste. Generiamolo utilizzando il nostro IDE!

Benjamin: A proposito, volevo chiederti esattamente questa frase. Renderlo compilabile è qualcosa che dobbiamo fare per andare avanti. Non avremmo dovuto aggiungere un commento `TODO` per le cose a cui dobbiamo tornare? Proprio come abbiamo fatto con il nome Statement, che era:

```
public void ShouldXXXXX() //TODO better name
```

Johnny: La mia opinione è che questo non sia necessario, perché il compilatore, fallendo su questa riga, ha già creato una sorta di elemento TODO per noi, solo non nella nostra lista TODO ma nel log degli errori del compilatore. Questo è diverso da ad es. la necessità di cambiare il nome di un metodo, cosa che il compilatore non ci ricorderà.

Benjamin: Quindi la mia lista TODO è composta da errori di compilazione, false dichiarazioni e elementi che contrassegno manualmente come `TODO`? È così che dovrei capirlo?

Johnny: Esattamente. Tornando alla variabile `requestDto`, creiamola.

Benjamin: Certo. È venuta fuori così:

```
ReservationRequestDto requestDto;
```

Dobbiamo assegnare qualcosa alla variabile.

Johnny: Sì, e poiché è un DTO, non sarà certamente un mock.

Benjamin: Vuoi dire che non creiamo i mock dei DTO?

Johnny: No, non ce n'è bisogno. I DTO sono, per definizione, strutture di dati e il mocking implica un polimorfismo che si applica al comportamento piuttosto che ai dati. Più tardi lo spiegherò più nel dettaglio. Per ora fidati della mia parola.

Benjamin: Allora... se non vuole essere un mock, allora generiamo un valore anonimo utilizzando il metodo `Any.Instance<>()`.

Johnny: Questo è esattamente quello che farei.

Benjamin: Quindi cambierò questa riga:

```
ReservationRequestDto requestDto;
```

in:

```
var requestDto = Any.Instance<ReservationRequestDto>();
```

44.4.2 Impostazione della expectation

Johnny: Sì, e ora lo Statement viene compilato e sembra essere falso. Questo è dovuto a questa riga:

```
Assert.True(false);
```

Benjamin: quindi cambiamo questo `false` in `true` e abbiamo finito, giusto?

Johnny: ...Eh?

Benjamin: Oh, ti stavo solo prendendo in giro. Quello che volevo davvero dire è: trasformiamo questa asserzione in qualcosa di utile.

Johnny: Uff, non spaventarmi così. Sì, questa asserzione deve essere riscritta. E succede così che quando guardiamo la riga seguente:

```
ticketOffice.MakeReservation(requestDto);
```

non fa alcun uso del valore restituito da `MakeReservation()` mentre è evidente dalla firma che il suo tipo restituito è `ReservationDto`. Guarda:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
```

Nel nostro Statement, non ne facciamo nulla.

Benjamin: Fammi indovinare, vuoi che vada allo Statement, assegni questo valore di ritorno a una variabile e poi se ne asserisce l'uguaglianza con... cosa esattamente?

Johnny: Per ora, ad un valore previsto, che non conosciamo ancora, ma di questo ci preoccuperemo più tardi quando ci bloccherà davvero.

Benjamin: Questa è una di quelle situazioni in cui dobbiamo immaginare di avere già qualcosa che non abbiamo, giusto?. Ok, ecco qui:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var ticketOffice = new TicketOffice();

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  //doesn't compile - we don't have expectedReservationDto yet:
  Assert.Equal(expectedReservationDto, reservationDto);
}
```

Ecco, ho fatto quello che mi hai chiesto. Quindi, per favore, spiegami ora come ci siamo avvicinati al nostro obiettivo?

Johnny: Abbiamo trasformato il nostro problema da "quale affermazione scrivere" in "qual è la prenotazione che ci aspettiamo". Questo è davvero un passo nella giusta direzione.

Benjamin: Per favore illuminami allora: cos'è "la prenotazione che ci aspettiamo"?

Johnny: Per ora, lo Statement non si compila affatto, quindi per fare un ulteriore passo avanti, possiamo semplicemente introdurre un `expectedReservationDto` come oggetto anonimo. Pertanto, possiamo semplicemente scrivere nella sezione `GIVEN`:

```
var expectedReservationDto = Any.Instance<ReservationDto>();
```

e farà compilare il seguente codice:

```
//THEN
Assert.Equal(expectedReservationDto, reservationDto);
```

Benjamin: Ma questa asserzione fallirà comunque...

Johnny: È comunque meglio che non compilare, vero?

Benjamin: Beh, se la metti in questo modo... Ora il nostro problema è che il valore atteso dall'asserzione è qualcosa di cui il codice di produzione non è a conoscenza - è semplicemente qualcosa che abbiamo creato nel nostro Statement. Ciò significa che questa affermazione non asserisce l'esito del comportamento del codice di produzione. Come risolviamo questo problema?

Johnny: È qui che dobbiamo esercitare le nostre capacità di progettazione per introdurre alcuni nuovi collaboratori. Questo compito è difficile ai confini della logica applicativa in cui dobbiamo attirare i collaboratori non dal dominio, ma piuttosto pensare a pattern di progettazione che ci consentiranno di raggiungere i nostri obiettivi. Ogni volta che entriamo nella logica della nostra applicazione, lo facciamo pensando a un caso d'uso specifico. In questo esempio particolare, il nostro caso d'uso è "effettuare una prenotazione". Un caso d'uso è tipicamente rappresentato da un metodo in una facade[2] o un oggetto command[3]. I "command" sono un po' più complessi ma più scalabili. Se effettuare una prenotazione fosse il nostro unico caso d'uso, probabilmente non avrebbe senso utilizzarlo. Ma poiché abbiamo già richieste di funzionalità con priorità più alta, credo che possiamo supporre che i "command" siano più adatti.

Benjamin: Quindi proponi di utilizzare una soluzione più complessa - non è questo "big design up front"?

Johnny: Credo di no. Ricorda che sto usando solo una soluzione *un po'* più complessa. Il costo di implementazione è solo leggermente più alto così come il costo di manutenzione nel caso mi sbagliassi. Se per qualche motivo particolare qualcuno domani dicesse di non aver più bisogno del resto delle funzionalità, l'aumento di complessità sarà trascurabile tenendo conto delle dimensioni ridotte del la base di codice complessiva. Se, tuttavia, aggiungiamo più funzionalità, l'utilizzo dei comandi ci farà risparmiare tempo a lungo andare. Pertanto, dato quello che so, non lo sto aggiungendo per supportare nuove funzionalità speculative, ma per rendere il codice più facile da modificare a lungo termine. Sono d'accordo, tuttavia, che scegliere la complessità sufficiente per un dato momento è un compito difficile.

Benjamin: Ancora non capisco come l'introduzione di un "command" possa aiutarci in questo caso. Tipicamente, un "command" ha un metodo `Execute()` che non restituisce nulla. Come ci fornirà allora la risposta che dobbiamo restituire da `MakeReservation()`? Inoltre, c'è un altro problema: come verrà creato questo "command"? Probabilmente richiederà che la richiesta venga passata come uno dei parametri del costruttore. Ciò significa che non possiamo semplicemente passare il "command" al costruttore di `TicketOffice` poiché la prima volta che possiamo accedere alla richiesta è quando viene invocato il metodo `MakeReservation()`.

Johnny: Sì, sono d'accordo con entrambe le tue preoccupazioni. Fortunatamente, quando scegli di utilizzare i "command", in genere esistono soluzioni standard ai problemi che hai menzionato. I comandi possono essere creati utilizzando le factory e possono trasmettere i loro risultati utilizzando un pattern chiamato *collecting parameter*[4] - passeremo un oggetto all'interno del "command" per raccogliere tutti gli eventi dalla gestione del caso d'uso e quindi essere in grado di prepararci una risposta.

44.4.3 Introdurre un collaboratore con prenotazione in corso

Johnny: Iniziamo con il parametro di raccolta [collecting], che rappresenterà un concetto di dominio di una prenotazione in corso. Ne abbiamo bisogno per raccogliere i dati necessari per costruire una risposta DTO. Pertanto, ciò che attualmente sappiamo è che verrà convertito in una risposta DTO alla fine. Tutti e tre gli oggetti: il "command", i parametri di raccolta e la factory, sono collaboratori, quindi saranno mock nel nostro Statement.

Benjamin: Ok, fai strada.

Johnny: Iniziamo con la sezione `GIVEN`. Qui, dobbiamo dire che assumiamo che il parametro di raccolta mock, chiamiamolo `reservationInProgress`, ci fornirà il `expectedReservationDto` (che è già definito nel corpo dello Statement) quando richiesto:

[2] https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-facade-pattern
[3] https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-command-pattern
[4] Refactoring to Patterns, Joshua Kerievsky

```
//GIVEN
//...
reservationInProgress.ToDto().Returns(expectedReservationDto);
```

Naturalmente non abbiamo ancora `reservationInProgress`, quindi dobbiamo introdurlo. Come ho spiegato prima, questo deve essere un mock, perché altrimenti non saremmo in grado di chiamare `Returns()` su di esso:

```
///GIVEN
var reservationInProgress = Substitute.For<ReservationInProgress>();
//...
reservationInProgress.ToDto().Returns(expectedReservationDto);
//...
```

Ora, lo Statement non viene compilato perché l'interfaccia `ReservationInProgress` che ho appena utilizzato nella definizione mock non è ancora stata introdotta.

Benjamin: In altre parole, hai appena scoperto di aver bisogno di questa interfaccia.

Johnny: Esattamente. Quello che sto facendo attualmente è inserire astrazioni e oggetti nel mio codice quando ne ho bisogno. E la mia attuale esigenza è avere la seguente interfaccia nel mio codice:

```
public interface ReservationInProgress
{

}
```

Ora, lo Statement continua a non essere compilato, perché c'è questa riga:

```
reservationInProgress.ToDto().Returns(expectedReservationDto);
```

che richiede che l'interfaccia `ReservationInProgress` abbia un metodo `ToDto()`, ma per ora questa interfaccia è vuota. Dopo aver aggiunto il metodo richiesto, appare così:

```
public interface ReservationInProgress
{
  ReservationDto ToDto();
}
```

e lo Statement viene compilato di nuovo, sebbene sia ancora falso.

Benjamin: Ok. Ora concedetemi un secondo per cogliere l'intero Statement nel suo stato attuale.

Johnny: Certo, prenditi il tuo tempo, ecco come appare attualmente:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var ticketOffice = new TicketOffice();
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();

  reservationInProgress.ToDto().Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
```

(continues on next page)

(continua dalla pagina precedente)

```
  Assert.Equal(expectedReservationDto, reservationDto);
}
```

44.4.4 Vi presentiamo un collaboratore della factory

Benjamin: Penso di essere riuscito a recuperare. Posso prendere la tastiera?

Johnny: Stavo per suggerirlo. Ecco.

Benjamin: Grazie. Osservando questo Statement, abbiamo questo `ReservationInProgress` tutto configurato e creato, ma questo nostro mock non viene affatto passato a `TicketOffice`. Quindi, come dovrebbe `TicketOffice` utilizzare il nostro `reservationInProgress` preconfigurato?

Johnny: Ricordi la nostra discussione sulla separazione dell'uso degli oggetti dalla costruzione?

Benjamin: Immagino di sapere a cosa vuoi arrivare. Il `TicketOffice` dovrebbe in qualche modo ottenere dall'esterno un oggetto `ReservationInProgress` già creato. Può ottenerlo ad es. tramite un costruttore o da una factory.

Johnny: Sì, e se guardi l'ambito di durata del nostro `TicketOffice`, che viene creato una volta all'avvio dell'applicazione, non può accettare un `ReservationInProgress` tramite un costruttore, perché ogni volta che viene effettuata una nuova richiesta di prenotazione, vogliamo avere un nuovo `ReservationInProgress`, poi passarlo attraverso un costruttore `TicketOffice` ci costringerebbe a creare anche un nuovo `TicketOffice` ogni volta. Pertanto, la soluzione che meglio si adatta alla nostra situazione attuale è...

Benjamin: Una factory, giusto? Stai suggerendo che invece di passare un `ReservationInProgress` attraverso un costruttore, dovremmo piuttosto passare qualcosa che sappia come creare istanze di `ReservationInProgress`?

Johnny: Esattamente.

Benjamin: Come scriverlo nello Statement?

Johnny: Innanzitutto, scrivi solo ciò di cui hai bisogno. La factory deve essere un mock perché dobbiamo configurarlo in modo che, quando richiesto, restituisca il nostro mock `ReservationInProgress`. Quindi scriviamo prima la configurazione di ritorno, fingendo di avere già la factory disponibile nel corpo del nostro Statement.

Benjamin: Fammi vedere... questo dovrebbe bastare:

```
//GIVEN
...
reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
```

Johnny: Bello. Ora il codice non viene compilato, perché non abbiamo una `reservationInProgressFactory`. Quindi creiamolo.

Benjamin: E, come hai detto prima, dovrebbe essere un oggetto mock. Allora questa sarà la definizione:

```
var reservationInProgressFactory
  = Substitute.For<ReservationInProgressFactory>();
```

Per la necessità di questa riga di codice, ho fatto finta di avere un'interfaccia chiamata `ReservationInProgressFactory` e, lasciami indovinare, vuoi che introduca questa interfaccia adesso?

Johnny: (sorride)

Benjamin: Va bene. Ecco:

```
public interface ReservationInProgressRepository
{

}
```

E ora, il compilatore ci dice che non abbiamo il metodo `FreshInstance()`, quindi lasciami presentarlo:

```
public interface ReservationInProgressRepository
{
    ReservationInProgress FreshInstance();
}
```

Bene, il compilatore non si lamenta più, ma lo Statement fallisce con una `NotImplementedException`.

Johnny: Sì, questo perché il corpo attuale del metodo `MakeReservation()` della classe `TicketOffice` assomiglia a questo:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  throw new NotImplementedException("Not implemented");
}
```

44.4.5 Ampliamento del costruttore della biglietteria (ticket office)

Benjamin: Quindi dovremmo implementarlo adesso?

Johnny: Abbiamo ancora alcune cose da fare per lo Statement.

Benjamin: Tipo..?

Johnny: Ad esempio, il mock `ReservationInProgressFactory` che abbiamo appena creato non è ancora passato al costruttore `TicketOffice`, quindi non c'è modo per la biglietteria [ticket office] di utilizzare questa factory.

Benjamin: Ok, quindi lo aggiungo. Lo Statement cambierà in questo posto:

```
var reservationInProgressFactory
  = Substitute.For<ReservationInProgressFactory>();
var ticketOffice = new TicketOffice();
```

in:

```
var reservationInProgressFactory
  = Substitute.For<ReservationInProgressFactory>();
var ticketOffice = new TicketOffice(reservationInProgressFactory);
```

ed è necessario aggiungere un costruttore alla classe `TicketOffice`:

```
public TicketOffice(ReservationInProgressFactory reservationInProgressFactory)
{

}
```

Johnny: D'accordo. Inoltre, dobbiamo mantenere la "composition root" che ha appena interrotto la compilazione. Questo perché lì viene richiamato il costruttore di `TicketOffice` e anch'esso necessita di un aggiornamento:

```
public class Application
{
  public static void Main(string[] args)
  {
    new WebApp(
        new TicketOffice(/* compile error - instance missing */)
    ).Host();
  }
}
```

Benjamin: Ma cosa dovremmo passare? Abbiamo un'interfaccia, ma nessuna classe di cui potremmo fare un'istanza.

Johnny: Dobbiamo creare la classe. In genere, se ho un'idea sul nome della classe richiesta, creo la classe con quel nome. Se non ho ancora idea di come chiamarlo, posso chiamarlo ad es. `TodoReservationInProgressFactory` lasciando un commento TODO per tornarci più tardi. Per ora, abbiamo solo bisogno di questa classe per compilare il codice. E' ancora fuori dallo "scope" del nostro attuale Statement.

Benjamin: Quindi forse potremmo passare un `null` in modo da avere `new TicketOffice(null)`?

Johnny: Potremmo, ma non è la mia opzione preferita. In genere creo semplicemente la classe. Uno dei motivi è che la classe dovrà implementare un'interfaccia per compilare e quindi dovremo introdurre un metodo che, per default, lancerà una `NotImplementedException` e queste eccezioni finiranno anche nella mia lista TODO.

Benjamin: Ok, mi sembra ragionevole. Come questa riga:

```
new TicketOffice(/* compile error - instance missing */)
```

diventa:

```
new TicketOffice(
    new TodoReservationInProgressFactory()) //TODO change the name
```

E non viene compilato, perché dobbiamo creare questa classe, quindi lasciamelo fare:

```
public class TodoReservationInProgressFactory : ReservationInProgressFactory
{
}
```

Johnny: Ancora non viene compilato, perché l'interfaccia `ReservationInProgressFactory` ha alcuni metodi che dobbiamo implementare. Fortunatamente, possiamo farlo con un singolo comando IDE e ottenere:

```
public class TodoReservationInProgressFactory : ReservationInProgressFactory
{
    public ReservationInProgress FreshInstance()
    {
        throw new NotImplementedException();
    }
}
```

e, come ho detto un attimo fa, questa eccezione finirà nella mia lista delle cose da fare, ricordandomi che devo affrontarla.

Ora che il codice viene nuovamente compilato, torniamo al costruttore di `TicketOffice`:

```
public TicketOffice(ReservationInProgressFactory reservationInProgressFactory)
{

}
```

qui potremmo già assegnare il parametro del costruttore a un campo, ma va bene anche farlo in seguito.

Benjamin: Facciamolo più tardi, mi chiedo fino a che punto possiamo ritardare un lavoro come questo.

44.4.6 Introduzione di un collaboratore di "Command"

Johnny: Certo. Diamo quindi un'occhiata allo Statement che stiamo scrivendo. Sembra che manchi un'altra aspettativa nella nostra sezione `THEN`. se guardi il corpo completo dello Statement's così com'è ora:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var reservationInProgressFactory
```

(continues on next page)

(continua dalla pagina precedente)

```
    = Substitute.For<ReservationInProgressFactory>();
  var ticketOffice = new TicketOffice(reservationInProgressFactory);
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();

  reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
  reservationInProgress.ToDto().Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservation, reservationDto);
}
```

l'unica interazione tra `TicketOffice` e `ReservationInProgress` descritta è che il primo chiama il metodo `ToDto`sul secondo. Quindi la domanda che dobbiamo porci ora è "come farà l'istanza di `ReservationInProgress` a sapere cosa `ReservationDto` creare quando viene chiamato questo metodo?".

Benjamin: Oh giusto... `ReservationDto` deve essere creato da `ReservationInProgress` in base allo stato attuale dell'applicazione e ai dati in `ReservationRequestDto`, ma `ReservationInProgress` non sa nulla di queste cose finora.

Johnny: Sì, riempire `ReservationInProgress` è una delle responsabilità dell'applicazione che stiamo scrivendo. Se facessimo tutto nella classe `TicketOffice`, questa classe avrebbe sicuramente troppo da gestire e il nostro Statement aumenterebbe enormemente. Quindi dobbiamo estendere la responsabilità di gestire il nostro caso d'uso ad altri ruoli di collaborazione e utilizzare mock per svolgere tali ruoli qui.

Benjamin: Allora cosa proponi?

Johnny: Ricordi la nostra discussione di qualche minuto fa? Di solito, quando inserisco una logica relativa al caso d'uso su un altro oggetto al confine del sistema, scelgo tra il pattern Facade e il pattern Command. Le Facade sono più semplici ma meno scalabili, mentre i "command" sono molto più scalabili e componibili, ma l'applicazione deve creare un nuovo oggetto "command" ogni volta che viene attivato un caso d'uso e una nuova classe di comando deve essere creata da un programmatore quando il supporto per un un nuovo caso d'uso viene aggiunto al sistema.

Benjamin: So già che preferisci i "command".

Johnny: Sì, abbi pazienza, anche solo per vedere come possono essere usati i "command" qui. Sono sicuro che potresti capire da solo l'opzione Facade.

Benjamin: Cosa scrivo?

Johnny: Supponiamo di avere questo "command" e poi pensiamo a cosa vogliamo che ne faccia il nostro `TicketOffice`.

Benjamin: Vogliamo che `TicketOffice` esegua il "command", ovviamente..?

Johnny: Giusto, scriviamolo sotto forma di "expectation".

Benjamin: Ok, scriverei qualcosa del genere:

```
reservationCommand.Received(1).Execute(reservationInProgress);
```

Ho già passato `reservationInProgress` poiché il "command" dovrà riempirlo.

Johnny: Aspetta, si dà il caso che io preferisca un altro modo di passare questo `reservationInProgress` a `reservationCommand`. Per favore, per ora, rendi il metodo `Execute()` senza parametri.

Benjamin: Come desideri, ma ho pensato che questo sarebbe stato un buon posto per passarlo.

Johnny: Potrebbe sembrare, ma in genere voglio che i miei "command" abbiano metodi di esecuzione senza parametri. In questo modo posso comporli più liberamente, utilizzando pattern come decorator[5].

Benjamin: Come desideri. Ho ripristinato l'ultima modifica e ora la sezione `THEN` assomiglia a questa:

[5] https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-decorator-pattern

```
//THEN
reservationCommand.Received(1).Execute();
Assert.Equal(expectedReservationDto, reservationDto);
```

e ovviamente non viene compilato. `reservationCommand` non esiste, quindi devo introdurlo. Naturalmente, anche il tipo di questa variabile non esiste -- devo far finta che esista. E so già che dovrebbe essere un mock poiché specifico che avrebbe dovuto ricevere una chiamata al suo metodo `Execute()`.

Johnny: (annuisce)

Benjamin: Nella sezione `GIVEN`, aggiungerò il `reservationCommand` come un mock:

```
var reservationCommand = Substitute.For<ReservationCommand>();
```

Per il bene di questa riga, fingo di avere un'interfaccia chiamata `ReservationCommand` e ora devo crearla per compilare il codice.

```
public interface ReservationCommand
{

}
```

ma questo non è sufficiente, perché nello Statement mi aspetto di ricevere una chiamata a un metodo `Execute()` sul comando e non esiste un metodo del genere. Posso risolverlo aggiungendolo sull'interfaccia di "command":

```
public interface ReservationCommand
{
  void Execute();
}
```

e ora tutto si compila di nuovo.

44.4.7 Vi presentiamo un collaboratore della Factory Command

Johnny: Certo, ora dobbiamo capire come passare questo comando a `TicketOffice`. Come abbiamo discusso, per natura, un oggetto "command" rappresenta, beh, un comando emesso, quindi non può essere creato una volta nella "composition root" e poi passato al costruttore, perché in tal caso:

1. diventerebbe essenzialmente una facade,
2. dovremmo passare `reservationInProgress` al metodo `Execute()` che volevamo evitare.

Benjamin: Aspetta, non dirmi... vuoi aggiungere un'altra factory qui?

Johnny: Sì, è quello che mi piacerebbe fare.

Benjamin: Ma... questa è la seconda factory in un unico Statement. Non stiamo esagerando un po'?

Johnny: Capisco perché ti senti così. Tuttavia, questa è una conseguenza della mia scelta progettuale. Non avremmo bisogno di una factory di "command" se adottassimo una facade. Nelle app semplici, utilizzo semplicemente una facade ed elimino questo dilemma. Potrei anche eliminare l'uso del pattern del "collecting parameter" e quindi non avrei bisogno di una factory per le prenotazioni in corso, ma ciò significherebbe che non risolverei la violazione del principio di separazione comando-query e dovrei spingere ulteriormente questa violazione nel mio codice. Per tirarti su di morale, questo è un punto di ingresso per un caso d'uso in cui dobbiamo racchiudere alcune cose in astrazioni, quindi non mi aspetto così tante factory nel resto del codice. Tratto questa parte come una sorta di layer di adattamento che mi protegge da tutto ciò che è imposto dall'esterno della mia logica applicativa e che non voglio affrontare all'interno della mia logica applicativa.

Benjamin: Dovrò fidarmi di te su questo. Spero che questo renda le cose più facili in seguito perché per ora... ugh...

Johnny: Introduciamo la factory mock. Naturalmente, prima di definirlo, voglio usarlo nello Statement per sentirne il bisogno. Espanderò la sezione `GIVEN` partendo dal presupposto che una factory, alla richiesta di un "command", restituisca `reservationCommand`:

```
//GIVEN
...
commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
```

Questo non viene compilato perché non abbiamo ancora `commandFactory`.

Benjamin: Oh, vedo che `CreateNewReservationCommand()` della factory è il punto in cui hai deciso di passare il `reservationInProgress` che volevo passare al metodo `Execute()` in precedenza. Intelligente. Lasciando il metodo `Execute()` del "command" senza parametri, lo hai reso più astratto e hai reso l'interfaccia disaccoppiata da qualsiasi tipo di argomento particolare. D'altro canto, il "command" viene creato nello stesso ambito in cui viene utilizzato, quindi non vi è letteralmente alcun problema nel passare tutti i parametri tramite il metodo factory.

Johnny: Esatto. Ora sappiamo che abbiamo bisogno di una factory, inoltre che deve essere un mock poiché la configuriamo per restituire un "command" quando ne viene richiesto uno. Quindi dichiariamo la factory, fingendo di avere un'interfaccia per essa:

```
//GIVEN
...
var commandFactory = Substitute.For<CommandFactory>();
...
commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
```

Benjamin: ...e l'interfaccia `CommandFactory` non esiste, quindi creiamola:

```
public interface CommandFactory
{

}
```

e aggiungiamo il metodo `CreateNewReservationCommand` mancante:

```
public interface CommandFactory
{
  ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress);
}
```

Benjamin: Ora il codice viene compilato e assomiglia a questo:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var commandFactory = Substitute.For<CommandFactory>();
  var reservationInProgressFactory
    = Substitute.For<ReservationInProgressFactory>();
  var ticketOffice = new TicketOffice(reservationInProgressFactory);
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();
  var reservationCommand = Substitute.For<ReservationCommand>();

  commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
  reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
  reservationInProgress.ToDto().Returns(expectedReservationDto);
```

(continues on next page)

(continua dalla pagina precedente)

```
  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservationDto, reservationDto);
  reservationCommand.Received(1).Execute();
}
```

Benjamin: Vedo che il "command factory" non viene passato da nessuna parte nello Statement - `TicketOffice` non lo conosce.

Johnny: Sì e, per nostra fortuna, una factory è qualcosa che può avere la stessa durata di `TicketOffice` poiché per creare la factory non abbiamo bisogno di sapere nulla di una richiesta di prenotazione. Questo è il motivo per cui possiamo passarlo in sicurezza attraverso il costruttore di `TicketOffice`. Ciò significa che queste due righe:

```
var commandFactory = Substitute.For<CommandFactory>();
var ticketOffice = new TicketOffice(reservationInProgressFactory);
```

ora saranno come queso:

```
var commandFactory = Substitute.For<CommandFactory>();
var ticketOffice = new TicketOffice(
    reservationInProgressFactory, commandFactory);
```

Un costruttore con tale firma non esiste, quindi per effettuare questa compilazione dobbiamo aggiungere un parametro di tipo `CommandFactory` al costruttore:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{

}
```

che ci costringe ad aggiungere un parametro alla nostra "composition root". Quindi questa parte della "composition root":

```
new TicketOffice(
    new TodoReservationInProgressFactory()) //TODO change the name
```

diventa:

```
new TicketOffice(
    new TodoReservationInProgressFactory(), //TODO change the name
    new TicketOfficeCommandFactory())
```

che a sua volta ci obbliga a creare la classe `TicketOfficeCommandFactory`:

```
public class TicketOfficeCommandFactory : CommandFactory
{
  public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
      throw new NotImplementedException();
  }
}
```

Benjamin: Ehi, questa volta hai dato alla classe un nome migliore rispetto alla factory precedente che si chiamava `TodoReservationInProgressFactory`. Perché non hai voluto rimandare per più tardi questa volta?

Johnny: Questa volta penso di avere un'idea migliore su come chiamare questa classe. In genere, chiamo le classi concrete in base a qualcosa della loro implementazione e trovo difficile trovare i nomi quando non ho ancora questa implementazione. Questa volta credo di avere un nome che possa durare un po', anche per questo non ho lasciato un commento TODO accanto a questo nome. Tuttavia, ulteriore lavoro può invalidare la mia scelta del nome e sarò felice di cambiare il nome quando se ne presenterà la necessità. Per ora dovrebbe bastare.

44.4.8 Dare un nome allo Statement

Johnny: Comunque, tornando allo Statement, penso che abbiamo coperto tutto. Diamogli solo un bel nome. Considerando le asserzioni:

```
reservationCommand.Received(1).Execute();
Assert.Equal(expectedReservationDto, reservationDto);
```

Penso che possiamo dire:

```
public void
ShouldExecuteReservationCommandAndReturnResponseWhenAskedToMakeReservation()
```

Benjamin: Sono solo curioso... Non mi hai detto di fare attenzione alla parola "and" nei nomi degli Statement e che potrebbe suggerire che qualcosa non va nello scenario.

Johnny: Sì, e in questo caso particolare, *c'è* qualcosa che non va - la classe `TicketOffice` viola il principio di separazione comando-query. Questo è anche il motivo per cui lo Statement appare così confuso. Per questa classe, tuttavia, non abbiamo una grande scelta poiché il nostro framework richiede questo tipo di firma del metodo. Ecco perché stiamo lavorando così duramente in questo corso per introdurre il "collecting parameter" [*parametro di raccolta*] e proteggere il resto del progetto dalla violazione.

Benjamin: Spero che i futuri Statement siano più facili da scrivere di questo.

Johnny: Anch'io.

44.4.9 Rendere vero lo Statement - la prima asserzione

Johnny: Diamo un'occhiata al codice dello Statement:

```
[Fact] public void
ShouldExecuteReservationCommandAndReturnResponseWhenAskedToMakeReservation()
{
  //GIVEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var commandFactory = Substitute.For<CommandFactory>();
  var reservationInProgressFactory
    = Substitute.For<ReservationInProgressFactory>();
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();
  var reservationCommand = Substitute.For<ReservationCommand>();

  var ticketOffice = new TicketOffice(
    reservationInProgressFactory,
    commandFactory);

  reservationInProgressFactory.FreshInstance()
    .Returns(reservationInProgress);
  commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
  reservationInProgress.ToDto()
```

(continues on next page)

(continua dalla pagina precedente)

```
    .Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservationDto, reservationDto);
  reservationCommand.Received(1).Execute();
}
```

Johnny: Penso che sia completo, ma non lo sapremo finché non vedremo le asserzioni fallire e poi passare. Per ora, l'implementazione del metodo `MakeReservation()` lancia un'eccezione che fa sì che il nostro Statement si fermi alla fase `WHEN`, senza nemmeno arrivare alle asserzioni.

Benjamin: Ma non posso ancora inserire la giusta implementazione, giusto? Questo è quello che mi hai sempre detto.

Johnny: Sì, idealmente, dovremmo vedere gli errori delle asserzioni per avere la certezza che lo Statement *può* diventare falso quando il comportamento atteso non è in atto.

Benjamin: Questo può significare solo una cosa -- restituisce `null` da `TicketOffice` invece di generare un'eccezione. Giusto?

Johnny: Sì, lasciamelo fare. Cambierò semplicemente questo codice:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  throw new NotImplementedException("Not implemented");
}
```

in:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  return null;
}
```

Ora posso vedere che la prima affermazione:

```
Assert.Equal(expectedReservationDto, reservationDto);
```

fallisce, perché si aspetta un `expectedReservationDto` da `reservationInProgress` ma il `reservationInProgress` stesso può essere ricevuto solo dalla factory. Le righe pertinenti nello Statement che affermano ciò sono:

```
reservationInProgressFactory.FreshInstance()
    .Returns(reservationInProgress);
reservationInProgress.ToDto()
    .Returns(expectedReservationDto);
```

Implementiamo semplicemente la parte necessaria per superare questa prima asserzione. Per fare ciò, dovrò creare un campo nella classe `TicketOffice` per la factory e inizializzarlo con uno dei parametri del costruttore. Quindi questo codice:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{

}
```

diventa:

```
private readonly ReservationInProgressFactory _reservationInProgressFactory;

public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
  _reservationInProgressFactory = reservationInProgressFactory;
}
```

Benjamin: Non potresti semplicemente introdurre anche l'altro campo?

Johnny: Sì, potrei e di solito lo faccio. Ma visto che ci stiamo allenando, voglio dimostrarvi che saremo comunque costretti a farlo per far passare la seconda asserzione.

Benjamin: Vai avanti.

Johnny: Ora devo modificare il metodo `MakeReservation()` aggiungendo il seguente codice che crea la prenotazione in corso e le fa restituire un DTO:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
return reservationInProgress.ToDto();
```

Benjamin: ...e la prima affermazione passa.

Johnny: Sì, e questo chiaramente non è abbastanza. Se guardi il codice di produzione, non stiamo facendo nulla con l'istanza `ReservationRequestDto`. Per fortuna, abbiamo la seconda asserzione:

```
reservationCommand.Received(1).Execute();
```

e questo fallisce. Per farlo passare, dobbiamo creare un "command" ed eseguirlo.

44.4.10 Rendere lo Statement vero -- la seconda asserzione

Benjamin: Aspetta, perché la chiami "asserzione"? Non c'è una parola "asserire" da nessuna parte in questa riga. Non dovremmo chiamarla semplicemente "verifica mock" o qualcosa del genere?

Johnny: Sono d'accordo con la "verifica mock", tuttavia, ritengo corretto chiamarla anche un'asserzione, perché, in sostanza, è quello che è -- un controllo che genera un'eccezione quando una condizione non è soddisfatta.

Benjamin: OK, se è così che la metti...

Johnny: comunque, abbiamo ancora bisogno che questa asserzione passi.

Benjamin: Lasciamelo fare. Quindi, per creare un comando, abbiamo bisogno della "command factory". Questa factory è già passata a `TicketOffice` tramite il suo costruttore, quindi dobbiamo assegnarla a un campo perché d'ora in poi il costruttore assomiglierà a questo:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
  _reservationInProgressFactory = reservationInProgressFactory;
}
```

Dopo aver aggiunto l'assegnazione, il codice appare così:

```
private readonly CommandFactory _commandFactory;

public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  _reservationInProgressFactory = reservationInProgressFactory;
  _commandFactory = commandFactory;
}
```

e ora nel metodo `MakeReservation()`, posso chiedere alla factory di creare un "command":

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
return reservationInProgress.ToDto();
```

Ma questo codice non è ancora completo -- devo ancora eseguire il comando creato in questo modo:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

Johnny: Fantastico! Ora lo Statement è "true".

Benjamin: Wow, non è molto codice per uno Statement così grande che abbiamo scritto.

Johnny: La vera complessità non sta nemmeno nelle righe di codice, ma nel numero di dipendenze che abbiamo dovuto trascinare al suo interno. Tieni presente che abbiamo due factory qui. Ogni factory è una dipendenza e crea un'altra dipendenza. Questo si vede meglio nello Statement che nel metodo di produzione ed è per questo che trovo una buona idea prestare molta attenzione a come uno Statement sta crescendo e usarlo come meccanismo di feedback per la qualità del mio design. Per questa classe particolare, il problema di progettazione che osserviamo nello Statement non può essere risolto molto poiché, come ho già detto, questo è il confine in cui dobbiamo racchiudere le cose nelle astrazioni.

Benjamin: Dovrai spiegare meglio questo aspetto della qualità del design più avanti.

Johnny: Certo. Tè?

Benjamin: Caffè.

Johnny: Comunque, andiamo.

44.5 Riepilogo

È così che Johnny e Benjamin hanno realizzato il loro primo Statement utilizzando TDD e mock con un approccio progettuale "outside-in". Ciò che seguirà nel prossimo capitolo è una piccola retrospettiva con commenti su ciò che hanno fatto questi ragazzi. Una cosa che vorrei menzionare ora è che l'approccio "outside-in" non si basa esclusivamente su Statement a livello di unità, quindi quello che hai visto qui non è il quadro completo. Ci arriveremo presto.

CAPITOLO 45

Il test-driving ai confini di input -- una retrospettiva

Sono successe molte cose nell'ultimo capitolo e sento che alcune di esse meritano un approfondimento. Lo scopo di questo capitolo è quello di fare una piccola retrospettiva di ciò che Johnny e Benjamin hanno reso " test-driving" una classe "controller-type" per il sistema di prenotazione dei treni.

45.1 Sviluppo "outside-in"

Johnny e Benjamin hanno iniziato il loro sviluppo quasi interamente alla "periferia" del sistema, all'inizio del flusso di controllo. Questo è tipico dellapproccio "outside-in" dello sviluppo del software. Mi ci è voluto un po' per abituarmi. Per illustrarlo, consideriamo un sistema di tre classi, dove `Object3` dipende da `Object2` e `Object2` dipende da `Object1`:

```
Object3 -> Object2 -> Object1
```

Prima di adottare l'approccio "outside-in", la mia abitudine era quella di iniziare con gli oggetti alla fine della catena delle dipendenze (che in genere si trovavano anche alla fine del flusso di controllo) perché avevo tutto ciò di cui avevo bisogno per eseguirli e controllarli. Osservando il grafico sopra, ho potuto sviluppare ed eseguire la logica in `Object1` perché non richiedeva dipendenze. Poi, ho potuto sviluppare `Object2` perché dipendeva da `Object1` che avevo già creato e poi avrei potuto fare lo stesso con `Object3` perché dipendeva solo da `Object2` che avevo già. In qualsiasi momento potevo eseguire tutto ciò che avevo creato fino a quel momento.

L'approccio "outside-in" contraddiceva questa mia abitudine, perché gli oggetti da cui dovevo cominciare erano quelli che dovevano avere delle dipendenze e tali dipendenze ancora non esistevano. Guardando l'esempio sopra, dovrei iniziare con `Object3`, che non può essere istanziato senza `Object2`, il quale, a sua volta, non può essere istanziato senza `Object1`.

"Se sembra difficile, allora perché preoccuparsi?" ci si potrebbe chiedere. Le mie ragioni sono:

1. Partendo dagli input e andando all'interno, permetto che le mie interfacce e i miei protocolli siano modellati dai casi d'uso piuttosto che dalla tecnologia sottostante. Ciò non significa che posso sempre ignorare gli aspetti tecnologici, ma considero la logica del caso d'uso la guida principale. In questo modo, i miei protocolli tendono ad essere più astratti, il che a sua volta impone una maggiore componibilità.
2. Ogni riga di codice che introduco è lì perché il caso d'uso ne ha bisogno. Ogni metodo, ogni interfaccia e la classe esistono perché esiste già qualcuno che ne ha bisogno per adempiere ai propri obblighi. In questo modo, mi assicuro di implementare solo le cose necessarie e che siano modellate nel modo in cui gli utenti lo trovano comodo da usare. Prima di adottare questo approccio, avrei iniziato dall'interno del sistema e progettando le classi indovinando come sarebbero state utilizzate e in seguito spesso mi pentivo di queste ipotesi, a causa della rielaborazione e della complessità che avrebbero creato.

Per me questi pro superano i contro. Inoltre, ho scoperto di poter mitigare la sensazione di disagio di partire dagli input ("non c'è nulla che io possa eseguire completamente") con le seguenti pratiche:

1. Utilizzo del TDD con i mock -- il TDD incoraggia l'esecuzione di ogni piccola parte di codice ben prima del completamento dell'intera attività. Gli oggetti mock fungono da primi collaboratori che consentono che questa esecuzione avvenga. Per sfruttare i mock, almeno in C#, è necessario utilizzare le interfacce in modo più libero rispetto a un tipico approccio di sviluppo. Una proprietà delle interfacce è che in genere non contengono implementazione[1], quindi possono essere utilizzate per tagliare la catena delle dipendenze. In altre parole, mentre senza interfacce avrei bisogno di tutti e tre: `Object1`, `Object2` e `Object3` per istanziare e utilizzare `Object3`, potrei in alternativa introdurre un'interfaccia da cui dipenda `Object3` e `Object2` lo implementerebbe. Ciò mi consentirebbe di utilizzare `Object3` prima che esistano i suoi collaboratori concreti, semplicemente fornendo un oggetto mock come dipendenza.
2. Suddividere lo "scope" del lavoro in parti verticali più piccole (ad esempio scenari, storie, ecc.) che possono essere implementate più velocemente di una funzionalità completa. Abbiamo avuto un assaggio di questa azione quando Johnny e Benjamin stavano sviluppando la calcolatrice in uno dei primi capitoli di questo libro.
3. Non iniziare con uno Statement a livello di unità, ma scrivere prima a un livello superiore (ad esempio end-to-end o rispetto a un altro confine architetturale). Potrei quindi far funzionare questo Statement più grande, poi rifattorizzare gli oggetti iniziali da questo piccolo pezzo di codice funzionante. Solo dopo aver messo a punto questa struttura iniziale inizierei a utilizzare dichiarazioni a livello di unità con i mock. Questo approccio è ciò a cui mirerò alla fine, ma per questo e molti altri capitoli voglio concentrarmi sui mock e sulla progettazione object-oriented, quindi ho lasciato questa parte per dopo.

45.2 Specifica del flusso di lavoro

Lo Statement sul controller è un esempio di ciò che Amir Kolsky e Scott Bain chiamano Statement del flusso di lavoro[2]. Questo tipo di Statement descrive come una specifica unità di comportamento (nel nostro caso, un oggetto) interagisce con altre unità inviando messaggi e ricevendo risposte. Negli Statement che specificano il flusso di lavoro, documentiamo lo scopo previsto e i comportamenti della classe specificata in termini di interazione con altri ruoli nel sistema. Utilizziamo oggetti mock per svolgere questi ruoli specificando i valori restituiti di alcuni metodi e asserendo che vengano chiamati altri metodi.

Ad esempio, nello Statement scritto da Johnny e Benjamin nell'ultimo capitolo, hanno descritto come reagisce una "command factory" quando viene richiesto un nuovo "command" e hanno anche asserito la chiamata al metodo `Execute()` del "command". Quella era la descrizione di un workflow [*flusso di lavoro*].

45.2.1 Devo verificare che la factory sia stata chiamata?

Si nota nello stesso Statement che alcune interazioni sono state verificate (usando la sintassi `.Received()`) mentre alcune erano impostate solo per restituire qualcosa. Un esempio di quest'ultimo è una factory, ad es.

```
reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);`
```

Ci si potrebbe chiedere perché Johnny e Benjamin non hanno scritto qualcosa come `reservationInProgressFactory.Received().FreshInstance()` alla fine.

Uno dei motivi è che una factory somiglia a una funzione -- non dovrebbe avere effetti collaterali visibili. Pertanto, chiamare la factory non è l'obiettivo del comportamento che ho specificato -- sarà sempre solo un mezzo per raggiungere un fine. Ad esempio, l'obiettivo del comportamento specificato da Johnny e Benjamin era eseguire il comando e restituirne il risultato. La factory è stata creata per rendere più facile arrivarci.

Inoltre, Johnny e Benjamin hanno permesso che la factory venisse chiamata più volte durante l'implementazione senza alterare il comportamento previsto nello Statement. Ad esempio, se il codice del metodo `MakeReservation()` che stavano testando non fosse simile a questo:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
```

(continues on next page)

[1] sebbene sia C# che Java nelle loro versioni attuali consentano di inserire la logica nelle interfacce. Tuttavia, a causa delle convenzioni e di alcuni vincoli, le interfacce in questi linguaggi sono ancora considerate come esseri senza implementazione.

[2] http://www.sustainabletdd.com/2012/02/testing-best-practices-test-categories.html

(continua dalla pagina precedente)

```
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

ma così:

```
// Repeated multiple times:
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();

var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

allora il comportamento di questo metodo sarebbe comunque corretto. Certo, farebbe del lavoro inutile, ma quando scrivo gli Statement, mi preoccupo del comportamento visibile esternamente, non delle linee di codice di produzione. Lascio più libertà all'implementazione e cerco di non specificare eccessivamente.

D'altra parte, si consideri il "command" -- dovrebbe avere un effetto collaterale, perché mi aspetto che alla fine alteri una sorta di registro delle prenotazioni. Quindi, se invio il messaggio `Execute()` più di una volta in questo modo:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

allora potrebbe eventualmente alterare il comportamento -- magari riservando più posti di quelli richiesti dall'utente, magari lanciando un errore dal secondo `Execute()`... Questo è il motivo per cui voglio specificare rigorosamente quante volte deve essere inviato il messaggio `Execute()`:

```
reservationCommand.Received(1).Execute();
```

L'approccio alla specifica delle funzioni e degli effetti collaterali che ho descritto sopra è quello che Steve Freeman e Nat Pryce chiamano "Consenti query; aspettati comandi"[3]. Secondo la loro terminologia, la factory è una query -- una logica priva di effetti collaterali che restituisce un qualche tipo di risultato. Il `ReservationCommand`, d'altro canto, è un "command" - che non produce alcun tipo di risultato, ma causa un effetto collaterale come un cambiamento di stato o un'operazione di I/O.

45.3 Oggetti di Trasferimento Dati e TDD

Osservando le strutture dati iniziali, Johnny e Benjamin le chiamarono Data Transfer Objects [*Oggetti di Trasferimento Dati*]

Un Data Transfer Object è un pattern per descrivere gli oggetti responsabili dello scambio di informazioni tra processi[4]. Poiché i processi non possono realmente scambiare oggetti, lo scopo dei DTO è quello di essere serializzati in un qualche tipo di formato dati e poi trasferiti in quella forma a un altro processo che deserializza i dati. Quindi, possiamo avere DTO che rappresentano l'output che il nostro processo invia e i DTO che rappresentano l'input che il nostro processo riceve.

[3] Steve Freeman, Nat Pryce, Growing Object Oriented Software Guided By Tests
[4] Patterns of Enterprise Application Architecture, Martin Fowler

Come visto, i DTO sono in genere solo strutture dati. Ciò potrebbe sorprendere, perché ormai da diversi capitoli ho ripetutamente sottolineato come preferisco raggruppare insieme dati e comportamento. Questo non significa violare tutti i principi che ho citato?

La mia risposta a ciò sarebbe che lo scambio di informazioni tra processi è il luogo in cui i principi menzionati non si applicano e che ci sono alcune buone ragioni per cui i dati vengono scambiati tra processi.

1. È più facile scambiare dati che scambiare comportamenti. Se volessi inviare il comportamento a un altro processo, dovrei comunque inviarlo come dati, ad es. sotto forma di codice sorgente. In tal caso, l'altra parte dovrebbe interpretare il codice sorgente, fornire tutte le dipendenze, ecc., il che potrebbe essere complicato e accoppiare fortemente le implementazioni di entrambi i processi.
2. Concordare un formato dati semplice semplifica la creazione e l'interpretazione dei dati in diversi linguaggi di programmazione.
3. Molte volte, i confini tra i processi sono progettati allo stesso tempo come confini funzionali.e same time. In altre parole, anche se un processo inviasse dei dati a un altro, entrambi questi processi non vorrebbero eseguire lo stesso comportamento sui dati.

Questi sono alcuni dei motivi per cui i processi si scambiano dati. E quando lo fanno, in genere raggruppano i dati in strutture più grandi per motivi di coerenza e prestazioni.

45.3.1 DTO e *oggetti valore*

Sebbene i DTO, analogamente agli *oggetti valore*, trasportino e rappresentino dati, il loro scopo e i vincoli di progettazione sono diversi.

1. I valori hanno una semantica di valore, cioè possono essere confrontati in base al loro contenuto. Questo è uno dei principi fondamentali del loro design. I DTO non hanno bisogno di avere una semantica di valore (se aggiungo una semantica di valore ai DTO, lo faccio perché lo trovo conveniente per qualche motivo, non perché fa parte del "domain model").
2. I DTO devono essere facilmente serializzabili e deserializzabili da qualche tipo di formato di scambio dati (ad esempio JSON o XML).
3. I valori possono contenere comportamenti, anche piuttosto complessi (un esempio di ciò potrebbe essere il metodo `Replace()` della classe `String`), mentre i DTO tipicamente non contengono alcun comportamento.
4. Nonostante il punto precedente, i DTO possono contenere *oggetti valore*, purché questi *oggetti valore* possano essere serializzati e deserializzati in modo affidabile senza perdita di informazioni. Gli *oggetti valore* non contengono DTO.
5. I valori rappresentano concetti atomici e ben definiti (come testo, data, denaro), mentre i DTO funzionano principalmente come insiemi di dati.

45.3.2 I DTO e i mock

Come abbiamo osservato nellesempio di Johnny e Benjamin in cui scrivono il loro primo Statement, non hanno fatto mock dei DTO. È una regola generale -- un DTO è un dato, non rappresenta un'implementazione di un protocollo astratto né beneficia del polimorfismo come fanno gli oggetti. Inoltre, in genere è molto più semplice creare un'istanza di DTO piuttosto che un suo mock. Immaginiamo di avere il seguente DTO:

```
public class LoginDto
{
  public LoginDto(string login, string password)
  {
    Login = login;
    Password = password;
  }

  public string Login { get; }
  public string Password { get;}
}
```

Un'istanza di questa classe può essere creata digitando:

```
var loginDto = new LoginDto("James", "007");
```

Se dovessimo creare un mock, probabilmente avremmo bisogno di estrarre un'interfaccia:

```
public class ILoginDto
{
  public string Login { get; }
  public string Password { get; }
}
```

e poi scrivere qualcosa del genere nel nostro Statement:

```
var loginDto = Substitute.For<ILoginDto>();
loginDto.Login.Returns("James");
loginDto.Password.Returns("Bond");
```

Non solo è più prolisso, ma non ci apporta nulla. Da qui il mio consiglio:

{{keyToDo}} Non cercare di creare mock di un DTO negli Statement. Crea la cosa reale.

45.3.3 Creazione di DTO negli Statement

Poiché i DTO tendono a raggruppare dati, crearli per Statement specifici potrebbe essere un compito ingrato poiché a volte potrebbero esserci diversi campi che dovremmo inizializzare in ciascuno Statement. Come mi approccio alla creazione di istanze di DTO per evitare questo? Ho riassunto i miei consigli su come affrontare questo problema nell'elenco in ordine di priorità riportato di seguito:

1. Limit the reach of your DTOs in the production code

Come regola generale, meno conosco sui tipi e sui metodi, meglio è. I DTO rappresentano un contratto di applicazione esterna. Sono inoltre vincolati da alcune regole menzionate in precedenza (come la facilità di serializzazione), quindi non possono evolversi allo stesso modo degli oggetti normali. Pertanto, cerco di limitare al minimo necessario il numero di oggetti che conoscono i DTO nella mia applicazione. Utilizzo una delle due strategie: wrap o mapping.

Nel wrapping, ho un altro oggetto che contiene un riferimento al DTO e quindi tutti gli altri elementi logici interagiscono con questo oggetto che avvolge invece che direttamente con un DTO:

```
var user = new User(userDto);
//...
user.Assign(resource);
```

Considero questo approccio più semplice ma più limitato. Trovo che mi incoraggi a modellare gli oggetti del dominio in modo simile a come sono progettati i DTO (perché un oggetto avvolge un DTO). Di solito inizio con questo approccio quando il contratto esterno è abbastanza vicino al mio modello di dominio e passo all'altra strategia -- il mapping -- quando la relazione inizia a diventare più complessa.

Nel mapping, scompatto il DTO e passo parti specifiche negli oggetti del mio dominio:

```
var user = new User(
  userDto.Name,
  userDto.Surname,
  new Address(
    userDto.City,
    userDto.Street));
//...
user.Assign(resource);
```

Questo approccio mi richiede di riscrivere i dati in nuovi oggetti campo per campo, ma in cambio mi lascia più spazio per modellare i miei oggetti di dominio indipendentemente dalla struttura DTO[5]. Nell'esempio sopra, ho potuto introdurre un'astrazione `Address` anche se il DTO non ha un campo esplicito contenente l'indirizzo.

In che modo tutto ciò mi aiuta a evitare la noiosità della creazione di DTO? Bene, meno oggetti e metodi conoscono un DTO, meno Statement dovranno conoscere, il che porta a meno posti in cui devo crearne e inizializzarne uno.

2. Use constrained non-determinism if you don't need specific data

In molti Statement in cui ho bisogno di creare DTO, i valori specifici contenuti al loro interno non hanno importanza per me. Mi interessa solo che *alcuni* dati siano presenti. Per situazioni come questa, mi piace usare il non-determinismo vincolato Posso semplicemente creare un'istanza anonima e utilizzarla, cosa che trovo più semplice rispetto all'assegnazione campo per campo.

Ad esempio, guardiamo la seguente riga della dichiarazione che Johnny e Benjamin hanno scritto nell'ultimo capitolo:

```
var requestDto = Any.Instance<ReservationRequestDto>();
```

In quello Statement, non avevano bisogno di preoccuparsi dei valori esatti detenuti dal DTO, quindi hanno semplicemente creato un'istanza anonima. In questo caso particolare, l'utilizzo del non-determinismo vincolato non solo ha semplificato la creazione della DTO, ma ha anche permesso di disaccoppiare completamente lo Statement dalla struttura del DTO.

3. Use patterns such as factory methods or builders

Quando tutto il resto fallisce, utilizzo metodi factory e builder di dati di test per alleviare il lavoro della creazione di DTO per nascondere la complessità e fornire alcuni buoni valori di default per le parti che non mi interessano.

Un metodo factory può essere utile se c'è un unico fattore distintivo nell'istanza particolare che desidero creare. Per esempio:

```
public UserDto AnyUserWith(params Privilege[] privileges)
{
  var dto = Any.Instance<UserDto>()
    .WithPropertyValue(user => user.Privileges, privileges);
  return dto;
}
```

Questo metodo crea qualsiasi utente con un particolare insieme di privilegi. Si noti che in questo metodo ho utilizzato anche il non-determinismo vincolato, il che mi ha risparmiato del codice di inizializzazione. Se ciò non è possibile, provo a trovare una sorta di valori "predefiniti sicuri" per ciascuno dei campi.

Mi piacciono i metodi factory, ma maggiore è la flessibilità di cui ho bisogno, più gravito verso i builder di dati di test[6].

Un builder di dati di test è un oggetto speciale che mi consente di creare un oggetto con alcuni valori di default, ma mi consente di personalizzare la ricetta predefinita in base alla quale viene creato l'oggetto. Mi lascia molta più flessibilità nel modo in cui imposto i miei DTO. La sintassi tipica per l'utilizzo di un builder si può trovare su Internet[7] è simile alla seguente:

```
var user = new UserBuilder().WithName("Johnny").WithAge("43").Build();
```

Notare che il valore per ciascun campo è configurato separatamente. In genere, il builder contiene una sorta di valori di default per i campi che non specifico:

```
//some safe default age will be used when not specified:
var user = new UserBuilder().WithName("Johnny").Build();
```

Non mostrerò di proposito un esempio di implementazione, perché in uno dei prossimi capitoli ci sarà una discussione più lunga sui builder di dati di test.

[5] https://martinfowler.com/eaaCatalog/mapper.html
[6] http://www.natpryce.com/articles/000714.html
[7] https://blog.ploeh.dk/2017/08/15/test-data-builders-in-c/

45.4 Usare una ReservationInProgress

Un punto controverso della progettazione nell'ultimo capitolo potrebbe essere l'utilizzo di una classe `ReservationInProgress`. L'idea centrale di questa astrazione è raccogliere i dati necessari per produrre un risultato. Per introdurre questo oggetto avevamo bisogno di una factory separata, il che ha reso il design più complesso. Pertanto, alcune domande potrebbero sorgere:

1. Cos'è esattamente `ReservationInProgress`?
2. la `ReservationInProgress` è davvero necessaria e, in caso contrario, quali sono le alternative?
3. È necessaria una factory separata per `ReservationInProgress`?

Proviamo a rispondere.

45.4.1 Cos'è esattamente ReservationInProgress?

Come accennato in precedenza, l'intento di questo oggetto è quello di raccogliere informazioni sull'elaborazione di un comando, in modo che l'emittente del comando (nel nostro caso, un oggetto controller) possa agire su tali informazioni (ad esempio utilizzarle per creare una risposta) alla fine dell'elaborazione. Parlando nel linguaggio dei pattern, questa è un'implementazione di un pattern Collecting Parameter[8].

C'è una cosa che faccio spesso, ma non ho messo l'esempio per ragioni di semplicità. Quando implemento un "collecting parameter", in genere lo faccio implementando due interfacce -- una più stretta e l'altra -- più ampia. Lasciate che lo mostri:

```
public interface ReservationInProgress
{
   void Success(SomeData data);
   //... methods for reporting other events
}

public interface ReservationInProgressMakingReservationDto
  : ReservationInProgress
{
  ReservationDto ToDto();
}
```

Il punto è che solo chi ha emesso il comando può vedere l'interfaccia più ampia (`ReservationInProgressMakingReservationDto`) e quando passa questa interfaccia lungo la catena di chiamate, l'oggetto successivo vede solo i metodi per segnalare eventi (`ReservationInProgress`). In questo modo, linterfaccia più ampia può anche essere legata a una tecnologia specifica, a meno che quella più stretta non lo sia. Ad esempio, se avessi bisogno di una risposta tramite stringa JSON, potrei fare qualcosa del genere:

```
public interface ReservationInProgressMakingReservationJson
  : ReservationInProgress
{
  string ToJsonString();
}
```

e solo l'emittente del comando (nel nostro caso, l'oggetto controller) ne sarebbe a conoscenza. Il resto delle classi che utilizzano l'interfaccia più ristretta interagirebbero felicemente con essa senza mai sapere che è destinata a produrre un output JSON.

45.4.2 È necessario ReservationInProgress?

In breve -- no, anche se lo trovo utile. Esistono diversi design alternativi.

Prima di tutto, potremmo decidere di ritornare dal metodo `Execute()` del comando. Quindi, il comando sarebbe simile a questo:

[8] https://wiki.c2.com/?CollectingParameter

```
public interface ReservationCommand
{
  public ReservationDto Execute();
}
```

Ciò farebbe il lavoro per l'attività in questione, ma farebbe sì che `ReservationCommand` infranga il principio di separazione comando-query, che mi piace sostenere il più possibile. Inoltre, l'interfaccia `ReservationCommand` diventerebbe molto meno riutilizzabile. Se la nostra applicazione dovesse supportare diversi comandi, ciascuno dei quali restituisse un diverso tipo di risultato, non potremmo avere un'unica interfaccia per tutti. Ciò, a sua volta, renderebbe più difficile decorare i comandi utilizzando il pattern decorator. Potremmo provare a risolvere questo problema rendendo il comando generico:

```
public interface ReservationCommand<T>
{
  public T Execute();
}
```

ma questo lascia ancora una distinzione tra comandi `void` e non-`void` (che alcune persone risolvono parametrizzando i potenziali comandi `void` con `bool` e restituendo sempre `true` alla fine).

La seconda opzione per evitare un "collecting parameter" sarebbe semplicemente quella di lasciare che il comando venga eseguito e poi ottenere il risultato interrogando lo stato che è stato modificato dal comando (analogamente a un comando, una query può essere un oggetto separato). Il codice di `MakeReservation()` sarebbe simile a questo:

```
var reservationId = _idGenerator.GenerateId();
var reservationCommand = _factory.CreateNewReservationCommand(
  requestDto, reservationId);
var reservationQuery = _factory.CreateReservationQuery(reservationId);

reservationCommand.Execute();
return reservationQuery.Make();
```

Notare che in questo caso non c'è niente come "risultato in corso", ma d'altra parte dobbiamo generare l'id per il comando, poiché la query deve utilizzare lo stesso id. Questo approccio potrebbe interessare se:

1. Non importa che se si memorizzano le modifiche in un database o in un servizio esterno, la logica potrebbe doverlo chiamare due volte -- una volta durante il comando e un'altra durante la query.
2. Lo stato modificato dal comando è interrogabile (ovvero una potenziale API di destinazione per i dati consente di eseguire sia comandi che query sui dati). Non è sempre scontato.

Ci sono altre opzioni, ma vorrei fermarmi qui perché questa non è la preoccupazione principale di questo libro.

45.4.3 Abbiamo bisogno di una factory separata per `ReservationInProgress`?

Questa domanda può essere suddivisa in due parti:

1. Possiamo usare la stessa factory dei comandi?
2. Abbiamo davvero bisogno di una factory?

La risposta alla prima è: dipende da cosa è accoppiato `ReservationInProgress`. In questo esempio specifico, si tratta semplicemente di creare un DTO da restituire al client. In tal caso, non è necessaria alcuna conoscenza del framework utilizzato per eseguire l'applicazione. Questa mancanza di accoppiamento al framework mi consentirebbe di collocare la creazione di `ReservationInProgress` nella stessa factory. Tuttavia, se questa classe avesse bisogno di decidere, ad es. codici di stato HTTP o creare risposte richieste da un framework specifico o in un formato specificato (ad esempio JSON o XML), allora opterei, come hanno fatto Johnny e Benjamin, per separarlo dalla factory dei "command". Questo perché la "command factory" appartiene al mondo della logica applicativa e voglio che la logica della mia applicazione sia indipendente dal framework, dai protocolli di trasporto e dai formati dei payload usati.

La risposta alla seconda domanda (se abbiamo davvero bisogno di una factory) è: dipende se interessa specificare il comportamento del controller a livello di unità. Se sì, allora potrebbe essere utile avere una factory per controllare la creazione di `ReservationInProgress`. Se non interessa (ad esempio, si guida questa logica con una specifica di livello

superiore, di cui parleremo in una delle parti successive), si può decidere semplicemente di creare l'oggetto all'interno del metodo controller o anche di creare il controller stesso che implementa l'interfaccia `ReservationInProgress` (anche se, se lo si facesse, ci si dovrebbe assicurare che una singola istanza del controller non sia condivisa tra più richieste, poiché conterrebbe uno stato mutabile).

Devo specificare il comportamento del controller a livello di unità?

Come ho già detto, dipende dalle proprie scelte. Poiché i controller spesso fungono da adattatori tra un framework e la logica dell'applicazione, sono vincolati dal framework e quindi non ha molto valore nel guidare la loro progettazione con istruzioni a livello di unità. Potrebbe essere più accurato dire che questi Statement sono più vicini alle *Specifiche di integrazione* perché spesso descrivono come l'applicazione è integrata in un framework.

Un'alternativa alla specifica dell'integrazione a livello "unit" potrebbe essere quella di guidarla con Statement di livello superiore (di cui parlerò in dettaglio nelle prossime parti di questo libro). Ad esempio, potrei scrivere uno Statement che descrive come funziona la mia applicazione end-to-end, poi scrivere il codice di un controller (o un altro adattatore del framework) senza uno Statement a livello di unità dedicata e quindi utilizzare Statement a livello di unità per guidare la logica dell'applicazione. Quando lo Statement end-to-end passa, significa che l'integrazione che il mio controller avrebbe dovuto fornire funziona.

Ci sono alcuni prerequisiti per questo approccio al lavoro, ma li tratterò quando parlerò degli Statement di livello superiore nelle parti successive di questo libro.

45.5 Scoperta dell'interfaccia e sorgenti di astrazioni

Come promesso, il capitolo precedente includeva alcune scoperte sull'interfaccia, anche se non era un modo tipico di applicare questo approccio. Questo perché, come ho accennato, l'obiettivo dello Statement era quello di descrivere l'integrazione tra il framework e il codice dell'applicazione. Tuttavia, la stesura dello Statement ha permesso a Johnny e Benjamin di scoprire di cosa ha bisogno il loro controller dall'applicazione per trasmetterle tutti i dati necessari e ottenere il risultato senza violare il principio di separazione comando-query.

Le diverse astrazioni a cui Johnny ha dato vita sono state guidate da:

- il suo bisogno di convertirsi a uno stile di programmazione diverso (principio di separazione comandi-query, "tell, don't ask"...),
- la sua conoscenza dei "design pattern",
- la sua esperienza con problemi simili.

Se gli fossero mancate queste cose, probabilmente avrebbe scritto la cosa più semplice che gli sarebbe venuta in mente e poi avrebbe cambiato il progetto dopo averne appreso di più.

45.6 Ho bisogno di tutto questo per fare TDD?

L'ultimo capitolo potrebbe aver lasciato un po' di confusione. Se ci si sta chiedendo se è necessario utilizzare controller, "collecting parameter", comandi e factory per eseguire TDD, la mia risposta è *non necessariamente*. Inoltre, ci sono molti dibattiti su Internet se questi particolari pattern siano quelli giusti da usare e anche a me non piace usare controller con problemi come questo (anche se l'ho usato perché è ciò che molti framework web offrono come scelta di default).

Avevo bisogno di tutto questo per creare un esempio che somigliasse a un problema della vita reale. La lezione più importante, tuttavia, è che, mentre prendere le decisioni di progettazione dipendeva dalla conoscenza e dall'esperienza di Johnny e Benjamin, scrivere il codice dello Statement prima dell'implementazione *informava* queste decisioni di progettazione, aiutandoli a distribuire le responsabilità tra gli oggetti che collaboravano.

45.7 Qual è il prossimo?

Si spera che questo capitolo abbia collegato tutti i punti mancanti dell'ultimo. Il prossimo capitolo si concentrerà sulla creazione di "oggetti test-driving".

CAPITOLO 46

Creazione di oggetti "test-driving"

In questo capitolo ci uniamo a Johnny e Benjamin mentre continuano il loro viaggio fuori dal TDD. In questo capitolo faranno del "test-driving " per una factory.

Johnny: Qual è il prossimo elemento della nostra lista dei TODO?

Benjamin: Ci sono due istanze di `NotImplementedException`, entrambe provenienti da factory che abbiamo scoperto durante l'implementazione dell'ultimo Statement.

La prima factory serve per creare istanze di `ReservationInProgress` -- ricordi? Gli hai dato un nome strano apposta. Guarda:

```
public class TodoReservationInProgressFactory : ReservationInProgressFactory
{
  public ReservationInProgress FreshInstance()
  {
   throw new NotImplementedException();
  }
}
```

Johnny: Oh, sì, quello...

Benjamin: e la seconda factory serve per creare comandi -- si chiama `TicketOfficeCommandFactory`:

```
public class TicketOfficeCommandFactory : CommandFactory
{
  public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   throw new NotImplementedException();
  }
}
```

Johnny: Facciamolo.

Benjamin: Perché questo?

Johnny: Nessun motivo in particolare. Voglio solo approfondire la logica del dominio il prima possibile.

Benjamin: Giusto. Non abbiamo una Specifica per questa classe, quindi creiamone una.

```
public class TicketOfficeCommandFactorySpecification
{

}
```

Johnny: Continua, penso che sarai in grado di scrivere questo Statement senza il mio aiuto.

Benjamin: L'inizio sembra facile. Aggiungerò un nuovo Statement che non so ancora come chiamerò.

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  Assert.True(false);
 }
}
```

Ora, come mi hai detto prima, so già quale classe sto descrivendo -- è `TicketOfficeCommandFactory`. Implementa un'interfaccia che ho scoperto nello Statement precedente, quindi ha già un metodo: `CreateNewReservationCommand`. Quindi nello Statementcreerò semplicemente un'istanza della classe e chiamerò il metodo, perché "why not"?

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  factory.CreateNewReservationCommand();

  //THEN
  Assert.True(false);
 }
}
```

Questo non viene compilato. Per scoprire perché, guardo la firma del metodo `CreateNewReservationCommand` e vedo che non solo restituisce un comando, ma accetta anche due argomenti. Devo tenerne conto nello Statement:

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  var command = factory.CreateNewReservationCommand(
    Any.Instance<ReservationRequestDto>(),
    Any.Instance<ReservationInProgress>());

  //THEN
  Assert.True(false);
 }
}
```

Ho utilizzato `Any.Instance<>()` per generare istanze anonime sia di DTO che di `ReservationInProgress`.

Johnny: Questo è esattamente quello che farei. Li assegnerei alle variabili solo se avessi bisogno di usarli da qualche altra parte nello Statement.

Benjamin: Lo Statement viene compilato. Ho ancora l'asserzione da scrivere. Cosa dovrei affermare esattamente? Il comando che ricevo dalla factory ha solo un metodo `void Execute()`. Tutto il resto è nascosto.

Johnny: In genere, non c'è molto che possiamo specificare sugli oggetti creati dalle factory. In questo caso possiamo solo indicare il tipo atteso del comando.

Benjamin: Aspetta, non c'è già questo nella firma della factory? Osservando il metodo di creazione, posso già vedere che il tipo restituito è `ReservationCommand`... aspetta!

Johnny: Pensavo l'avresti notato. `ReservationCommand` è un'interfaccia. Ma un oggetto deve avere un tipo concreto e questo tipo è ciò che dobbiamo specificare in questo Statement. Non abbiamo ancora questo tipo. Siamo sul punto di scoprirlo e, quando lo faremo, alcuni nuovi elementi verranno aggiunti alla nostra lista dei TODO.

Benjamin: Chiamerò la classe `NewReservationCommand` e modificherò il mio Statement per asserire il tipo. Inoltre, penso di sapere ora come chiamare questo Statement:

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateNewReservationCommandWhenRequested()
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  var command = factory.CreateNewReservationCommand(
    Any.Instance<ReservationRequestDto>(),
    Any.Instance<ReservationInProgress>());

  //THEN
  Assert.IsType<NewReservationCommand>(command);
 }
}
```

Johnny: Ora vediamo -- la dichiarazione sembra completa. L'hai fatto quasi senza il mio aiuto.

Benjamin: Grazie. Il codice, però, non viene compilato. Il tipo `NewReservationCommand` non esiste.

Johnny: Allora introduciamolo:

```
public class NewReservationCommand
{

}
```

Benjamin: Non dovrebbe implementare l'interfaccia `ReservationCommand`?

Johnny: Non abbiamo bisogno di farlo ancora. Il compilatore si è lamentato solo del fatto che il tipo non esiste.

Benjamin: Il codice viene compilato ora, ma lo Statement è falso perché il metodo `CreateNewReservationCommand` lancia una `NotImplementedException`:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   throw new NotImplementedException();
  }
```

Johnny: Ricordi cosa ti ho detto l'ultima volta?

Benjamin: Sì, lo Statement deve essere falso per la ragione giusta.

Johnny: Esattamente. `NotImplementedException` non è il motivo giusto. Cambierò il codice sopra per restituire null:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   return null;
  }
```

Ora lo Statement è falso perché questa asserzione fallisce:

```
Assert.IsType<NewReservationCommand>(command);
```

Benjamin: si lamenta che il comando è null.

Johnny: Quindi è arrivato il momento giusto per inserire l'implementazione corretta:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   return new NewReservationCommand();
  }
```

Benjamin: Ma questo non viene compilato -- `NewReservationCommand` non implementa l'interfaccia `ReservationCommand`, te l'ho detto prima.

Johnny: e il compilatore ci obbliga a implementare questa interfaccia:

```
public class NewReservationCommand : ReservationCommand
{

}
```

Il codice continua a non essere compilabile, perché l'interfaccia ha un metodo `Execute()` che dobbiamo implementare in `NewReservationCommand`. Qualsiasi implementazione andrà bene poiché la logica di questo metodo non rientra nello "scope" dello Statement corrente. Dobbiamo solo accontentare il compilatore. Il nostro IDE può generare per noi il corpo del metodo di default. Questo dovrebbe fare:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

La `NotImplementedException` verrà aggiunta alla nostra lista TODO e specificheremo il suo comportamento con un'altro Statement in seguito.

Benjamin: Il nostro Statement corrente sembra essere "true" adesso. Qualcosa mi dà fastidio, però. Che dire degli argomenti che stiamo passando al metodo `CreateNewReservationCommand`? Ce ne sono due:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
```

(continues on next page)

(continua dalla pagina precedente)

```
   return new NewReservationCommand();
  }
```

e sono entrambi inutilizzati. Come mai?

Johnny: Avremmo potuto usarli e passarli al comando. A volte lo faccio anche se lo Statement non dipende da questo, quindi non c'è pressione. Ho scelto di non farlo questa volta per mostrarti che dovremo comunque passarli quando specifichiamo i comportamenti del nostro comando.

Benjamin: diciamo che hai ragione e quando specifichiamo il comportamento del comando, dovremo modificare la factory per passare questi due, ad es. come questo:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
   return new NewReservationCommand(requestDto, reservationInProgress);
  }
```

ma nessuno Statement dice quali valori dovrebbero essere usati, quindi potrò anche imbrogliare scrivendo qualcosa come:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
   return new NewReservationCommand(null, null);
  }
```

e tutti gli Statement saranno comunque "true".

Johnny: Esatto. Sebbene esistano alcune tecniche per specificarlo a livello di unità, preferirei fare affidamento su Statement di livello superiore per garantire il passaggio dei valori corretti. Ti mostrerò come farlo un'altra volta.

Benjamin: Grazie, sembra che dovrò solo aspettare.

Johnny: Comunque sembra che abbiamo finito con questo Statement, quindi facciamo una breve pausa.

Benjamin: Certo!

CAPITOLO 47

Creazione di oggetti "test-driving" -- una retrospettiva

Nell'ultimo capitolo, Johnny e Benjamin hanno specificato il comportamento di una factory, scrivendo uno Statement della Specifica sulla creazione di oggetti. Si spera che questo capitolo risponda ad alcune domande che potrebbero sorgere su cosa hanno fatto e cosa si dovrebbe fare in situazioni simili.

47.1 Limiti della specifica della creazione

La creazione di oggetti potrebbe significare cose diverse, ma nello stile del design che sto descrivendo, si tratta principalmente di creare astrazioni. Queste astrazioni di solito incapsulano molto, esponendo solo interfacce ristrette. Un esempio potrebbe essere il `ReservationCommand` del nostro sistema di prenotazione dei treni -- contiene un singolo metodo chiamato `Execute()`, che non restituisce nulla. L'unica cosa che Johnny e Benjamin potevano specificare riguardo al comando creato nello Statement era il suo tipo concreto. Non potevano imporre che gli argomenti del metodo factory venissero passati all'istanza creata, quindi la loro implementazione lo ha semplicemente tralasciato. Tuttavia, a un certo punto avrebbero dovuto fornire gli argomenti corretti, altrimenti lintera logica non avrebbe funzionato.

Per far sì che lo Statement sia falso quando non vengono passati i parametri corretti, potrebbero provare a utilizzare tecniche come la "reflection" per ispezionare il grafo dell'oggetto creato e verificare se contiene gli oggetti giusti, ma ciò trasformerebbe presto la specifica in una fragile replica del codice di produzione. La composizione dell'oggetto è una definizione di comportamento per lo più dichiarativa, più o meno come l'HTML è una descrizione dichiarativa di una GUI. Dovremmo specificare il comportamento piuttosto che la struttura di ciò che rende possibile il comportamento.

Quindi come facciamo a sapere se il comando è stato creato esattamente come lo intendevamo? Possiamo specificarlo del tutto?

Ho detto che la composizione dell'oggetto ha lo scopo di fornire una sorta di comportamento a livello superiore -- risultante dal modo in cui gli oggetti sono composti. Gli Statement di livello superiore possono descrivere tale comportamento. Ad esempio, quando specifichiamo il comportamento dell'intero componente, questi Statement ci obbligheranno indirettamente a ottenere la giusta composizione dell'oggetto -- sia nelle nostre factory che a livello di "composition root".

Per ora, lascerò perdere le cose e tratterò le Specifiche di livello superiore nelle parti successive del libro.

47.2 Perché specificare la creazione dell'oggetto?

Quindi, se ci affidiamo alla Specifica di livello superiore per imporre la corretta composizione dell'oggetto, perché scrivere una Specifica a livello di unità per la creazione dell'oggetto? Il motivo principale è: il progresso. Ripensando allo Statement scritto da Johnny e Benjamin -- li ha costretti a creare una classe che implementa l'interfaccia `ReservationCommand`. Questa classe doveva poi implementare il metodo dell'interfaccia in un modo che soddisfacesse il compilatore. Giusto per ricordare, è andata a finire così:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

In questo modo, hanno ottenuto una nuova classe "test-drive" e la `NotImplementedException` che appariva nel metodo `Execute` generato, è finita nella loro lista TODO. Come passo successivo, potrebbero scegliere questo elemento TODO e iniziare a lavorarci sopra.

Come si può vedere, uno Statement che specificava il comportamento della factory, sebbene non perfetto come prova, ha permesso di continuare il flusso del TDD.

47.3 Cosa specifichiamo negli Statement creazionali?

Johnny e Benjamin specificarono quale doveva essere il tipo dell'oggetto creato. Indirettamente, hanno anche specificato che l'oggetto creato non dovrebbe essere nullo. C'è qualcos'altro che potremmo voler includere in una specifica creazionale?

47.3.1 La scelta

A volte sono le factory a decidere quale tipo di oggetto restituire. In questi casi, potremmo voler specificare le diverse scelte che la factory può fare. Ad esempio, se una factory è simile a questa:

```
public class SomeKindOfFactory
{
 public Command CreateFrom(string commandName)
 {
  if(commandName == "command1")
  {
   return new Command1();
  }
  if(commandName == "command2")
  {
   return new Command2();
  }
  else
  {
   throw new InvalidCommandException(commandName);
  }
 }
}
```

allora avremmo bisogno di tre Statement per specificarne il comportamento:

1. Per lo scenario in cui viene restituita un'istanza di `Command1`.
2. Per lo scenario in cui viene restituita un'istanza di `Command2`.
3. Per lo scenario in cui viene generata un'eccezione.

47.3.2 Contenuto della collection

Allo stesso modo, se la nostra factory deve restituire una collection - dobbiamo dichiarare cosa ci aspettiamo che contenga. Si consideri la seguente dichiarazione su una factory che deve creare una sequenza di fasi di elaborazioni:

```
[Fact] public void
ShouldCreateProcessingStepsInTheRightOrder()
```

(continues on next page)

(continua dalla pagina precedente)

```
{
 //GIVEN
 var factory = new ProcessingStepsFactory();

 //WHEN
 var steps = factory.CreateStepsForNewItemRequest();

 //THEN
 Assert.Equal(3, steps.Count);
 Assert.IsType<ValidationStep>(steps[0]);
 Assert.IsType<ExecutionStep>(steps[1]);
 Assert.IsType<ReportingStep>(steps[2]);
}
```

Dice che la collection creata dovrebbe contenere tre elementi in un ordine specifico e indica i tipi previsti di tali elementi.

Ultimo ma non meno importante, a volte specifichiamo la creazione di *oggetti valore*.

47.4 Creazione di *oggetti di valore*

Gli *oggetti valore* vengono generalmente creati utilizzando metodi factory. A seconda che il costruttore di tale *oggetto valore* sia pubblico o meno, possiamo accogliere questi metodi factory in modo diverso nelle nostre Specifiche.

47.4.1 *Oggetto valore* con un costruttore pubblico

Quando è disponibile un costruttore pubblico, possiamo dichiarare l'uguaglianza attesa di un oggetto creato utilizzando un metodo factory con un altro oggetto creato utilizzando il costruttore:

```
[Fact] public void
ShouldBeEqualToIdWithGroupType()
{
 //GIVEN
 var groupName = Any.String();

 //WHEN
 var id = EntityId.ForGroup(groupName);

 //THEN
 Assert.Equal(new EntityId(groupName, EntityTypes.Group), id);
}
```

47.4.2 *Oggetto valore* con un costruttore non pubblico

Quando i metodi factory sono l'unico mezzo per creare *oggetti valore*, utilizziamo semplicemente questi metodi nelle nostre dichiarazioni e indichiamo i dati attesi o i comportamenti attesi. Ad esempio, il seguente Statement afferma che un path assoluto di directory dovrebbe diventare un path assoluto di file dopo aver aggiunto un nome di file:

```
[Fact]
public void ShouldBecomeAnAbsoluteFilePathAfterAppendingFileName()
{
 //GIVEN
 var dirPath = AbsoluteDirectoryPath.Value("C:\\");
 var fileName = FileName.Value("file.txt");

 //WHEN
 AbsoluteFilePath absoluteFilePath = dirPath.Append(fileName);
```

(continues on next page)

(continua dalla pagina precedente)

```
 //THEN
 Assert.Equal(
     AbsoluteDirectoryPath.Value("C:\\file.txt"),
     absoluteFilePath);
}
```

Notare che lo Statement crea i valori utilizzando i rispettivi metodi factory. Ciò è inevitabile poiché questi metodi sono l'unico modo in cui è possibile creare gli oggetti.

47.4.3 Validazione

I metodi factory per gli *oggetti valore* possono avere alcuni comportamenti aggiuntivi come la validazione -- dobbiamo specificarlo anche noi. Ad esempio, lo Statement seguente dice che dovrebbe essere generata un'eccezione quando provo a creare un path assoluto di file da una stringa nulla:

```
[Fact]
public void ShouldNotAllowToBeCreatedWithNullValue()
{
 Assert.Throws<ArgumentNullException>(() => AbsoluteFilePath.Value(null));
}
```

47.5 Riepilogo

È stato un viaggio veloce attraverso la scrittura di Statement creazionali. Sebbene non siano approfonditi come test, creano la necessità di nuove astrazioni, permettendoci di continuare il processo del TDD. Quando iniziamo il nostro processo TDD da Statement di livello superiore, possiamo rimandare la specifica delle factory e talvolta addirittura evitarlo del tutto.

CAPITOLO 48

Logica dell'applicazione test-driving

Johnny: Qual è il prossimo elemento della nostra lista dei TODO?

Benjamin: Abbiamo...

- un unico promemoria per modificare il nome della factory che crea le istanze `ReservationInProgress`.
- Inoltre, abbiamo due punti in cui una `NotImplementedException` suggerisce che c'è ancora qualcosa da implementare.
  - La prima è la factory menzionata.
  - La seconda è la classe `NewReservationCommand` che abbiamo scoperto scrivendo uno Statement per la factory.

Johnny: Facciamo la `NewReservationCommand`. Sospetto che la complessità che abbiamo inserito nel controller ci ripagherà e saremo in grado di testare la logica dei comandi alle nostre condizioni.

Benjamin: Non vedo l'ora di vederlo. Ecco il codice attuale del comando:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

Johnny: Quindi abbiamo la firma della classe e del metodo. Sembra che possiamo usare la consueta strategia di scrivere un nuovo Statement.

Benjamin: Intendi "inizia invocando un metodo se ne hai uno"? Ho pensato la stessa cosa. Fammi provare.

```
public class NewReservationCommandSpecification
{
 [Fact] public void
 ShouldXXXXXXXXXXX() //TODO change the name
 {
  //GIVEN
  var command = new NewReservationCommand();
```

(continues on next page)

(continua dalla pagina precedente)

```
  //WHEN
  command.Execute();

  //THEN
  Assert.True(false); //TODO unfinished
 }
}
```

Questo è quello che potrei scrivere quasi senza pensarci. Ho semplicemente preso le parti in nostro possesso e le ho inserite nello Statement.

Manca ancora l'asserzione - quella che abbiamo è semplicemente un segnaposto. Questo è il momento giusto per pensare a cosa dovrebbe accadere quando eseguiamo il comando.

Johnny: Per ottenere il comportamento previsto, dobbiamo guardare indietro ai dati di input per l'intero caso d'uso. L'abbiamo passato alla factory e ce ne siamo dimenticati. Quindi, solo come promemoria - ecco come appare:

```
public class ReservationRequestDto
{
  public readonly string TrainId;
  public readonly uint SeatCount;

  public ReservationRequestDto(string trainId, uint seatCount)
  {
    TrainId = trainId;
    SeatCount = seatCount;
  }
}
```

La prima parte è l'ID del treno -- dice su quale treno dovremmo prenotare i posti. Quindi dobbiamo in qualche modo scegliere un treno dalla flotta per la prenotazione. Quindi, su quel treno, dobbiamo prenotare tutti i posti richiesti dal cliente. Il numero di posti richiesto è la seconda parte della richiesta dell'utente.

Benjamin: Non aggiorneremo i dati in una sorta di memoria persistente? Dubito che la compagnia ferroviaria vorrebbe che la prenotazione scomparisse al riavvio dell'applicazione.

Johnny: Sì, dobbiamo comportarci come se ci fosse una sorta di persistenza.

Considerato tutto quanto sopra, posso vedere due nuovi ruoli nel nostro scenario:

1. Una flotta - da cui prendiamo il treno e dove salviamo le nostre modifiche
2. Un treno - che gestirà la logica della prenotazione.

Entrambi questi ruoli devono essere modellati come mock, perché mi aspetto che svolgano un ruolo attivo in questo scenario.

Ampliamo il nostro Statement con le nostre scoperte.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var command = new NewReservationCommand(fleet);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
```

(continues on next page)

(continua dalla pagina precedente)

```
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount);
  fleet.UpdateInformationAbout(train);
 };
}
```

Benjamin: Vedo che mancano molte cose qui. Ad esempio, non abbiamo le variabili `train` e `fleet`.

Johnny: Abbiamo creato una necessità per loro. Penso che ora possiamo tranquillamente introdurli nello Statement.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var command = new NewReservationCommand(fleet);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount);
  fleet.UpdateInformationAbout(train);
 };
}
```

Benjamin: Vedo due nuovi tipi: `TrainFleet` e `ReservableTrain`.

Johnny: Simboleggiano i ruoli che abbiamo appena scoperto.

Benjamin: Inoltre, ho notato che hai utilizzato `Received.InOrder()` da NSubstitute per specificare l'ordine di chiamata previsto.

Johnny: Questo perché dobbiamo prenotare i posti prima di aggiornare le informazioni in una sorta di archivio. Se sbagliassimo l'ordine, il resto potrebbe andare perso.

Benjamin: Ma in questo Statement manca qualcosa. Ho appena esaminato gli output che i nostri utenti si aspettano:

```
public class ReservationDto
{
  public readonly string TrainId;
  public readonly string ReservationId;
  public readonly List<TicketDto> PerSeatTickets;

  public ReservationDto(
    string trainId,
    List<TicketDto> perSeatTickets,
    string reservationId)
  {
    TrainId = trainId;
    PerSeatTickets = perSeatTickets;
    ReservationId = reservationId;
```

(continues on next page)

(continua dalla pagina precedente)

```
  }
}
```

Sono molte le informazioni che dobbiamo restituire all'utente. Come lo faremo esattamente quando la chiamata a `train.ReserveSeats(seatCount)` che hai inventato non dovrebbe restituire nulla?

Johnny: Ah, sì, quasi dimenticavo - abbiamo l'istanza `ReservationInProgress` che abbiamo passato alla factory, ma non ancora al comando, giusto? La `ReservationInProgress` è stato inventata proprio per questo scopo - raccogliere le informazioni necessarie per produrre un risultato dell'intera operazione. Vorrei solo aggiornare rapidamente lo Statement:

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var reservationInProgress = Any.Instance<ReservationInProgress>();
 var command = new NewReservationCommand(fleet, reservationInProgress);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount, reservationInProgress);
  fleet.UpdateInformationAbout(train);
 };
}
```

Ora il metodo `ReserveSeats` accetta `reservationInProgress`.

Benjamin: Perché stai passando il metodo `reservationInProgress` al metodo `ReserveSeats`?

Johnny: Il comando non contiene le informazioni necessarie per compilare `reservationInProgress` una volta che la prenotazione ha avuto esito positivo. Dobbiamo rimandarlo alle implementazioni di `ReservableTrain` per decidere ulteriormente il posto migliore per farlo.

Benjamin: Capisco. Osservando di nuovo lo Statement, mancano altre due variabili -- `trainId` e `seatCount` -- e non solo le loro definizioni, ma non le passiamo affatto al comando. Sono presenti solo nelle nostre assunzioni ed "expectation".

Johnny: Esatto, lasciami correggere.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var trainId = Any.String();
 var seatCount = Any.UnsignedInt();
 var reservationInProgress = Any.Instance<ReservationInProgress>();
 var command = new NewReservationCommand(
   trainId,
   seatCount,
   fleet,
   reservationInProgress);
```

(continues on next page)

(continua dalla pagina precedente)

```
 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount, reservationInProgress);
  fleet.UpdateInformationAbout(train);
 };
}
```

Benjamin: Perché `seatCount` è un `uint`?

Johnny: Cercalo nel DTO: lì c'è un `uint`. Non vedo la necessità di ridefinirlo qui.

Benjamin: Bene, ma per quanto riguarda `trainId` - è una `string`. Non mi hai detto che dobbiamo utilizzare *oggetti valore* relativi al dominio per concetti come questo?

Johnny: Sì, effettueremo il refactoring di questa `string` in un *oggetto valore*, soprattutto perché abbiamo il requisito che i confronti degli ID dei treni non facciano distinzione tra maiuscole e minuscole. Ma prima voglio finire questa istruzione prima di definire e specificare un nuovo tipo. Tuttavia, è meglio lasciare una nota TODO per tornarci più tardi:

```
 var trainId = Any.String(); //TODO extract value object
```

Fin qui tutto bene, penso che abbiamo una dichiarazione completa. Vuoi prendere la tastiera?

Benjamin: Grazie. Iniziamo allora a implementarlo. Innanzitutto, inizierò con queste due interfacce:

```
var fleet = Substitute.For<TrainFleet>();
var train = Substitute.For<ReservableTrain>();
```

Non esistono, quindi questo codice non viene compilato. Posso facilmente risolvere questo problema creando le interfacce nel codice di produzione:

```
public interface TrainFleet
{
}

public interface ReservableTrain
{
}
```

Ora per questa parte:

```
var command = new NewReservationCommand(
   trainId,
   seatCount,
   fleet,
   reservationInProgress);
```

Non viene compilato perché il comando non accetta ancora alcun parametro del costruttore. Creiamo quindi un costruttore di adattamento:

```
public class NewReservationCommand : ReservationCommand
{
 public NewReservationCommand(
```

(continues on next page)

(continua dalla pagina precedente)

```
  string trainId,
  uint seatCount,
  TrainFleet fleet,
  ReservationInProgress reservationInProgress)
 {

 }

 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

Il nostro Statement ora può invocare questo costruttore, ma abbiamo guastato `TicketOfficeCommandFactory` che crea anche un'istanza `NewReservationCommand`. Guarda:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
 {
  //Stopped compiling:
  return new NewReservationCommand();
 }
```

Johnny: Dobbiamo correggere la factory nello stesso modo in cui abbiamo dovuto correggere la "composition root" durante il test-driving del controller. Vediamo... Ecco:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
 {
  return new NewReservationCommand(
   requestDto.TrainId,
   requestDto.SeatCount,
   new TodoTrainFleet(), // TODO fix name and scope
   reservatonInProgress
   );
 }
```

Benjamin: Il passaggio dei parametri mi sembra semplice, tranne `TodoTrainFleet()` -- so già che il nome è un segnaposto - hai già fatto qualcosa del genere prima. Ma per quanto riguarda il ciclo di vita dello "scope"?

Johnny: È anch'esso un segnaposto. Per ora, voglio soddisfare il compilatore, mantenendo allo stesso tempo veri gli Statement esistenti e introducendo una nuova classe -- `TodoTrainFleet` -- che porterà nuovi elementi nella nostra lista TODO.

Benjamin: Nuovi articoli TODO?

Johnny: Sì. Look -- il tipo `TodoTrainFleet` non esiste ancora. Lo creerò ora:

```
public class TodoTrainFleet
{

}
```

Questo non corrisponde alla firma del costruttore del comando, che prevede un `TrainFleet`, quindi devo fare in modo che `TodoTrainFleet` implementi questa interfaccia:

```
public class TodoTrainFleet : TrainFleet
{

}
```

Ora sono costretto a implementare i metodi dall'interfaccia `TrainFleet`. Anche se questa interfaccia non definisce ancora alcun metodo, ne abbiamo già scoperti due nel nostro Statement, quindi bisognerà presto ottenerli per accontentare il compilatore. Entrambi conterranno il codice che lancia `NotImplementedException`, che verrà inserito nell'elenco TODO.

Benjamin: Capisco. Ad ogni modo, la factory compila di nuovo. Ci resta ancora questa parte dello Statement:

```
 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount, reservationInProgress);
  fleet.UpdateInformationAbout(train);
 };
```

Johnny: Si tratta solo di introdurre tre metodi. Puoi gestirlo.

Benjamin: Grazie. La prima riga è `fleet.Pick(trainId).Returns(train)`. Genererò semplicemente il metodo `Pick` utilizzando il mio IDE:

```
public interface TrainFleet
{
 ReservableTrain Pick(string trainId);
}
```

L'interfaccia `TrainFleet` è implementata dal `TodoTrainFleet` di cui abbiamo parlato qualche minuto fa. È necessario implementare anche il metodo `Pick` altrimenti non verrà compilato:

```
public class TodoTrainFleet : TrainFleet
{
 public ReservableTrain Pick(string trainId)
 {
   throw new NotImplementedException();
 }
}
```

Questa `NotImplementedException` verrà inserita nel nostro elenco TODO proprio come hai menzionato. Carino!

Poi arriva la riga successiva dallo Statement: `train.ReserveSeats(seatCount, reservationInProgress)` e ne genererò una firma del metodo come dalla riga precedente.

```
public interface ReservableTrain
{
 void ReserveSeats(uint seatCount, ReservationInProgress reservationInProgress);
}
```

L'interfaccia `ReservableTrain` non ha alcuna implementazione finora, quindi non c'è più nulla a che fare con questo metodo.

L'ultima riga: `fleet.UpdateInformationAbout(train)` che deve essere aggiunta all'interfaccia `TrainFleet`:

```
public interface TrainFleet
{
 ReservableTrain Pick(string trainId);
 void UpdateInformationAbout(ReservableTrain train);
}
```

Inoltre, dobbiamo definire questo metodo nella classe `TodoTrainFleet`:

```
public class TodoTrainFleet : TrainFleet
{
 public ReservableTrain Pick(string trainId)
 {
   throw new NotImplementedException();
 }

 void UpdateInformationAbout(ReservableTrain train)
 {
   throw new NotImplementedException();
 }
}
```

Johnny: Anche questa `NotImplementedException` verrà aggiunta all'elenco delle cose da fare, quindi potremo rivisitarla in seguito. Sembra che lo Statement venga compilato e, come previsto, sia falso, ma non per la ragione giusta.

Benjamin: Vediamo... sì, viene lanciata una `NotImplementedException` dal metodo `Execute()` del comando.

Johnny: Liberiamocene.

Benjamin: Certo. Ho rimosso `throw` e ora il metodo è vuoto:

```
 public void Execute()
 {

 }
```

Lo Statement ora è falso perché le chiamate previste non corrispondono.

Johnny: Ciò significa che siamo finalmente pronti a codificare alcuni comportamenti nella classe `NewReservationCommand`. Innanzitutto, assegniamo tutti i parametri del costruttore ai campi -- ne avremo bisogno.

Benjamin: Ecco:

```
public class NewReservationCommand : ReservationCommand
{
 private readonly string _trainId;
 private readonly uint _seatCount;
 private readonly TraingFleet _fleet;
 private readonly ReservationInProgress _reservationInProgress;

 public NewReservationCommand(
  string trainId,
  uint seatCount,
  TrainFleet fleet,
  ReservationInProgress reservationInProgress)
 {
  _trainId = trainId;
  _seatCount = seatCount;
  _fleet = fleet;
  _reservationInProgress = reservationInProgress;
```

(continues on next page)

(continua dalla pagina precedente)

```
 }

 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

Johnny: Ora aggiungiamo le chiamate previste nello Statement, ma nell'ordine opposto.

Benjamin: Per assicurarsi che l'ordine sia asserito correttamente nello Statement?

Johnny: Esattamente.

Benjamin: Ok.

```
public void Execute()
{
 var train = _fleet.Pick(_trainId);
 _fleet.UpdateInformationAbout(train);
 train.ReserveSeats(seatCount);
}
```

Lo Statement è ancora falso, questa volta a causa dell'ordine di chiamata sbagliato. Ora che abbiamo la conferma che dobbiamo effettuare le chiamate nell'ordine giusto, sospetto che tu voglia che lo inverta, quindi...

```
public void Execute()
{
 var train = _fleet.Pick(_trainId);
 train.ReserveSeats(seatCount, reservationInProgress);
  _fleet.UpdateInformationAbout(train);
}
```

Johnny: Esattamente. Lo Statement ora è "true". Congratulazioni!

Benjamin: Ora che guardo questo codice, non è protetto da alcun tipo di eccezione che potrebbe essere generata da `_fleet` o da `train`.

Johnny: Aggiungilo alla lista TODO - prima o poi dovremo occuparcene. Per ora, prendiamoci una pausa.

48.1 Riepilogo

In questo capitolo, Johnny e Benjamin hanno utilizzato nuovamente il rilevamento (discovery) dell'interfaccia. Hanno utilizzato alcune ragioni tecniche e altre legate al dominio per creare la necessità di nuove astrazioni e progettare i loro protocolli di comunicazione. Queste astrazioni sono state poi inserite nello Statement.

Ricordrsi che Johnny e Benjamin hanno prolungato lo sforzo durante il test-driving del controller. Questo lavoro ora ha dato i suoi frutti - erano liberi di modellare astrazioni per lo più al di fuori dei vincoli imposti da un framework specifico.

Questo capitolo non ha una retrospettiva complementare come i precedenti. La maggior parte delle cose interessanti che sono accadute qui sono già state spiegate.

CAPITOLO 49

Oggetti valore nel test-driving

In questo capitolo andiamo più avanti nel tempo. Johnny e Benjamin hanno appena estratto un tipo di *oggetto valore* per un ID treno e hanno iniziato il lavoro per specificarlo ulteriormente.

49.1 *Oggetto valore* iniziale

Johnny: Oh, sei tornato. Il refactoring è terminato -- abbiamo estratto un tipo di *oggetto di valore* interessante dal codice corrente. Ecco il codice sorgente della classe `TrainId`:

```
public class TrainId
{
 private readonly string _value;

 public static TrainId From(string trainIdAsString)
 {
  return new TrainId(trainIdAsString);
 }

 private TrainId(string value)
 {
  _value = value;
 }

 public override bool Equals(object obj)
 {
  return _value == ((TrainId) obj)._value;
 }

 public override string ToString()
 {
  return _value;
 }
}
```

Benjamin: Aspetta, ma non abbiamo ancora alcuna specifica per questa classe. Da dove viene tutta questa implementazione?

Johnny: Questo perché mentre stavi bevendo il tuo tè, ho estratto questo tipo da un'implementazione esistente che era già una risposta Statement falsi.

Benjamin: Capisco. Quindi non abbiamo la mock della classe `TrainId` class in altri Statement, giusto?

Johnny: No. Questa è una regola generale -- non creiamo mock di *oggetti di valore*. This is a general rule -- we don't mock value objects. Non rappresentano comportamenti astratti e polimorfici. Per gli stessi motivi per cui non creiamo interfacce e mock per `string` e `int`, non lo facciamo per `TrainId`.

49.2 Semantica del valore

Benjamin: Quindi, data l'implementazione esistente, c'è ancora qualcosa da scrivere?

Johnny: Sì. Ho deciso che questo `TrainId` dovrebbe essere un *oggetto valore* e i miei principi di progettazione per gli *oggetti valore* richiedono che fornisca alcune garanzie in più rispetto a quelle risultanti da un semplice refactoring. Inoltre, non si deve dimenticare che il confronto degli ID dei treni non deve fare distinzione tra maiuscole e minuscole. Questo è qualcosa che non abbiamo specificato da nessuna parte.

Benjamin: Hai parlato di "maggiori garanzie". Intendi l'uguaglianza?

Johnny: Sì, C# come linguaggio prevede che l'uguaglianza segua determinate regole. Voglio che questa classe li implementi. Inoltre, voglio che implementi `GetHashCode` correttamente e mi assicuro che le istanze di questa classe siano immutabili. Ultimo ma non meno importante, vorrei che il metodo factory `From` lanciasse un'eccezione quando viene passata una stringa nulla o vuota. Nessuno di questi input dovrebbe portare alla creazione di un ID valido.

Benjamin: Sembra che ci siano molte dichiarazioni da scrivere.

Johnny: Non proprio. Aggiungi la validazione e il confronto in minuscolo alla lista dei TODO. Nel frattempo -- guarda -- ho scaricato una libreria che mi aiuterà con la parte relativa all'immutabilità e all'uguaglianza. In questo modo, tutto ciò che devo scrivere è:

```
[Fact] public void
ShouldHaveValueObjectSemantics()
{
 var trainIdString = Any.String();
 var otherTrainIdString = Any.OtherThan(trainIdString);
 Assert.HasValueSemantics<TrainId>(
   new Func<TrainId>[]
   {
    () => TrainId.From(trainIdString)
   },
   new Func<TrainId>[]
   {
    () => TrainId.From(otherTrainIdString);
   },
 );
}
```

Questa asserzione accetta due array:

- il primo array contiene funzioni di factory che creano oggetti che dovrebbero essere uguali tra loro. Per ora, abbiamo solo un esempio, perché non ho toccato la questione delle lettere minuscole e maiuscole. Ma quando lo faccio, l'array conterrà più voci per sottolineare che gli ID creati dalla stessa stringa con lettere maiuscole e minuscole diverse dovrebbero essere considerati uguali.
- il secondo array contiene funzioni factory che creano oggetti di esempio che dovrebbero essere considerati non uguali a nessuno degli oggetti generati dalle funzioni factory "uguali". C'è anche un singolo esempio qui poiché il metodo `From` di `TrainId` ha un singolo argomento, quindi l'unico modo in cui un'istanza può differire da un'altra è creata con un valore diverso di questo argomento.

Dopo aver valutato questo Statement, ottengo il seguente output:

```
- TrainId must be sealed, or derivatives will be able to override GetHashCode() with␣
↪mutable code.

- a.GetHashCode() and b.GetHashCode() should return same values for equal objects.

- a.Equals(null) should return false, but instead threw System.NullReferenceException:␣
↪Object reference not set to an instance of an object.

- '==' and '!=' operators are not implemented
```

Benjamin: Molto furbo. Quindi queste sono le regole che il nostro `TrainId` non segue ancora. Le implementerete una per una?

Johnny: Eheh, no. L'implementazione sarebbe così noiosa che utilizzerei il mio IDE per generare l'implementazione necessaria o, ancora una volta, utilizzare una libreria. Ultimamente preferisco quest'ultima. Lasciami quindi scaricare una libreria chiamata Value e usarla sul nostro `TrainId`.

Prima di tutto, `TrainId` deve ereditare dalla classe generica `ValueType` come questa:

```
public class TrainId : ValueType<TrainId>
```

Questa eredità ci impone di implementare il seguente metodo "speciale":

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 throw new NotImplementedException();
}
```

Benjamin: Strano, cosa fa?

Johnny: È così che la libreria automatizza l'implementazione dell'uguaglianza. Dobbiamo solo restituire un array di valori che vogliamo confrontare tra due istanze. Poiché la nostra uguaglianza si basa esclusivamente sul campo `_value`, dobbiamo restituire proprio questo:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value;
}
```

Dobbiamo anche rimuovere il metodo Equals() esistente.

Benjamin: Ottimo, ora l'unica ragione per cui l'asserzione fallisce è:

```
- TrainId must be sealed, or derivatives will be able to override GetHashCode() with␣
↪mutable code.
```

Sono colpito dal fatto che questa libreria Value si sia occupata dei metodi di uguaglianza, degli operatori di uguaglianza e di `GetHashCode()`.

Johnny: Bello, eh? Ok, terminiamo questa parte e aggiungiamo la parola chiave `sealed`. Il codice sorgente completo della classe è simile al seguente:

```
public sealed class TrainId : ValueType<TrainId>
{
 private readonly string _value;

 public static TrainId From(string trainIdAsString)
 {
  return new TrainId(trainIdAsString);
 }
```

(continues on next page)

(continua dalla pagina precedente)

```
 private TrainId(string value)
 {
  _value = value;
 }

 protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
 {
  yield return _value;
 }

 public override string ToString()
 {
  return _value;
 }
}
```

Benjamin: E lo Statement è "true".

49.3 Confronto "case-insensitive"

Johnny: Cosa resta nella nostra lista dei TODO?

Benjamin: Due elementi:

- Il confronto degli ID dei treni non deve fare distinzione tra maiuscole e minuscole.
- `null` e stringhe vuote non dovrebbero essere consentite come ID treno validi.

Johnny: Facciamo il confronto case-insensitive , dovrebbe essere relativamente semplice.

Benjamin: Ok, hai detto che avremmo bisogno di espandere lo Statement che hai scritto, aggiungendovi un'altra "equal value factory". Fammi provare. Cosa ne pensi di questo?

```
[Fact] public void
ShouldHaveValueSemantics()
{
 var trainIdString = Any.String();
 var otherTrainIdString = Any.OtherThan(trainIdString);
 Assert.HasValueSemantics<TrainId>(
   new Func<TrainId>[]
   {
    () => TrainId.From(trainIdString.ToUpper())
    () => TrainId.From(trainIdString.ToLower())
   },
   new Func<TrainId>[]
   {
    () => TrainId.From(otherTrainIdString);
   },
 );
}
```

Che ne dici di quello? Da quello che mi hai spiegato, capisco che aggiungendo una seconda factory al primo array, dico che entrambe le istanze dovrebbero essere trattate come uguali - quella con la stringa minuscola e quella con la stringa maiuscola.

Johnny: Esattamente. Ora, rendiamo "true" lo Statement. Fortunatamente, possiamo farlo modificando il metodo `GetAllAttributesToBeUsedForEquality` da:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value;
}
```

in:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value.ToLower();
}
```

Eeee!!! fatto! L'asserzione ha controllato `Equals`, operatori di uguaglianza e `GetHashCode()` e tutto sembra funzionare. Possiamo passare all'elemento successivo nel nostro elenco di TODO.

49.4 Validazione dell'Input

Johnny: Prendiamoci cura del metodo `From` - dovrebbe non consentire l'input `null` -- ci aspettiamo un'eccezione quando passiamo un `null` all'interno.

Per ora, il metodo è il seguente:

```
public static TrainId From(string trainIdAsString)
{
 return new TrainId(trainIdAsString);
}
```

Benjamin: OK, lasciami scrivere uno Statement sul comportamento previsto:

```
[Fact] public void
ShouldThrowWhenCreatedWithANullInput()
{
  Assert.Throws<ArgumentNullException>(() => TrainId.From(null));
}
```

È stato facile, eh?

Johnny: Grazie. Lo Statement è attualmente falso perché prevede un'eccezione ma non viene generato nulla. Rendiamolo "true" implementando il controllo di `null`.

```
public static TrainId From(string trainIdAsString)
{
 if(trainIdAsString == null)
 {
  throw new ArgumentNullException(nameof(trainIdAsString));
 }
 return new TrainId(trainIdAsString);
}
```

Benjamin: Fantastico, ha funzionato!

49.5 Riepilogo

Johnny e Benjamin hanno ancora un comportamento da specificare -- lanciare un'eccezione quando una stringa vuota viene passata al metodo factory -- ma seguirli ulteriormente probabilmente non ci porterà alcuna nuova intuizione. Vorrei quindi chiudere questo capitolo. Prima di farlo, alcuni punti di riepilogo su cosa ricordare su test-driving degli *oggetti valore*:

1. Gli *oggetti valore* vengono spesso (anche se non sempre) sottoposti a refactoring retroattivamente dal codice esistente. In tal caso, avranno già una certa copertura dalle specifiche delle classi che utilizzano questi *oggetti valore*.

Non è necessario specificare nuovamente i comportamenti coperti nella specifica dell'*oggetto valore*. Anche se lo faccio quasi sempre, Johnny e Benjamin hanno scelto di non farlo e rispetto la loro decisione.

2. Anche se scegliamo di non scrivere Statement per comportamenti già coperti dalle specifiche esistenti, dobbiamo comunque scrivere Statement aggiuntivi per garantire che un tipo sia un *oggetto valore* appropriato. Queste dichiarazioni non sono guidate dalla logica esistente, ma dai nostri principi di progettazione.
3. Esistono molte condizioni che si applicano ai codici di uguaglianza e hash degli *oggetti valore*. Invece di scrivere test per questi comportamenti per ogni *oggetto valore*, consiglio di utilizzare un'asserzione personalizzata generica per questo. Trovare una libreria che contenga già tale asserzione o scriverne una propria.
4. Quando si implementano metodi di uguaglianza e codici hash, anch'io consiglio di considerare fortemente l'utilizzo di qualche tipo di libreria di supporto o almeno di generarle utilizzando una funzionalità dell'IDE.
5. Non facciamo mai mock di *oggetti di valore* nelle Specifiche di altre classi. In queste Specifiche utilizziamo gli *oggetti valore* allo stesso modo degli `int` e delle `string`.

CAPITOLO 50

Raggiungere la rete dei confini degli oggetti

Quando si esegue il rilevamento dell'interfaccia outside-in e si implementa un collaboratore dopo l'altro, spesso arriva un punto in cui raggiungiamo i limiti della nostra applicazione, il che significa che dobbiamo eseguire qualche tipo di operazione di I/O (come chiamare un'API esterna tramite HTTP) o utilizzare un classe che fa parte di una sorta di pacchetto di terze parti (ad esempio fornito da un framework). In questi luoghi, la libertà con cui potremmo modellare la nostra realtà object-oriented è enormemente limitata da queste dipendenze. Anche se guidiamo ancora i comportamenti utilizzando la nostra intenzione, dobbiamo essere sempre più consapevoli che la direzione che prendiamo deve corrispondere alla tecnologia con cui comunichiamo con il mondo esterno.

Esistono molteplici esempi di risorse che si trovano ai confini della nostra rete di oggetti. File, thread, clock, database, canali di comunicazione (ad esempio http, bus di servizio, websocket), interfacce utente grafiche... queste risorse vengono utilizzate per implementare meccanismi per cose che svolgono un ruolo nel nostro design. Dobbiamo in qualche modo raggiungere un consenso tra ciò che questi meccanismi di implementazione richiedono per svolgere il loro lavoro e qual è il ruolo del dominio che devono ricoprire.

In questo capitolo descriverò diversi esempi di come raggiungere questo consenso con alcune indicazioni su come affrontarli. L'essenza di questa guida è nascondere tali dipendenze sotto interfacce che modellano i ruoli di cui abbiamo bisogno.

50.1 Che ore sono?

Considerazione l'idea di ottenere l'ora corrente.

Immaginiamo che la nostra applicazione gestisca sessioni che scadono a un certo punto. Modelliamo il concetto di sessione creando una classe chiamata `Session` e consentiamo di interrogarla se è scaduta o meno. Per rispondere a questa domanda, la sessione deve conoscere l'ora di scadenza e l'ora corrente e calcolare la loro differenza. In tal caso, possiamo modellare la sorgente dell'ora corrente come una sorta di astrazione di "orologio". Ecco alcuni esempi di come farlo.

50.1.1 Una query per l'ora corrente con un framework mock

Possiamo modellare l'orologio semplicemente come un servizio che fornisce l'ora corrente (ad esempio tramite una sorta di metodo `.CurrentTime()`). La classe `Session` è quindi responsabile dell'esecuzione del calcolo. Uno Statement di esempio che descrive questo comportamento potrebbe assomigliare a questo:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
 var expiryDate = Any.DateTime();
 var clock = Substitute.For<Clock>();
 var session = new Session(expiryDate, clock);

 clock.CurrentTime().Returns(expiryDate + TimeSpan.FromTicks(1));

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
 Assert.True(isExpired);
}
```

Ho bloccato il mio orologio per restituire un punto temporale specifico per rendere deterministica l'esecuzione dello Statement e per descrivere un comportamento specifico, che indica la scadenza della sessione.

50.1.2 Maggiore allocazione delle responsabilità nell'astrazione dell'orologio

Il nostro precedente tentativo di astrazione dell'orologio era semplicemente quello di astrarre i servizi garantiti da una fonte in tempo reale ("qual è l'ora corrente?"). Possiamo rendere questa astrazione più adatta al nostro caso d'uso specifico fornendogli un metodo `IsPast()` che accetterà un tempo di scadenza e ci dirà semplicemente se è passato o meno quel tempo. Uno degli Statements che utilizzano il metodo `IsPast()` potrebbe assomigliare a questo:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
 var expiryDate = Any.DateTime();
 var clock = Substitute.For<Clock>();
 var session = new Session(expiryDate, clock);

 clock.IsPast(expiryDate).Returns(true);

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
 Assert.True(isExpired);
}
```

Questa volta utilizzo anche un mock per sostituire un'implementazione di `Clock`.

Il vantaggio di ciò è che lastrazione è più sintonizzata sulle nostre esigenze specifiche. Lo svantaggio è che il calcolo della differenza temporale tra il tempo corrente e quello di scadenza viene effettuato in una classe che sarà quantomeno molto difficile da descrivere con uno Statement deterministico poiché si baserà sull'orologio del sistema.

50.1.3 Un orologio fake

Se i servizi forniti da un'interfaccia come `Clock` sono abbastanza semplici, possiamo creare la nostra implementazione da utilizzare negli Statement. Tale implementazione eliminerebbe tutta la fatica di lavorare con una vera risorsa esterna e ci fornirebbe invece una versione semplificata o più controllabile. Ad esempio, una implementazione fake dell'interfaccia `Clock` potrebbe assomigliare a questa:

```
class SettableClock : Clock
{
 public DateTime _currentTime = DateTime.MinValue;
```

(continues on next page)

(continua dalla pagina precedente)

```
 public void SetCurrentTime(DateTime currentTime)
 {
  _currentTime = currentTime;
 }

 public DateTime CurrentTime()
 {
  return _currentTime;
 }
}
```

Questa implementazione di `Clock` non ha solo un getter per l'ora corrente (che è ereditato dall'interfaccia `Clock`), ma ha anche un setter. L'"ora corrente" restituita dall'istanza `SettableClock` non proviene dall'orologio di sistema, ma può invece essere impostata su un valore specifico e deterministico. Ecco come appare l'uso di `SettableClock` in uno Statement:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
 var expiryDate = Any.DateTime();
 var clock = new SettableClock();
 var session = new Session(expiryDate, clock);

 clock.SetCurrentTime(expiryDate + TimeSpan.FromSeconds(1));

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
 Assert.True(isExpired);
}
```

Potrei anche realizzare la seconda versione dell'orologio - quella con il metodo `IsPast()` - in modo molto simile. Avrei solo bisogno di aggiungere un po' di intelligenza extra a `SettableClock`, duplicando piccoli frammenti dell'implementazione reale di `Clock`. In questo caso, non è un grosso problema, ma ci possono essere casi in cui questo può essere eccessivo. Affinché un fake sia garantito, l'implementazione fake deve essere molto, molto più semplice dell'implementazione reale che intendiamo utilizzare.

Un vantaggio di un falso è che può essere riutilizzato (ad esempio, un'astrazione di orologio impostabile riutilizzabile può essere trovata nelle librerie "Noda Time" e "Joda Time"). Uno svantaggio è che più il codice sofisticato si trova all'interno del falso, più è probabile che contenga bug. Se un falso molto intelligente vale ancora la pena nonostante la complessità (e ho visto diversi casi in cui lo era), prenderei in considerazione la possibilità di scrivere alcune specifiche sull'implementazione fake.

50.1.4 Altri usi di questo approccio

Lo stesso approccio che abbiamo utilizzato con un orologio di sistema può essere adottato con altre fonti di valori non deterministici, ad es. generatori di valori casuali.

50.2 Timer

I timer sono oggetti che consentono l'esecuzione di codice differita o periodica, solitamente in modo asincrono.

{{keyInfo}} Al giorno d'oggi in C#, quasi tutto ciò che è asincrono viene eseguito utilizzando la classe `Task` e le parole chiave `async-await`. Nonostante ciò, ho deciso di escluderli da questo capitolo per renderlo più comprensibile agli utenti non C#. Quindi perdonatemi per un esempio un po' meno realistico e spero che si possa mapparlo al proprio codice moderno.

In genere, l'utilizzo del timer è composto da tre fasi:

1. lo "scheduling" [*pianificazione*] della scadenza del timer per un momento specifico, passandogli una callback da eseguire alla scadenza. Di solito è possibile dire a un timer se continuare il ciclo successivo al termine di quello corrente o interromperlo.
2. Quando la scadenza del timer è schedulata, viene eseguita in modo asincrono in una sorta di thread in background.
3. Quando il timer scade, la callback passata durante la pianificazione viene eseguita e il timer inizia un altro ciclo oppure si arresta.

Dal punto di vista della progettazione della nostra collaborazione, le parti importanti sono i punti 1 e 3. Affrontiamoli uno per uno.

50.2.1 Schedulare un task periodico

Per prima cosa immaginiamo di creare una nuova sessione, aggiungerla a una sorta di cache e impostare un timer che scade ogni 10 secondi per verificare se i privilegi del proprietario della sessione sono ancora validi. Uno Statement per questo potrebbe assomigliare a questo:

```
[Fact] public void
ShouldAddCreatedSessionToCacheAndScheduleItsPeriodicPrivilegesRefreshWhenExecuted()
{
 //GIVEN
 var sessionData = Any.Instance<SessionData>();
 var id = Any.Instance<SessionId>();
 var session = Any.Instance<Session>();
 var sessionFactory = Substitute.For<SessionFactory>();
 var cache = Substitute.For<SessionCache>();
 var periodicTasks = Substitute.For<PeriodicTasks>();
 var command = new CreateSessionCommand(id, sessionFactory, cache, sessionData);

 sessionFactory.CreateNewSessionFrom(sessionData).Returns(session);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  cache.Add(id, session);
  periodicTasks.RunEvery(TimeSpan.FromSeconds(10), session.RefreshPrivileges);
 });
}
```

Notare che ho creato un'interfaccia `PeriodicTasks` per modellare un'astrazione per l'esecuzione... beh... di attività periodiche. Sembra un'astrazione generica e, se necessario, potrebbe essere resa un po' più orientata al dominio. Per il nostro esempio, dovrebbe andare bene. L'interfaccia `PeriodicTask` si presenta così:

```
interface PeriodicTasks
{
 void RunEvery(TimeSpan period, Action actionRanOnExpiry);
}
```

Nello Statement sopra ho solo specificato che dovrà essere schedulata un'operazione periodica. Specificare come l'implementazione `PeriodicTasks` svolge il suo lavoro non rientra nello "scope".

L'interfaccia `PeriodicTasks` è progettata in modo da poter passare un gruppo di metodi anziché una lambda, poiché la richiesta di lambda renderebbe più difficile confrontare gli argomenti tra le invocazioni previste e quelle effettive nello Statement. Quindi, se volessi pianificare un'invocazione periodica di un metodo che ha un argomento (ad esempio, un singolo `int`), aggiungerei un metodo `RunEvery` che sarebbe simile a questo:

```
interface PeriodicTasks
{
 void RunEvery(TimeSpan period, Action<int> actionRanOnExpiry, int argument);
}
```

oppure potrei rendere il metodo generico:

```
interface PeriodicTasks
{
 void RunEvery<TArg>(TimeSpan period, Action<TArg> actionRanOnExpiry, TArg argument);
}
```

Se avessi bisogno di insiemi di argomenti diversi, potrei semplicemente aggiungere più metodi all'interfaccia `PeriodicTasks`, a patto di non accoppiarlo a nessuna classe specifica relativa al dominio (se ne avessi bisogno, preferirei dividere `PeriodicTasks` in molte altre interfacce specifiche del dominio).

50.2.2 Scadenza

Specificare il caso in cui il timer scade e l'attività schedulata deve essere eseguita è ancora più semplice. Dobbiamo solo fare ciò che farebbe il codice del timer - richiamare il codice schedulato dal nostro Statement. Per il mio esempio di sessione, presupponendo che un'implementazione dell'interfaccia `Session` sia una classe chiamata `UserSession`, potrei usare il seguente Statement per descrivere un comportamento in cui la perdita dei privilegi per accedere al contenuto della sessione porta alla generazione di eventi:

```
[Fact] public void
ShouldNotifyObserverThatSessionIsClosedOnPrivilegesRefreshWhenUserLosesAccessToSessionContent()
{
 //GIVEN
 var user = Substitute.For<User>();
 var sessionContent = Any.Instance<SessionContent>();
 var sessionEventObserver = Substitute.For<SessionEventObserver>();
 var id = Any.Instance<SessionId>();
 var session = new UserSession(id, sessionContent, user, sessionEventObserver);

 user.HasAccessTo(sessionContent).Returns(false);

 //WHEN
 session.RefreshPrivileges();

 //THEN
 sessionEventObserver.Received(1).OnSessionClosed(id);
}
```

Ai fini di questo esempio, sto saltando altre istruzioni (ad esempio si potrebbe specificare un comportamento quando `RefreshPrivileges` viene chiamato più volte poiché è quello che farà il timer alla fine) e l'accesso multithread (le richiamate del timer sono in genere eseguiti da altri thread, quindi se accedono allo stato mutabile, questo stato deve essere protetto da modifiche simultanee).

50.3 Thread

Di solito vedo i thread utilizzati in due situazioni:

1. Per eseguire più attività in parallelo. Ad esempio, abbiamo diversi calcoli indipendenti e pesanti e vogliamo eseguirli contemporaneamente (non uno dopo l'altro) per far sì che l'esecuzione del codice finisca prima.
2. Per rinviare l'esecuzione di una parte della logica a un processo in background asincrono. Questo lavoro può essere ancora in esecuzione anche dopo che il metodo che lo ha avviato ha terminato l'esecuzione.

50.3.1 Esecuzione parallela

Nel primo caso -- quello di esecuzione parallela -- il multithreading è un dettaglio di implementazione del metodo chiamato dallo Statement. Lo Statement stesso non deve sapere nulla al riguardo. A volte, però, è necessario sapere che alcune operazioni potrebbero non essere eseguite ogni volta nello stesso ordine.

Consideriamo un esempio in cui valutiamo il pagamento per più dipendenti. Ogni valutazione è un'operazione costosa, quindi dal punto di vista dell'implementazione vogliamo farle in parallelo. Uno Statement che descrive tale operazione potrebbe assomigliare a questo:

```
public void [Fact]
ShouldEvaluatePaymentForAllEmployees()
{
 //GIVEN
 var employee1 = Substitute.For<Employee>();
 var employee2 = Substitute.For<Employee>();
 var employee3 = Substitute.For<Employee>();
 var employees = new Employees(employee1, employee2, employee3);

 //WHEN
 employees.EvaluatePayment();

 //THEN
 employee1.Received(1).EvaluatePayment();
 employee2.Received(1).EvaluatePayment();
 employee3.Received(1).EvaluatePayment();
}
```

Notare che lo Statement non menziona affatto il multithreading, perché questo è il dettaglio dell'implementazione del metodo `EvaluatePayment` su `Employees`. Tuttavia, si noti anche che lo Statement non specifica l'ordine in cui viene valutato il pagamento per ciascun dipendente. In parte ciò è dovuto al fatto che l'ordine non ha importanza e lo Statement lo descrive accuratamente. Se, tuttavia, lo Statement specificasse l'ordine, non solo sarebbe eccessivamente specificato, ma potrebbe anche essere valutato come vero o falso in modo non deterministico. Questo perché in caso di esecuzione parallela, l'ordine in cui i metodi verrebbero richiamati su ciascun dipendente potrebbe essere diverso.

50.3.2 Attività in background

Usare i thread per eseguire attività in background è come usare timer che vengono eseguiti solo una volta. Basta usare un approccio simile ai timer.

50.4 Altro

Esistono altre dipendenze come dispositivi I/O, generatori di valori casuali o SDK di terze parti (ad esempio un SDK per la connessione a un bus di messaggi), ma non li approfondirò poiché la strategia per loro è la stessa - non usarli direttamente nel codice relativo al proprio dominio. Pensare invece al ruolo che svolgono a livello di dominio e modellare quel ruolo come un'interfaccia. Quindi utilizzare la risorsa problematica all'interno di una classe che implementa questa interfaccia. Tale classe sarà coperta da un altro tipo di Specifica che tratterò più avanti nel libro.

CAPITOLO 51

Cosa c'è dentro l'oggetto?

51.1 Quali sono i peer [*pari/colleghi*] dell'oggetto?

Finora ho parlato molto di oggetti che si compongono in una rete e comunicano inviandosi messaggi tra loro. Lavorano insieme come *peers*.

Il nome peer deriva dal fatto che gli oggetti lavorano su un piano di parità -- non esiste una relazione gerarchica tra peer Quando due pari collaborano, nessuno di loro può essere definito "proprietario" dell'altro. Sono collegati da un oggetto che riceve un riferimento a un altro oggetto, il suo "indirizzo".

Ecco un esempio

```
class Sender
{
 public Sender(Recipient recipient)
 {
  _recipient = recipient;
  //...
 }
}
```

In questo esempio, il `Recipient` è un peer di `Sender` (a condizione che il `Recipient` non sia un *oggetto valore* o una struttura dati). Il mittente non sa:

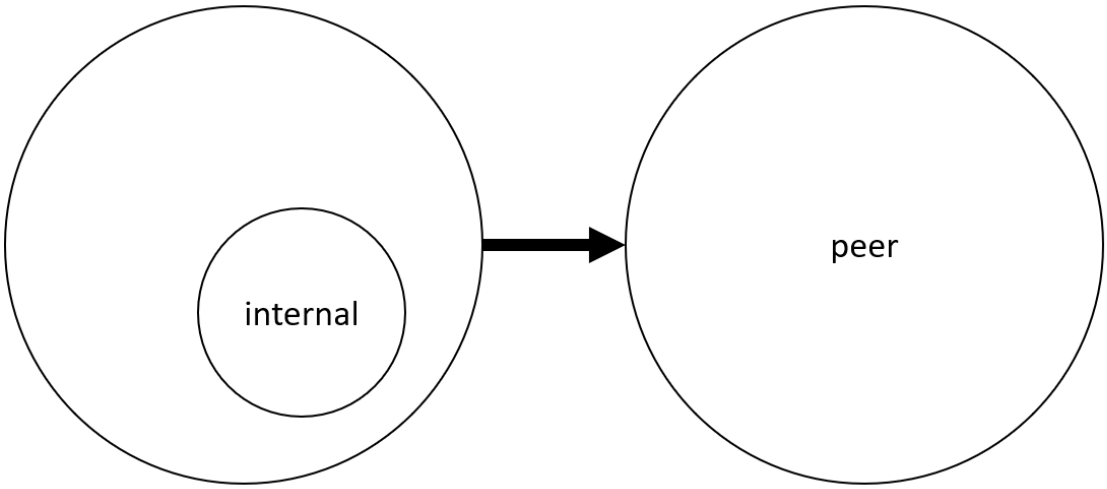
- il tipo concreto di `recipient`
- quando è stato creato il `recipient`
- da chi è stato creato il `recipient`
- per quanto tempo vivrà l'oggetto `recipient` dopo che l'istanza `Sender` sarà stata distrutta
- quali altri oggetti collaborano con `recipient`

51.2 Quali sono le parti interne dell'oggetto?

Non tutti gli oggetti fanno parte della rete. Ho già menzionato gli *oggetti valore* e le strutture dati. Potrebbero far parte dell'API pubblica di una classe ma non sono suoi pari.

Un'altra categoria utile sono gli *interni* -- oggetti creati e posseduti da altri oggetti, legati alla durata della vita dei loro proprietari.

Gli *interni* sono incapsulati negli oggetti proprietari, nascosti al mondo esterno. L'esposizione degli *interni* dovrebbe essere considerata una rara eccezione piuttosto che una regola. Fanno parte del motivo per cui i loro oggetti proprietari si comportano in un certo modo, ma non possiamo passare la nostra implementazione per personalizzare quel comportamento. Questo è anche il motivo per cui non possiamo avere mock degli *interni* - non esiste un punto di estensione che possiamo utilizzare. Per fortuna anche noi non vogliamo. Intromettersi negli *interni* di un oggetto significherebbe rompere l'incapsulamento e l'accoppiamento con le cose che sono volatili nel nostro design.



A volte gli *interni* possono essere passati ad altri oggetti senza interrompere l'incapsulamento. Diamo un'occhiata a questo pezzo di codice:

```
public class MessageChain
{
 private IReadOnlyList<Message> _messages = new List<Message>();
 private string _recipientAddress;

 public MessageChain(string recipientAddress)
 {
  _recipientAddress = recipientAddress;
 }

 public void Add(string title, string body)
 {
  _messages.Add(new Message(_recipientAddress, title, body));
 }

 public void Send(MessageDestination destination)
 {
  destination.SendMany(_messages);
 }
}
```

In questo esempio, l'oggetto `_messages` viene passato a `destination`, ma la `destination` non sa da dove ha preso questo elenco, quindi questa interazione non espone necessariamente i dettagli strutturali della classe `MessageChain`.

La distinzione tra peer e *interni* è stata introdotta da Steve Freeman e Nat Pryce nel loro libro Growing Object-Oriented Software Guided By Tests.

51.3 Esempi di *interni*

Come scoprire che un oggetto dovrebbe essere un *interno* piuttosto che un pari? Di seguito ho elencato diversi esempi di categorie di oggetti che possono diventare *interni* di altri oggetti. Spero che questi esempi aiutino ad allenare il senso di identificazione gli *interni* nel progetto.

51.3.1 Primitive

Consideriamo la seguente classe `CountingObserver` che conta quante volte è stata notificata e quando viene raggiunta una soglia, passa poi oltre tale notifica:

```
class ObserverWithThreshold : Observer
{
 private int _count = 0;
 private Observer _nextObserver;
 private int _threshold;

 public ObserverWithThreshold(int threshold, Observer nextObserver)
 {
  _threshold = threshold;
  _nextObserver = nextObserver;
 }

 public void Notify()
 {
  _count++;
  if(_count > _threshold)
  {
    _count = 0;
    _nextObserver.Notify();
  }
 }
}
```

Il campo `_count` è di proprietà e gestito da un'istanza `CountingObserver`. È invisibile all'esterno di un oggetto `ObserverWithThreshold`.

Un esempio di Statement di comportamento per questa classe potrebbe assomigliare a questo:

```
[Fact] public void
ShouldNotifyTheNextObserverWhenItIsNotifiedMoreTimesThanTheThreshold()
{
 //GIVEN
 var nextObserver = Substitute.For<Observer>();
 var observer = new ObserverWithThreshold(2, nextObserver);

 observer.Notify();
 observer.Notify();

 //WHEN
 observer.Notify();

 //THEN
 nextObserver.Received(1).Notify();
}
```

Il conteggio corrente delle notifiche non viene esposto all'esterno dell'oggetto ObserverWithThreshold. Sto solo superando la soglia che il conteggio delle notifiche deve superare. Se davvero lo volessi, potrei, invece di usare un int, introdurre un'interfaccia di un collaboratore per il contatore, ad es. chiamato `Counter`, fornirgli metodi come `Increment()` e

`GetValue()` e passa un mock dallo Statement, ma per me, se tutto ciò che voglio è contare quante volte viene chiamato qualcosa, preferirei renderlo parte dell'implementazione della classe per eseguire il conteggio. Sembra più semplice se il counter non è esposto.

51.3.2 Campi *oggetto valore*

Il contatore dell'ultimo esempio era già un valore, ma ho pensato di menzionare *oggetti valore* più ricchi.

Consideriamo una classe che rappresenta una sessione della riga di comando. Permette di eseguire comandi nell'ambito di una directory di lavoro. Un'implementazione di questa classe potrebbe assomigliare a questa:

```
public class CommandlineSession
{
 private AbsolutePath _workingDirectory;

 public CommandlineSession(PathRoot root)
 {
  _workingDirectory = root.AsAbsolutePath();
 }

 public void Enter(DirectoryName dirName)
 {
  _workingDirectory = _workingDirectory.Append(dirName);
 }

 public void Execute(Command command)
 {
  command.ExecuteIn(_workingDirectory);
 }
 //...
}
```

Questa classe `CommandlineSession` ha un campo privato chiamato `_workingDirectory`, che rappresenta la directory di lavoro della sessione corrente della riga di comando. Anche se il valore iniziale è basato sull'argomento del costruttore, da quel momento in poi il campo viene gestito internamente e passato solo a un comando in modo che sappia dove eseguirlo. Uno Statement di esempio per il comportamento di `CommandLineSession` potrebbe assomigliare a questo:

```
[Fact] public void
ShouldExecuteCommandInEnteredWorkingDirectory()
{
 //GIVEN
 var pathRoot = Any.Instance<PathRoot>();
 var commandline = new CommandLineSession(pathRoot);
 var subDirectoryLevel1 = Any,Insytance<DirectoryName>();
 var subDirectoryLevel2 = Any,Insytance<DirectoryName>();
 var subDirectoryLevel3 = Any,Insytance<DirectoryName>();
 var command = Substitute.For<Command>();

 commandline.Enter(subDirectoryLevel1);
 commandline.Enter(subDirectoryLevel2);
 commandline.Enter(subDirectoryLevel3);

 //WHEN
 commandLine.Execute(command);

 //THEN
 command.Received(1).Execute(
  AbsolutePath.Combine(
```

(continues on next page)

(continua dalla pagina precedente)

```
   pathRoot,
   subDirectoryLevel1,
   subDirectoryLevel2,
   subDirectoryLevel3));
}
```

Ancora una volta, non ho accesso al campo interno `_workingDirectory`. Posso solo prevederne il valore e creare un valore atteso nel mio Statement. Notare che non sto nemmeno utilizzando gli stessi metodi per combinare i path sia nello Statement che nel codice di produzione - mentre il codice di produzione utilizza il metodo `Append()`, il mio Statement utilizza un metodo statico `Combine` su un `AbsolutePath`. Ciò dimostra che il mio Statement ignora come esattamente lo stato interno viene gestito dalla classe `CommandLineSession`.

51.3.3 Le Collection

Le semplici collection di elementi (come elenchi, hashset, array ecc.) non vengono generalmente visualizzate come peer. Anche se scrivo una classe che accetta un'interfaccia di collection (ad esempio `IList` in C#) come parametro, non creo mai mock dell'interfaccia della collection, ma piuttosto utilizzo una delle collection native.

Ecco un esempio di una classe `InMemorySessions` che inizializza e utilizza una collection:

```
public class InMemorySessions : Sessions
{
 private Dictionary<SessionId, Session> _sessions
  = new Dictionary<SessionId, Session>();
 private SessionFactory _sessionFactory;

 public InMemorySessions(SessionFactory sessionFactory)
 {
  _sessionFactory = sessionFactory;
 }

 public void StartNew(SessionId id)
 {
  var session = _sessionFactory.CreateNewSession(id);
  session.Start();
  _sessions[id] = session;
 }

 public void StopAll()
 {
  foreach(var session in _sessions.Values())
  {
   _session.Stop();
  }
 }
 //...
}
```

Il dizionario qui utilizzato non è affatto esposto al mondo esterno. Viene utilizzato solo internamente. Non riesco a superare un'implementazione mock e, anche se potessi, preferirei lasciare il comportamento come posseduto da `InMemorySessions`. Un esempio di Statement per la classe `InMemorySessions` ha mostrato come il dizionario non sia visibile all'esterno della classe:

```
[Fact] public void
ShouldStopAddedSessionsWhenAskedToStopAll()
{
 //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
 var sessionFactory = Substitute.For<SessionFactory>();
 var sessions = new InMemorySessions(sessionFactory);
 var sessionId1 = Any.Instance<SessionId>();
 var sessionId2 = Any.Instance<SessionId>();
 var sessionId3 = Any.Instance<SessionId>();
 var session1 = Substitute.For<Session>();
 var session2 = Substitute.For<Session>();
 var session3 = Substitute.For<Session>();

 sessionFactory.CreateNewSession(sessionId1)
  .Returns(session1);
 sessionFactory.CreateNewSession(sessionId2)
  .Returns(session2);
 sessionFactory.CreateNewSession(sessionId3)
  .Returns(session3);

 sessions.StartNew(sessionId1);
 sessions.StartNew(sessionId2);
 sessions.StartNew(sessionId3);

 //WHEN
 sessions.StopAll();

 //THEN
 session1.Received(1).Stop();
 session2.Received(1).Stop();
 session3.Received(1).Stop();
}
```

51.3.4 Classi e oggetti toolbox

Classi e oggetti toolbox non sono realmente astrazioni di alcun dominio problematico specifico, ma aiutano a rendere l'implementazione più concisa, riducendo il numero di righe di codice che devo scrivere per portare a termine il lavoro. Un esempio è una classe C# `Regex` per le espressioni regolari. Ecco un esempio di calcolatore del conteggio delle righe che utilizza un'istanza di `Regex` per contare il numero di righe in una porzione di testo:

```
class LocCalculator
{
 private static readonly Regex NewlineRegex
  = new Regex(@"\r\n|\n", RegexOptions.Compiled);

 public uint CountLinesIn(string content)
 {
  return NewlineRegex.Split(contentText).Length;
 }
}
```

Ancora una volta, ritengo che la conoscenza su come dividere una stringa in più righe dovrebbe appartenere alla classe `LocCalculator`. Non introdurrei né creerei mock di un'astrazione (ad esempio chiamata `LineSplitter` a meno che non ci fossero qualche tipo di regole di dominio associate alla suddivisione del testo). Un esempio di Statement che descrive il comportamento della calcolatrice sarebbe simile a questo:

```
[Fact] public void
ShouldCountLinesDelimitedByCrLf()
{
 //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
 var text = $"{Any.String()}\r\n{Any.String()}\r\n{Any.String()}";
 var calculator = new LocCalculator();

 //WHEN
 var lineCount = calculator.CountLinesIn(text);

 //THEN
 Assert.Equal(3, lineCount);
}
```

L'oggetto dell'espressione regolare non si vede da nessuna parte - rimane nascosto come dettaglio di implementazione della classe `LocCalculator`.

51.3.5 Alcune classi di librerie di terze parti

Di seguito è riportato un esempio che utilizza un framework di injection [*inserimento*] di errori C#, Simmy. La classe decora una vera classe di archiviazione e consente di configurare il lancio di eccezioni invece di parlare con l'oggetto di archiviazione. L'esempio potrebbe sembrare un po' contorto e la classe non è comunque pronta per la produzione. Notare che molte classi e metodi vengono utilizzati solo all'interno della classe e non sono visibili dall'esterno.

```
public class FaultInjectablePersonStorage : PersonStorage
{
 private readonly PersonStorage _storage;
 private readonly InjectOutcomePolicy _chaosPolicy;

 public FaultInjectablePersonStorage(bool injectionEnabled, PersonStorage storage)
 {
  _storage = storage;
  _chaosPolicy = MonkeyPolicy.InjectException(with =>
    with.Fault(new Exception("thrown from exception attack!"))
     .InjectionRate(1)
     .EnabledWhen((context, token) => injectionEnabled)
  );
 }

 public List<Person> GetPeople()
 {
  var capturedResult = _chaosPolicy.ExecuteAndCapture(() => _storage.GetPeople());
  if (capturedResult.Outcome == OutcomeType.Failure)
  {
   throw capturedResult.FinalException;
  }
  else
  {
   return capturedResult.Result;
  }
 }
}
```

Uno Statement di esempio potrebbe assomigliare a questo:

```
[Fact] public void
ShouldReturnPeopleFromInnerInstanceWhenTheirRetrievalIsSuccessfulAndInjectionIsDisabled()
{
 //GIVEN
 var innerStorage = Substitute.For<PersonStorage>();
 var peopleInInnerStorage = Any.List<Person>();
```

(continues on next page)

(continua dalla pagina precedente)

```
 var storage = new FaultInjectablePersonStorage(false, innerStorage);

 innerStorage.GetPeople().Returns(peopleFromInnerStorage);

 //WHEN
 var result = storage.GetPeople();

 //THEN
 Assert.Equal(peopleInInnerStorage, result);
}
```

e della libreria Simmy non c'è traccia.

51.4 Riepilogo

In questo capitolo ho sostenuto che non tutte le comunicazioni tra oggetti dovrebbero essere rappresentate come protocolli pubblici. Invece, una parte di esso dovrebbe essere incapsulata all'interno dell'oggetto. Ho anche fornito diversi esempi per a trovare tali collaboratori *interni*.

CAPITOLO 52

Odori del design visibile nella Specifica

Nei capitoli precedenti, ho sottolineato più volte come gli oggetti mock possano e debbano essere utilizzati come strumento di progettazione, contribuendo a definire i protocolli tra oggetti collaborativi. Questo perché esiste qualcosa che chiamo *Discordanza tra approccio di test e di progettazione*. Il modo in cui ho scelto di esprimerlo è:

"L'approccio di automazione dei test e l'approccio di progettazione devono vivere in simbiosi, basandosi su un insieme di presupposti simili e puntando agli stessi obiettivi. Se lo fanno, si rafforzano a vicenda. Se non lo fanno, ci saranno attriti tra loro".

Ritengo che questo sia universalmente vero. Ad esempio, se testo un'applicazione asincrona come se stessi testando un'applicazione sincrona o se provo a testare un'API web JSON utilizzando uno strumento di clic, il mio codice non funzionerà correttamente o avrà un aspetto strano. Probabilmente dovrò impegnarmi di più per compensare la discrepanza tra l'approccio di test e quello di progettazione. Per questo motivo, alcuni che preferiscono approcci di progettazione molto diversi considerano gli oggetti mock un odore.

Lo sto riscontrando anche a livello di unità. In questo libro, utilizzo la terminologia di specifica invece di quella di test. Quindi, posso dire che, scrivendo la mia Specifica a livello di unità e utilizzando oggetti mock nelle mie Istruzioni, mi aspetto che il mio progetto abbia determinate proprietà e trovo queste proprietà desiderabili. Se il mio codice non mostra queste proprietà, mi renderà la vita più difficile. L'altro lato della medaglia è che se mi alleno a riconoscere le situazioni in cui gli approcci di progettazione e di specifica divergono e gli schemi con cui ciò avviene, posso usare questa conoscenza per migliorare il mio progetto.

In questo capitolo, presento alcuni degli odori che si possono vedere nelle Istruzioni ma che in realtà sono problemi di progettazione. Tutti questi odori provengono dalla community e li potete trovare catalogati altrove.

52.1 Catalogo degli odori del design

52.1.1 Specificare lo stesso comportamento in molti punti

A volte, quando scrivo un'istruzione, ho un déjà vu e inizio a pensare "Ricordo di aver già specificato un comportamento come questo". Mi capita soprattutto quando quell'altra volta non è poi così lontana, magari anche solo un paio di minuti fa.

Considerate la seguente Istruzione che descrive una logica di generazione di regole di reporting:

```
[Fact] public void
ShouldReportAllEmployeesWhenExecuted()
{
  //GIVEN
```

(continues on next page)

(continua dalla pagina precedente)

```
  var employee1 = Substitute.For<Employee>();
  var employee2 = Substitute.For<Employee>();
  var employee3 = Substitute.For<Employee>();
  var allEmployees = new List<Employee>() {employee1, employee2, employee3};
  var reportBuilder = Substitute.For<ReportBuilder>();
  var employeeReportingRule = new EmployeeReport(allEmployees, reportBuilder);

  //WHEN
  employeeReportingRule.Execute();

  //THEN
  Received.InOrder(() =>
  {
    reportBuilder.AddHeader();
    employee1.AddTo(report);
    employee2.AddTo(report);
    employee3.AddTo(report);
    reportBuilder.AddFooter();
  });
}
```

Ora, un'altra Istruzione per una regola diversa:

```
[Fact] public void
ShouldReportTotalCost()
{
  //GIVEN
  var calculatedTotalCost = Any.Integer();
  var reportBuilder = Substitute.For<ReportBuilder>();
  var costReportingRule = new EmployeeReport(totalCost, reportBuilder);

  //WHEN
  employeeReportingRule.Execute();

  //THEN
  Received.InOrder(() =>
  {
    reportBuilder.AddHeader();
    reportBuilder.AddTotalCost(totalCost);
    reportBuilder.AddFooter();
  });
}
```

E notate che devo specificare di nuovo che un'intestazione e un piè di pagina vengono aggiunti al report. Questo potrebbe significare che c'è ridondanza nel design e forse ogni report, o almeno la maggior parte di essi, necessita di un'intestazione e un piè di pagina. Questa osservazione, amplificata dalla sensazione di inutilità nel descrivere lo stesso comportamento più volte, potrebbe portarmi a provare a estrarre una logica in un oggetto separato, magari un decoratore, che aggiungerà un piè di pagina prima della parte principale del report e un piè di pagina alla fine.

Quando questo non è un problema

Il trucco per riconoscere la ridondanza nella progettazione è rilevare se il comportamento in entrambi i punti cambierà per le stesse ragioni e nello stesso modo. Questo è solitamente vero se entrambi i punti duplicano la stessa regola di dominio. Ma considerate queste due funzioni:

```
double CmToM(double value)
{
```

(continues on next page)

(continua dalla pagina precedente)

```
  return value/100;
}

double PercentToFraction(double value)
{
  return value/100;
}
```

Sebbene l'implementazione sia la stessa, la regola di dominio è diversa. In tali situazioni, non stiamo parlando di ridondanza, ma di coincidenza. Vedere anche l'eccellente post di Vladimir Khorikov su questo argomento.

La ridondanza nel codice è spesso definita "violazione DRY (don't repeat yourself)". Da ricordare che DRY riguarda le regole di dominio, non il codice.

52.1.2 Molti mock

Finora, ho pubblicizzato l'utilizzo di mock nella modellazione dei ruoli nelle Statement. Quindi, il mocking è una buona cosa, giusto? Beh, non esattamente. Attraverso gli oggetti mock, possiamo vedere come la classe che specifichiamo interagisce con il suo contesto. Se le interazioni sono complesse, le Istruzioni lo mostreranno. Questo non è qualcosa che dovremmo ignorare. Al contrario, la capacità di vedere questo problema attraverso le nostre istruzioni è uno dei motivi principali per cui utilizziamo i mock.

Uno dei sintomi dei problemi di progettazione è l'eccesso di mock. Possiamo provare fastidio sia durante la scrittura dell'istruzione ("oh no, mi serve un altro mock") sia durante la lettura ("così tanti mock, non riesco a vedere dove succede qualcosa di reale").

Fare troppo

Potrebbe essere che una classe abbia bisogno di molte dipendenze perché fa troppe cose. Potrebbe eseguire la convalida, leggere la configurazione, sfogliare record persistenti, gestire errori, ecc. Il problema è che quando la classe è accoppiata a troppe cose, ci sono molte ragioni per cui la classe potrebbe cambiare.

Un rimedio potrebbe essere quello di estrarre parti della logica in classi separate in modo che la classe specificata faccia meno con meno peer.

52.1.3 Mock che restituiscono mock che restituiscono mock

I mock che restituiscono mock che restituiscono mock in genere indicano che il nostro codice non è costruito attorno all'euristica "tell, don't ask" e che la classe specificata è accoppiata a troppi tipi.

52.1.4 Troppi mock inutilizzati

Se passiamo molti mock in ogni Istruzione solo per riempire l'elenco dei parametri, potrebbe essere un segno di scarsa coesione.

Avere molti mock inutilizzati è accettabile nelle classi facade, perché lo scopo delle facade è semplificare l'accesso a un sottosistema e in genere contengono riferimenti a molti oggetti, facendo poco con ciascuno di essi.

52.1.5 Blocco Stubdel e verifica della stessa chiamata

Consideriamo un esempio in cui la nostra applicazione gestisce un qualche tipo di risorsa e ogni richiedente ottiene uno "slot" in questa risorsa. La risorsa è rappresentata dalla seguente interfaccia:

```
interface ISharedResource
{
  public SlotId AssignSlot();
}
```

quando si specificano i comportamenti di qualsiasi classe che utilizza questa risorsa, verrà utilizzato un mock:

```
var resource = Substitute.For<ISharedResource>();
```

vogliamo descrivere che una prenotazione di slot in una risorsa condivisa viene tentata una sola volta, quindi avremo la seguente chiamata sul nostro mock:

```
resource.Received(1).AssignSlot();
```

Ma dovremo anche descrivere cosa succede quando lo slot viene assegnato correttamente o quando l'assegnazione fallisce, quindi dobbiamo configurare il mock in modo che restituisca qualcosa:

```
resource.AssignSlot().Returns(slotId);
```

Avere sia l'impostazione di un valore di ritorno che la verifica della chiamata allo stesso metodo in un'istruzione è un segno che stiamo violando il principio di separazione comando-query. Questo limiterà la nostra capacità di elevare il polimorfismo nelle implementazioni di questa interfaccia.

Ma cosa possiamo fare al riguardo?

passare una continuazione

Potremmo passare un ruolo più completo che continuerà il flusso. Di seguito è riportato un esempio di passaggio di una cache in cui lo slot assegnato verrà salvato se l'assegnazione ha esito positivo:

```
resource.Received(1).AssignSlot(cache);
```

passaggio di una callback

potremmo creare un ruolo speciale per un destinatario del risultato e passarlo all'interno del metodo `AssignSlot`, quindi farlo agire sul risultato:

```
Received.InOrder(() =>
{
  resource.AssignSlot(assignSlotResponse);
  assignSlotResponse.Received(1).DisplayOn(screen);
})
```

passare i collaboratori necessari attraverso il costruttore

A volte potremmo rimuovere completamente la nozione di risultato dall'interfaccia e rendere il suo passaggio un dettaglio implementativo. Ecco l'Istruzione in cui non specifichiamo nulla sulla gestione di un potenziale risultato:

```
resource.Received(1).AssignSlot();
```

le implementazioni sono ancora libere di fare ciò che vogliono (o nulla) a seconda del successo o del fallimento dell'assegnazione. Una di queste implementazioni potrebbe avere un costruttore simile a questo:

```
public CacheBackedSharedResource(IAssignmentCache cache) { ... }
```

e potrebbe usare il suo argomento costruttore per delegare parte della logica.

Quando questo non è un problema?

TODO: I/O boundary - a volte è più facile restituire qualcosa che far passare un collaboratore attraverso un limite architetturale.

52.1.6 Tentare di simulare un'interfaccia di terze parti

Il classico libro sul TDD, Growing Object-Oriented Software Guided By Tests, suggerisce di non simulare tipi di cui non si è proprietari. Poiché i mock sono uno strumento di progettazione, vogliamo creare mock solo di tipi che possiamo controllare e che rappresentano astrazioni. Altrimenti siamo vincolati dalla definizione dell'interfaccia "così com'è" e non possiamo migliorare i nostri Statement migliorando il design.

Significa che se ho una classe `File`, posso semplicemente racchiuderla in un'interfaccia sottile e creare un'interfaccia `IFile` per abilitare il mocking e sono a posto? Se il vincolo che impongo a questa interfaccia è che sia 1:1 rispetto al tipo che non possiedo, allora si torna al punto di partenza: ho ancora un'interfaccia definita in un modo che potrebbe rendere difficile la scrittura di Statement con mock e non c'è nulla che io possa fare al riguardo.

Quindi cosa dovrei fare quando sono vicino al limite di I/O? Dovrei usare un tipo diverso di Statement, che vi mostrerò in una delle prossime parti di questo libro.

E il logging?

Anche il libro GOOS affronta questo argomento: i logger delle librerie sono spesso onnipresenti in molte codebase. TODO https://learn.microsoft.com/en-us/dotnet/core/extensions/logger-message-generator and Support class.

Test lampeggianti a causa dei dati generati

Questo specifico "odore" non è correlato agli oggetti mock, ma piuttosto all'utilizzo del non determinismo vincolato.

Quando un'Istruzione diventa instabile a causa dei dati generati come valori diversi a ogni esecuzione, ciò potrebbe significare che il comportamento è troppo dipendente dai dati e manca di astrazione. Il codice potrebbe apparire così:

```
if(customer.Accounts.First(a => a.IsPrimary).IsActive)
{
  customer.Status = CustomerStatus.Active;
}
```

Si noti che il codice controlla il flag `IsActive` (un valore booleano) del primo account impostato come primario (di nuovo, utilizzando un flag booleano `IsPrimary`). I valori booleani sono particolarmente vulnerabili alla generazione non deterministica perché hanno due possibili valori, ognuno dei quali si trova in genere in una classe di equivalenza diversa. E qui ne abbiamo due. Quindi mi aspetterei che le Istruzioni che utilizzano dati generati falliscano regolarmente in modo non deterministico.

Una delle possibili soluzioni in questo caso potrebbe essere quella di creare un'astrazione sulla struttura dati del cliente e utilizzare un mock invece di una struttura dati generata. Qualcosa del genere:

```
if(customer.HasActiveAccount())
{
  customer.Activate();
}
```

Questa è la soluzione più ingenua, da manuale. In realtà, potremmo persino finire per ripensare l'intero concetto di attivazione e questo potrebbe portarci a inventare nuovi ruoli interessanti.

Molte volte ho incontrato persone che usavano generatori di dati come `Any` come lasciapassare per far circolare strutture dati mostruose e complicate (perché "perché no? Posso generarle con una singola chiamata a `Any.Instance<>()`"). Tuttavia, credo che dovrebbero lavorare nella direzione opposta: introducendo la nozione di non determinismo, mi costringono a essere più paranoico riguardo al contesto in cui opera il mio oggetto, per ridurre al minimo le sue "parti mobili".

TODO: esempio

TODO: mancanza di astrazione

TODO: ne ho bisogno?

Separazione dei ruoli troppo fine

Possiamo introdurre molti ruoli? Probabilmente.

Ad esempio, potremmo avere una classe cache che svolge tre ruoli: `ItemWriter` per salvare gli elementi, `ItemReader` per leggerli ed `ExpiryCheck` per verificare se un elemento specifico è scaduto. Sebbene consideri una buona idea puntare a ruoli più precisi, se tutti vengono utilizzati nello stesso posto, ora dobbiamo creare tre mock per quello che sembra un insieme coerente di obblighi distribuiti su tre interfacce diverse.

52.2 Descrizione generale

cosa si intende per smell [*puzza*]? perché test *puzzolenti* indicano codice *puzzolente*?

52.3 Essenze

- la stessa cosa in molti posti
  - Ridondanza
  - Quando questo non è un problema - disaccoppiamento di contesti diversi, nessuna reale ridondanza
- Molti mock/ Grossi costruttori (GOOS) / Troppe dipendenze (GOOS)
  - Accoppiamento
  - Mancanza di astrazioni
  - Non è un problema quando: facade
- Blocco Stubdel e verifica della stessa chiamata
  - Rompere il CQS
  - Quando non è un problema: vicino al confine dell'I/O
- Test blinking (Any + flag booleani e if su di essi, più modi per ottenere le stesse informazioni)
  - troppa attenzione ai dati invece che alle astrazioni
  - mancanza di incapsulamento dei dati
  - Quando non è un problema: quando è solo un problema di test (test scritto male). Ancora un problema ma non un problema di progettazione.
- Configurazioni eccessive (sostenibilità) - sia preparando i dati & preparando l'unità con chiamate aggiuntive
  - protocolli prolissi
  - problemi di coesione
  - accoppiamento (ad esempio a molti dati necessari)
  - Quando non è un problema: test di livello superiore (ad esempio caricamento della configurazione) - forse questo significa ancora un'architettura di livello superiore scadente?
- Esplosione combinatoria (TODO: confermare nome)
  - problemi di coesione?
  - livello di astrazione troppo basso (ad esempio, "if" alla collection)
  - Quando non è un problema: tabelle decisionali (la logica del dominio si basa su molte decisioni e le abbiamo già disaccoppiate da altre parti della logica)
- Necessità di affermarsi su campi privati
  - Coesione
- test pass-through (test di forwarding [*inoltro*] semplice delle chiamate)
  - troppi livelli di astrazione.

  - Succede anche nei decoratori quando le interfacce decorate sono troppo ampie
- Imposta la coerenza delle chiamate mock (molti modi per ottenere le stesse informazioni, il codice testato può utilizzarli entrambi, quindi dobbiamo impostare più chiamate per comportarsi in modo coerente)
  - Potrebbe significare un'interfaccia troppo ampia con troppe opzioni
  - o livello di astrazione insufficiente
- test eccessivamente protettivo (sustainabletdd) - controllo di comportamenti diversi da quello previsto per paura
  - mancanza di coesione
  - accoppiamento nascosto?
- Avere mock/preparare framework/oggetti di libreria (ad esempio HttpRequest o sth).
  - Nessun mock di ciò che non si possiede?
  - Nel codice del dominio: accoppiamento ai dettagli tecnici
- Matcher liberali (perdita del controllo su come gli oggetti fluiscono attraverso i metodi)
  - coesione
  - protocolli prolissi
  - separare l'uso dalla costruzione
- Incapsulare il protocollo (nascondere la configurazione del mock e la verifica dietro metodi helper perché sono troppo complessi o si ripetono in ogni test)
  - Coesione (il codice ripetuto potrebbe essere spostato in una classe separata)
- Ho bisogno di simulare un oggetto che non posso sostituire (GOOS)
- Moc di classi concrete (GOOS)
- Mock di oggetti simili a valori (GOOS)
  - creare oggetti anziché valori
- Oggetto confuso (GOOS)
- Troppe expectation (GOOS)
- Molti test in uno (difficile da configurare)
- guardare anche qui: https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html

CAPITOLO 53

QUESTO È TUTTO QUELLO CHE HO PER ORA. QUELLO CHE SEGUE È MATERIALE GREZZO, NON ORDINATO, NON ANCORA PRONTO PER ESSERE CONSUMATO COME PARTE DI QUESTO TUTORIAL

CAPITOLO 54

Mock di oggetti come strumento di progettazione

54.1 Design Responsibility-Driven [*guidato dalla responsabilità*]

TDD con mock segue RDD, e allora? Ruoli modello. Tutto il resto potrebbe rientrare nei valori e negli *interni*. Ma esiste già una metafora della rete

Ok, quindi forse una scheda CRC?

Non tutte le classi fanno parte della rete. Un esempio sono gli oggetti valore. Un altro esempio sono le semplici strutture dati o DTO.

CAPITOLO 55

Guidaa ai test smells [*puzzolenti*]

55.1 Statement Lunghi

55.2 Molti stub

55.3 Specifire i membri privati

CAPITOLO 56

Rivedere gli argomenti del capitolo 1

56.1 Non determinismo vincolato nel mondo OO

56.1.1 Ruoli passivi e attivi

56.2 Confini comportamentali

56.3 Triangolazione

CAPITOLO 57

Statement manutenibili basate su mock

57.1 Setup e teardown

CAPITOLO 58

Refactoring del codice mock

finché non lo si fa uscire tramite il costruttore, sono affari privati dell'oggetto.

i mock si basano sulla stabilità dei confini. Se questo è sbagliato, i test devono essere riscritti, ma il feedback dei test consente di stabilizzare ulteriormente i confini. E non ci sono molti test da modificare mentre testiamo piccole parti di codice.

interazioni ben progettate dovrebbero limitare limpatto del cambiamento

CAPITOLO 59

Parte 4: Architettura dell'applicazione

CAPITOLO 60

Sui confini stabili/architetturali

CAPITOLO 61

[Port] e adapter

61.1 Separazione fisica degli layer

61.1.1 Architettura "urlante".

CAPITOLO 62

Cosa entra in applicazione?

62.1 Applicazione e altri layer

Servizi, entità, interattori, dominio, ecc.: come si relazionano?

CAPITOLO 63

Cosa va nei [port]?

63.1 Oggetti di trasferimento dati

63.2 I [port] non solo un layer

CAPITOLO 64

Parte 5: TDD a livello di architettura applicativa

CAPITOLO 65

Progettazione del layer di automazione

65.1 Adattare il pattern screenplay

codificare in termini di intenzioni (quando si sa di più sull'intenzione) eseguire il refactoring dell'API specifica del dominio (quando si sa di più sulla tecnologia sottostante)

65.2 Driver

riutilizzo della "composition root"

65.2.1 Metodo start separato

65.2.2 Adapter fake

Includono impostazioni e asserzioni [port-specific].

Crea un nuovo adapter fake per ogni chiamata.

65.2.3 Uso dei fake

Per i thread e per. es. i database - oggetti più semplici con comportamento parzialmente definito

65.3 Attori

Dove vanno a finire le asserzioni? negli attori o nel contesto?

Come gestire il contesto per-attore (ad esempio, ogni attore ha i propri messaggi inviati e ricevuti archiviati)

Questi non sono attori come nel modello dell'attore

65.4 Builder di dati

builder nidificati, builder come oggetti immutabili.

CAPITOLO 66

Ulteriori Letture

66.1 Motivazione: il primo passo per imparare il TDD

- Fearless Change: Patterns for Introducing New Ideas by Mary Lynn Manns Ph.D. e Linda Rising Ph.D. vale la pena darci uno sguardo.
- Resistance Is Not to Change di Al Shalloway (vedere anche qui)

66.2 I Tool Essenziali

- Gerard Meszaros ha scritto un lungo libro sull'utilizzo della famiglia di framework di test XUnit, chiamato XUnit Test Patterns. Questo libro spiega anche molta della filosofia dietro questi strumenti.

66.3 Gli *Oggetti Valore*

- Ken Pugh ha un capitolo dedicato agli *oggetti valore* nel suo libro *Prefactoring* (il nome del capitolo è *Abstract Data Types*).
- *Growing Object Oriented Software Guided By Tests* contiene alcuni esempi di utilizzo di oggetti di valore e alcune strategie di refactoring verso di essi.
- Discussione sugli *oggetti valore* sul wiki C2.
- Il bliki di Martin Fowler menziona gli *oggetti valore*. Sono anche uno dei modelli nel suo libro Patterns of Enterprise Application Architecture
- Arlo Beshele descrive come usa gli *oggetti valore* (descritti sotto il nome di *Whole Value*) molto più di quanto faccio io in questo libro, presentando un stile di design alternativo più vicino al funzionale di quello di cui scrivo.
- Il libro Implementation Patterns di Kent Beck include l'*oggetto valore* come uno dei pattern.