Clean Code

Robert C. Martin

Jacopo De Angelis

1 settembre 2020

Indice

Capitolo 1

Clean code

Il codice non finirà con l'era dell'autogenerazione da IA. Qualcuno dovrà creare le IA, qualcuno dovrà imparare come dare le specifiche. Il codice sarà sempre presente.

1.1 Pessimo codice

Una delle prime cause del pessimo codice è la fretta dettata dall'ansia. L'idea di dover far uscire il codice il prima possibile ci porta a commettere errori, commettere inesattezze. Quello è ciò che può portare a seri problemi successivamente, il rileggere il proprio codice scritto in maniere quantomeno esecrabili è una tortura. E ricordiamo che se si pensa "lo metto a posto dopo", dopo equivale a mai.

I rallentamenti derivanti da nuovo codice di bassa qualità sono esponenziali, lentamente la produttività crolla perchè operare sul codice precedente è sempre più complicato.

Il che può portare ad un desiderio di ricreare da zero l'intera base del codice, cosa non solo dispendiosa ma che richiede anche molto tempo. I team si trovano a lavorare in parallelo, il nuovo team che ricrea tutto e integra il nuovo lavoro del vecchio team e, alla fine, ci si troverà nella stessa situazione.

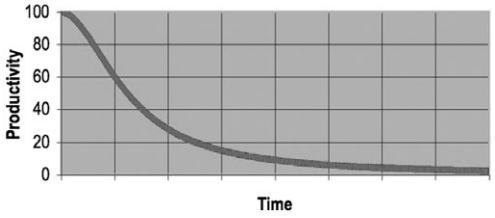

Figura 1.1: Produttività vs tempo

1.2 Scrivere buon codice

Scrivere buon codice richiede disciplina nell'uso di molte piccole tecniche applicate in ogni singolo momento. Tutto questo richiede anche una parte di "senso estetico", un percepire il perchè un bel codice sia, appunto, bello.

Una regola che possiamo ereditare dai boy scout: lascia il campo più pulito di come l'hai trovato. Ad esempio: una variabile può avere un nome più autoesplicativo? Cambiala. Una funzione può essere spezzata in più funzioni elementari? Dividila.

Capitolo 2

Nomi significativi

2.1 Usare nomi che rivelino l'intenzione

Trovare nomi significativi non è semplice ma il tempo che prendono nel farlo è sicuramente meno di quello speso a decifrare nomi non chiari.

Ogni nome, che sia di variabile o di funzione, deve rispondere alle domande:

Cosa fa

Perchè esiste

Come viene utilizzata

Se un nome richiede un commento allora il nome è sbagliato.

Ad esempio

```
int d; // elapsed time in days
```

d non dice molto come nome. Dovremmo scegliere un nome migliore, ad esempio

```
int elapsedTimeInDays;
```

```
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Scegliere un nome che rivela un intento rende molto più semplice cambiare e comprendere un codice. Ad esempio cosa fa questo pezzo di codice?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Il problema di questo codice non è la sua semplicità ma la sua capacità di avere un senso implicito. Ad esempio, le domande che ci possiamo porre sono:

Cos'è theList?

Che significato ha l'elemento 0 di theList?

Qual è il significato di 4?

Come viene usata la lista ritornata?

Queste risposte dovrebbero essere nel codice. Immaginiamo ora di lavorare a campo minato. Rinominiamo la lista con gameBoard.

Ogni cella della board è rappresentata da un array, il valore alla posizione 0 è la posizione dello status della cella e se è 4 vuol dire "segnata". Già dando implicitamente queste notazioni possiamo migliorare il codice:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
```

```
        flaggedCells.add(cell);
    return flaggedCells;
}
```

Possiamo andare anche oltre e scrivere una semplice classe per le celle invece di avere degli int. Può includere una funzione con un nome che ne sveli l'intento per nascondere questo numero. Il risultato è:

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

2.2 Evitare la disinformazione

Mai usare parole che non descrivono la realtà, ad esempio usare accountList solo se effettivamente ci troviamo davanti ad una List.

Non usare nomi che variano tra di loro per dei piccoli dettagli, ad esempio XYZControllerForEfficientHandlingOfStrings e XYZControllerForEfficientStorageOfStrings

Avere una naming convention consistente è essenziale. Un pessimo esempio di uso è quello di o ed l minuscoli, pericolosamente simili a 0 e 1.

2.3 Fare distinzioni significative

Evitare nomi con degli errori di battitura intenzionali perchè si vogliono nominare due variabili allo stesso modo. Evitare anche nomi che non danno informazioni, ad esempio:

```
public static void copyChars(char a1[], char a2[]) {
```

```
  for (int i = 0; i < a1.length; i++) {
  a2[i] = a1[i];
  }
}
```

Evitare nomi ”che creano rumore”, ad esempio ProductInfo e ProductData si differenziano per la seconda parola ma comunque non sappiamo cosa facciano.

Il tipo di entità non dovrebbe mai essere contenuto nel nome. NameString non ha senso, non ci chiederemmo mai se un semplice Name possa essere un float, questo perchè il nome stesso ci informa del suo contenuto.

Un esempio di confusione è:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Come potremmo mai sapere quale funzione chiamare dal suo nome?

Distinguere sempre i nomi in modo da intuire immediatamente la loro funzione leggendoli.

2.4 Usare nomi pronunciabili

Devi poterlo pronunciare. Hai mai provato a discutere della funzione della classe Genymdhms (generation date, year, month, day, hour, minute, and second)? Spero di no.

Immaginiamo comparare questa classe

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = ”102”;
  /* ... */
};
```

con questa

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

2.5 Usare nomi ricercabili

Le variabili con nome di una singola lettera vanno bene solo come variabili locali di un metodo, mai in altro modo, sarebbe impossibile cercarle altrimenti.

Compariamo

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

con

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays /
      ↪ WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

2.6 Prefissi

I prefissi erano utili anni fa, ormai sono abbastanza inutili.

2.7 Interfacce e implementazione

Ci sono due scuole di pensiero:

1. Interfaccia che inizia con I, implementazione senza decorazioni (IClasse, Classe)

2. Interfaccia senza decorazioni, implementazione con suffisso Imp (Classe, ClasseImp)

É uguale.

2.8 Evitare il mapping mentale

Il significato dei nomi non deve essere chiaro solo a chi scrive ma anche a chi legge.

2.9 Nomi delle classi

Le classi dovrebbero essere nomi o frasi di sostantivi, evitare Manager, Processor, Data o Info. Una classe non dovrebbe essere un verbo

2.10 Nomi dei metodi

I nomi dovrebbero contenere un verbo che descrivere cosa fanno, ad esempio postPayment, deletePage o save.

Accessori, mutatori e predicati dovrebbero iniziare con get, set e is.

2.11 Non usare nomignoli

Non chiamare variabili, metodi e classi con nomi che utilizzano inside joke, riferimenti culturali o battute. Chiamare la funzione delete() holyHandGranade non è simpatico, è un inferno.

2.12 Una parola per concetto

Scegli una parola che esprima un concetto e mantienila. Ad esempio scegli tra fetch, retrieve e get e poi usa solo quella, non alternare tra le tre.

2.13 Usare nomi del dominio della soluzione

Meglio usare nomi che hanno un significato speciale nel caso sia quello il caso. Ad esempio, dire AccountVisitor vuol dire molto se si è a conoscenza del pattern visitor. I nomi che usano un lessico tecnico sono comodi.

2.14 Usare nomi del dominio del problema

Nel caso non ci siano nomi immediati da utilizzare facenti parte del dominio della soluzione, allora possiamo iniziare a guardare il dominio del problema.

2.15 Aggiungere un contesto significativo

Solitamente è meglio avere nomi che si spieghino da soli, in certi casi, però, ciò è difficile. Prendiamo ad esempio firstName, lastName, street, houseNumber, city, state e zipcode. Assieme sappiamo che sono un indirizzo ma presi singolarmente? Se vedessimo state usato da un'altra parte? In questo caso, allora,

può essere accettabile usare, ad esempio addrFirstName, addrLastName, addrStreet, addrHouseNumber, addrCity, addrState e addrZipcode.

Non bisogna aggiungere del contesto a caso però, ad esempio dando a tutte le variabili un prefisso che descriva l'app.

Capitolo 3

Funzioni

La maggior parte delle regole di scrittura per le funzioni sono le stesse descritte dalle buone norme del refactoring.

3.1 Corte

Le funzioni dovrebbero essere lunghe massimo intorno alle 20 righe, nulla vieta di riuscire a ridurle ulteriormente però. Se è di più possiamo chiederci "è possibile estrarre una parte della funzione"?

Ad esempio la funzione

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
      WikiPage testPage = pageData.getWikiPage();
      StringBuffer newPageContent = new StringBuffer();
      includeSetupPages(testPage, newPageContent, isSuite);
      newPageContent.append(pageData.getContent());
      includeTeardownPages(testPage, newPageContent,
          ↪ isSuite);
      pageData.setContent(newPageContent.toString());
  }
```

```
  return pageData.getHtml();
}
```

è riscrivibile come

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

3.1.1 Blocchi e indentazione

Se possibile sarebbe meglio ridurre i blocchi di indentazione a una sola linea, come abbiamo visto nel codice precedente, e possibilmente che chiami un'altra funzione per svolgere le sue funzioni.

Tutto ciò rende il codice snello e leggibile

3.2 Fare una sola cosa

Un metodo non deve fare tutto e male ma solo una cosa e farla bene. Un metodo che si occupa di una sequenza di passi avrà richiami a metodi che eseguono le subroutine ma non avrà altra logica al suo interno.

Se in una funzione vediamo più sezioni come dichiarazione, inizializzazione e scrematura come possiamo vedere qua

```
/**
* This class Generates prime numbers up to a user specified
* maximum. The algorithm used is the Sieve of Eratosthenes.
* <p>
* Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
* d. c. 194, Alexandria. The first man to calculate the
* circumference of the Earth. Also known for working on
* calendars with leap years and ran the library at Alexandria.
```

```
* <p>
* The algorithm is quite simple. Given an array of integers
* starting at 2. Cross out all multiples of 2. Find the next
* uncrossed integer, and cross out all of its multiples.
* Repeat untilyou have passed the square root of the maximum
* value.
*
* @author Alphonse
* @version 13 Feb 2002 atp
*/
import java.util.*;
public class GeneratePrimes
{
  /**
  * @param maxValue is the generation limit.
  */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
      f[i] = true;
      // get rid of known non-primes
      f[0] = f[1] = false;
      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
      }
      // how many primes are there?
```

```
        int count = 0;
        for (i = 0; i < s; i++)
        {
          if (f[i])
            count++; // bump count.
        }
        int[] primes = new int[count];
        // move the primes into the result
        for (i = 0, j = 0; i < s; i++)
        {
          if (f[i]) // if prime
            primes[j++] = i;
        }
        return primes; // return the primes
        }
      else // maxValue < 2
      return new int[0]; // return null array if bad input.
    }
}
```

vuol dire che probabilmente possiamo scomporla ulteriormente.

3.3 Un livello di astrazione per funzione

Ci si deve assicurare che una funzione sia ad un solo livello di astrazione. Non dovremmo mai avere funzioni ad alto livello di astrazione mischiate a codice a basso livello di astrazione, crea solo confusione e non rispetta il principio di responsabilità.

3.3.1 Leggere il codice dall'alto verso il basso

Primo avviso: questo si applica solo a linguaggi compilati e a linguaggi che si occupano dell'analisi di tutto il codice prima dell'esecuzione. In un linguaggio interpretato questa regola non si applica in quanto le funzioni devono essere descritte bottom-top.

Come regola di base dovremmo poter leggere il codice come una narrativa, dalla funzione principale a quelle ausiliarie, scendendo sempre più tra i vari livelli di astrazione.

3.4 Controlli di flusso

É difficile mantenere brevi gli switch e gli if/else. Possiamo però separare lo switch in una classe di basso livello e non vederlo mai ripetuto.

Consideriamo questo codice

```
public Money calculatePay(Employee e)
  throws InvalidEmployeeType {
  switch (e.type) {
    case COMMISSIONED:
      return calculateCommissionedPay(e);
    case HOURLY:
      return calculateHourlyPay(e);
    case SALARIED:
      return calculateSalariedPay(e);
    default:
      throw new InvalidEmployeeType(e.type);
  }
}
```

1. è già grande e crescerà ancora di più all'aumentare delle tipologie di dipendenti
2. fa più d una cosa
3. viola il principio di singola responsabilità
4. viola il principio open close[1] perchè deve essere cambiata per ogni modifica nel dataset

La soluzione a questo problema, ad esempio, è una abstract factory. La factory passerà l'istanza dei dipendenti e i vari metodi sono creati polimorficamente usando le interfacce.

[1] Le entità dovrebbero essere aperte per l'estensione, chiuse per le modifiche

Una regola possibile è che gli switch:

1. devono compararire solo una volta
2. devono sfruttare il polimorfismo
3. devono essere nascosti tramite ereditarietà

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws
      ↪ InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements
    ↪ EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws
      ↪ InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

3.5 Usare nomi descrittivi

Così come per le variabili, i nomi devono essere significativi. Meglio un nome lungo rispetto ad un nome che non spiega ciò che accade nel metodo.

Importante è avere una naming convention, anche non scritta, in modo che i nomi siano consistenti e che siano facilmente interpretabili.

3.6 Argomenti delle funzioni

Il numero ideale di argomenti per le funzioni è 0. Uno va bene, due accettabile, tre già è strano, quattro o più richiede una giustificazione seria.

3.6.1 Forma comune per singolo argomento

Ci sono due ragioni per passare un argomento:

- stai ponendo delle domande su di esso
- stai operando su di esso

Un altra ragione è quella di generazione di un evento: il metodo ha in ingresso un argomento ma in uscita nessuno.

3.6.2 Argomenti di guardia

Sono brutti. Passare un boolean in una funzione come guardia per dire con che modalità eseguire la funzione è brutto a vedersi. Sarebbe meglio dividere la funzione in più metodi.

3.6.3 Funzioni con due argomenti

Una diade non per forza è negativa, ad esempio quando si crea un punto è necessario passare due variabili per gli assi x e y e ciò è giusto. Il problema è quando ci troviamo davanti ad altre forme come, ad esempio, assertEquals(expected, actual). Quale viene prima, quale dopo? Serve pratica perchè non c'è un ordine naturale.

Le soluzioni sono svariate, ad esempio estrarre un campo e renderlo appartenente alla classe.

3.6.4 Oggetti come argomenti

Spesso se vengono passati due o tre argomenti, questi saranno legati in qualche modo, probabilmente in un oggetto. In quel caso è meglio pensare di passare l'oggetto direttamente. Ad esempio qua è visibile la differenza di lettura dei due metodi.

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

3.6.5 Lista come argomento

L'utilizzo delle liste opzionali di argomenti è molto comodo e in certi casi aiuta a tenere pulito il codice. Ad esempio

```
public String format(String format, Object... args)
```

A tutti gli effetti ha due argomenti ma a runtime può essere usato con infiniti argomenti.

3.6.6 Verbi e parole chiave

Usare una combinazione di verbi per le funzioni e sostantivi per gli argomenti può rendere le funzioni molto evocative. Ad esempio writeField(name) subito fa capire che questo nome verrà scritto.

Codificare le variabili nel nome del metodo è anche un metodo interessante per renderla autodescrittiva. Tornando all'assertEquals(expected, actual) di prima, quando sarebbe più immediato e meno confusa la sequenza se si chiamasse assertExpectedEqualsActual(expected, actual)?

3.7 Non devono avere effetti collaterali

Gli effetti collaterali sono bugie. Una funzione dovrebbe fare una e una sola cosa, nascondere un effetto collaterale, ovvero l'agire su di una variabile esterna al suo scope, è un modo di farle fare più cose.

Ad esempio:

```
public class UserValidator {
  private Cryptographer cryptographer;
  public boolean checkPassword(String userName, String
      ↪ password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase =
          ↪ user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase,
          ↪ password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Qua evidentemente inizializza la sessione quando la funzione dovrebbe solo validare l'utente. Questo va contro il principio di singola resposabilità

3.8 Output

Certi nomi possono confondere, portando a chiedersi se stiano parlando dell'input o dell'output. Ad esempio appendFooter(s) attacca s ad un footer o attacca un qualche footer ad s? Solo guardando la firma del metodo si scopre che

```
public void appendFooter(StringBuffer report)
```

effettivamente attacca un footer al buffer passato.

Quello che abbiamo appena fatto è un controllo secondario, un qualcosa che può interrompere il flusso di programmazione.

In generale gli output dovrebbero essere evitati, se possibile è meglio far agire le funzioni sull'oggetto che le possiede.

3.9 Separazione dei comandi

Una funzione dovrebbe fare qualcosa o rispondere a qualcosa, non entrambe.

3.10 Preferire le eccezioni al ritornare direttamente messaggi di errore

Passare errori direttamente porta al doverli gestire subito, invece passare un'eccezione ha due benefici principali:

- sono rapidi da usare e non si confonde ciò che può essere passato
- possono essere messi in calce al percorso di esecuzione

3.10.1 Estrarre i blocchi try/catch

Con ciò si vuole dire di non scrivere l'interno del blocco catch con tutti i suoi passaggi ma di portarlo fuori come metodo in modo da avere una struttura molto più compatta, leggibile e gestibile. Ricordiamo che le eccezioni vanno gestite il più vicino possibile alla fonte ma non per questo non possiamo mandarle alla funzione chiamante.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}
private void deletePageAndAllReferences(Page page) throws
    ↪ Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

3.10.2 La gestione dell'errore è una sola cosa

Le funzioni dovrebbero fare una sola cosa + la gestione dell'errore è una sola cosa = Una funzione che gestisce l'errore non dovrebbe fare altro. Ciò vuol dire che se in una funzione eiste la parola try dovrebbe essere la prima e niente dopo i blocchi catch e finally.

3.10.3 Error.java e la dipendenza da esso

Molti scrivono i messaggi di errore in una enumerazione da importare in ogni classe che la sfrutta. Risultato? Dipendenze ovunque ed essere costretti a modificare codice e ricompilare ogni volta.

Creare eccezioni figlie della classe Error invece rende il codice meno codipendente e più manutenibile.

3.10.4 Non ripetersi

Mai ripetere funzioni, piuttosto meglio importarle ma scrivere più volte lo stesso codice porta a dover debuggare più volte e c'è il rischio di riparare da una parte e non dalle altre.

3.11 Programmazione strutturata

Molti programmatori seguono il principio di Dijkstra: ogni funzione e ogni blocco deve avere una sola entrata e una sola uscita. Ciò vuol dire:

- un solo return
- niente break o continue
- mai un goto

Questa regola perde di valore quando le funzioni sono molto brevi, questo perchè una sovraingegnerizzazione in un piccolo blocco di codice rischia di oscurare il vero significato dietro al metodo.

3.12 Come si scrivono funzioni così?

Ricorda: primo passaggio è la stesura, il successivo la pulizia. Va bene scrivere codice leggibile solo da te all'inizio, non devi

lasciarlo così poi. Poco alla volta puoi pulirlo, renderlo perfetto.

3.13 Esempio finale

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;
public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;
  public static String render(PageData pageData) throws
      ↪ Exception {
    return render(pageData, false);
  }
  public static String render(PageData pageData, boolean
      ↪ isSuite)
  throws Exception {
    return new
        ↪ SetupTeardownIncluder(pageData).render(isSuite);
  }
  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }
  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
    includeSetupAndTeardownPages();
    return pageData.getHtml();
  }
  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }
```

```
private void includeSetupAndTeardownPages() throws
    ↪ Exception {
  includeSetupPages();
  includePageContent();
  includeTeardownPages();
  updatePageContent();
}
private void includeSetupPages() throws Exception {
  if (isSuite)
  includeSuiteSetupPage();
  includeSetupPage();
}
private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME,
      ↪ "-setup");
}
private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}
private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}
private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
  includeSuiteTeardownPage();
}
private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}
private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME,
      ↪ "-teardown");
}
private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}
private void include(String pageName, String arg) throws
    ↪ Exception {
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
```

```
        String pagePathName =
            ↪ getPathNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);
      }
    }
    private WikiPage findInheritedPage(String pageName)
        ↪ throws Exception {
      return PageCrawlerImpl.getInheritedPage(pageName,
          ↪ testPage);
    }
    private String getPathNameForPage(WikiPage page) throws
        ↪ Exception {
      WikiPagePath pagePath = pageCrawler.getFullPath(page);
      return PathParser.render(pagePath);
    }
    private void buildIncludeDirective(String pagePathName,
        ↪ String arg) {
      newPageContent
      .append("\n!include ")
      .append(arg)
      .append(" .")
      .append(pagePathName)
      .append("\n");
    }
}
```

Capitolo 4

Commenti

I commenti possono essere molto utili come anche superflui. Se imparassimo a scrivere codice comprensibile non ci servirebbe minimamente scrivere commenti. Spesso vengono inseriti per sopperire ad una mancanza del linguaggio.

Attenzione: commenti e documentazione non sono per niente la stessa cosa.

Tutto ciò vuol dire che nel momento nel quale si sente il bisogno di scrivere un commento bisogna chiedersi: posso scrivere il codice in modo che non serva?

Uno dei motivi principali di questa pratica è che il codice cambia, il commento spesso no.

I commenti non correggono del pessimo codice!

4.1 Spiegati nel codice

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Questo codice ha un commento perchè non è chiaro cosa faccia. É molto più semplice racchiudere la logica in un metodo a parte il cui nome spieghi cosa accade e poi usarlo, come ad esempio

```
if (employee.isEligibleForFullBenefits())
```

Ora è molto più chiaro e prende anche meno spazio

4.2 Buoni commenti

Alcuni commenti sono utili.

4.2.1 Commenti legali

Non è insolito che le aziende chiedano di inserire all'interno del codice un header con dei commenti riguardanti il copyright.

Questi commenti non dovrebbero essere contratti o lunghi quanto un libro di diritto. Dovrebbero rimandare alla licenza di riferimento, salvata da un'altra parte.

4.2.2 Commenti informativi

In certi casi è utile dare delle informazioni di base, per esempio la spiegazione di cosa ritorni un metodo

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Il problema è che certe informazioni andrebbero date, come sempre, tramite il nome della funzione. Ecco un caso leggermente migliore

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

In questo caso il commento serve per dare la formattazione in maniera immediata

4.2.3 Spiegazione d'intento

In certi casi i commenti servono a spiegare quale fosse lo scopo di certe scelte di codice, ad esempio

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
  String text = """bold text""";
  ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), """bold text""");
  AtomicBoolean failFlag = new AtomicBoolean();
  failFlag.set(false);

  //This is our best attempt to get a race condition
  //by creating large number of threads.
  for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
      new WidgetBuilderThread(widgetBuilder, text, parent,
          ↪ failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
  }
  assertEquals(false, failFlag.get());
}
```

In questo modo lo sviluppatore rende nota la propria decisione e ciò che voleva raggiungere.

4.2.4 Chiarimenti

In certi casi è possibile scrivere commenti che spieghino al lettore una parte di codice

```
public void testCompareTo() throws Exception
{
```

```
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");
  assertTrue(a.compareTo(a) == 0); // a == a
  assertTrue(a.compareTo(b) != 0); // a != b
  assertTrue(ab.compareTo(ab) == 0); // ab == ab
  assertTrue(a.compareTo(b) == -1); // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1); // b > a
  assertTrue(ab.compareTo(aa) == 1); // ab > aa
  assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Chiaramente i commenti potrebbero essere sbagliati, per questo vanno presi con attenzione.

4.2.5 Avvisare delle conseguenze

In certi casi è meglio avvisare cosa accade usando certe feature, ad esempio

```
public static SimpleDateFormat
    ↪ makeStandardHttpDateFormat()
{
  //SimpleDateFormat is not thread safe,
  //so we need to create each instance independently.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd
      ↪ MMM yyyy HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

4.2.6 TODO

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```

I todo sono comodi anche per capire perchè non bisogna fucilare il creatore di un metodo del genere. Sono un promemoria per noi e un avviso per gli altri.

4.2.7 Aplificazione

I commenti possono essere usati per esprimere l'importanza di parti non triviali o la cui importanza non è immediata.

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

4.2.8 Javadoc o simili

Documentare in maniera corretta, utile e sufficiente tutto tramite le funzioni di documentazione è cosa buona e giusta. Ovviamente anche queste devono essere scritte bene ma possono contenere errori certe volte.

4.3 Pessimi commenti

4.3.1 Ragionamenti generici

```
public void loadProperties()
  {
  try
  {
    String propertiesPath = propertiesLocation + "/" +
        ↪ PROPERTIES_FILE;
    FileInputStream propertiesStream = new
        ↪ FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

Ad esempio qua la fretta ha fatto sì che l'autore lasciasse un commento non chiaro. Chi è che carica i default? Quali sono i default? cc.

4.3.2 Commenti ridondanti

```
// Utility method that returns when this.closed is true.
    ↪ Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
  throws Exception
  {
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not
          ↪ be closed");
  }
}
```

Ad esempio qua il commento è già chiaro da ciò che c'è scritto nel metodo. Spesso un commento ridondante deriva dall'aver già scritto codice chiaro ma volerlo comunque commentare per paura.

4.3.3 Commenti fuorvianti

In certi casi, per puro errore umano, nonostante le buone intenzioni, si possono lasciare commenti fuorvianti. Anche un piccolo errore nella spiegazione del flusso dei dati può creare gravi problemi.

4.3.4 Commenti obbligatori

É ridicolo avere una regola che prescriva di scrivere per ogni funzione un javadoc o simili.

4.3.5 Commenti giornali

In certi casi gli sviluppatori scrivono un giornale delle modifiche con data e cos'è stato fatto. É inutile, non aggiunge informazioni e diventa ancora più inutile coi sistemi di versionamento.

4.3.6 Commenti di rumore

Sono i commenti che non hanno nemmeno uno scopo, occupano spazio e basta.

4.3.7 Rumore spaventoso

C'è di peggio, ci sono i commenti che fanno ciò che devono ma che sono completamente inutili e poi ci sono quelli anche sbagliati.

4.3.8 Indicatori per le parti

Un commento tipo

```
// Actions ////////////////////////////////
```

per indicare un blocco di codice è utile ad una prima vista ma in realtà se il codice è ben compartimentato non c'è da preoccuparsi per qualcosa del genere.

4.3.9 Commenti per chiudere i blocchi

```
try {
  while ((line = in.readLine()) != null) {
    lineCount++;
    charCount += line.length();
    String words[] = line.split("\\W");
    wordCount += words.length;
  } //while
  System.out.println("wordCount = " + wordCount);
  System.out.println("lineCount = " + lineCount);
  System.out.println("charCount = " + charCount);
} // try
catch (IOException e) {
  System.err.println("Error:" + e.getMessage());
} //catch
```

Questo modo di chiudere i blocchi per rendere più evidente a cosa si riferiscano è diventato completamente inutile grazie alle IDE. In più se un blocco è così vasto da non poterne riconoscere la fine, forse è meglio applicare un po' di refactoring.

4.3.10 Attribuzione

Scrivere in un commento l'autore di una parte di codice è inutile ed è reso semplice dal versionamento

4.3.11 Codice rimosso tramite commento

Eliminare del codice commentandolo, quindi in realtà lasciandolo in bella mostra, è rischioso, crea spreco di spazio e memoria e si accumula come pochi. Il versionamento aiuta a ricordare il codice cancellato, è inutile questa prassi.

4.3.12 Commenti HTML

```
/**
* Task to run fit tests.
* This task runs fitnesse tests and publishes the results.
* <p/>
* <pre>
* Usage:
* &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
*
    ↪ classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&
* classpathref=&quot;classpath&quot; /&gt;
* OR
* &lt;taskdef classpathref=&quot;classpath&quot;
* resource=&quot;tasks.properties&quot; /&gt;
* <p/>
* &lt;execute-fitnesse-tests
* suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
* fitnesseport=&quot;8082&quot;
* resultsdir=&quot;${results.dir}&quot;
* resultshtmlpage=&quot;fit-results.html&quot;
* classpathref=&quot;classpath&quot; /&gt;
* </pre>
*/
```

No.

4.3.13 Informazione non locale

Un commento è buono se si riferisce alla sua prossimità, fare riferimento a codice da altre parti è sbagliato.

4.3.14 Troppe informazioni

Non essere logorroico.

4.3.15 Connessioni non ovvie

```
/*
* start with an array that is big enough to hold all the pixels
* (plus filter bytes), and an extra 200 bytes for header info
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3)
    ↪ + 200];
```

Ad esempio qua cos'è un byte filtro? Si collega al +1 o al *3?

Se un commento richiede una spiegazione allora è un pessimo commento

4.3.16 Javadocs in codice non pubblico

Se il codice deve essere acceduto dall'esterno allora va bene che sia commentato, altrimenti è superfluo.

Capitolo 5

Formattazione

5.1 Formattazione verticale

Il livello di dettaglio di una classe dovrebbe aumentare discendendo.

- costanti
- variabili
- costruttori
- metodi pubblici
- metodi privati
- getter e setter

5.1.1 Separazione delle parti

Gli spazi tra i blocchi di codice sono utili per discriminare le parti del codice. L'apertura verticale separa i concetti.

5.1.2 Associazione delle parti

La densità verticale implica associazione, quindi le linee che verticalmente sono dense esprimono concetti legati

Se una parte ne chiama un'altra allora dovrebbero essere verticalmente vicine.

5.2 Formattazione orizzontale

Una volta il limite era 80 ma con gli schermi odierni il limite può essere alzato anche a 120. La regola da seguire, idealmente, è che a font standard non si debba mai scorrere a in orizzontale sullo schermo.

5.2.1 Apertura orizzontale e densità

Tra operatori e parti lo spazio serve, nella dichiarazione o nella chiamata di una funzione lo spazio tra nome della funzione e parentesi no, questo perchè sono strettamente legate.

5.2.2 Allineamento orizzontale

L'indentazione serve per rendere evidenti i blocchi di codice, aiutando così nella loro identificazione.

5.3 Regole del gruppo

Tutti abbiamo delle regole preferite ma se si lavora in gruppo allora si deve concordare su di uno stile unificato. Queste regole devono essere seguite e documentate.

Capitolo 6

Oggetti e strutture dati

6.1 Astrazione

Guardiamo i due listati seguenti. Entrambi rappresentano un punto del piano cartesiano, uno espone completamente la sua implementazione, l'altro lo nasconde.

```
public class Point {
  public double x;
  public double y;
}
```

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

La seconda classe nasconde come vengono salvate le sue variabili e, soprattutto, crea delle regole d'accesso ai dati.

6.2 Asimmetria dati/oggetti

La differenza è:

- gli oggetti nascondono i dati ed espone funzioni
- le strutture dati espongono i dati e non hanno funzioni significative

6.3 Legge di Demetra

- ogni unità di programma dovrebbe conoscere solo poche altre unità di programma strettamente correlate
- ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente

Ovvero, data una classe C con un metodo f, questo metodo dovrebbe chiamare solo:

- C
- Un oggetto creato da f
- Un oggetto passato come argomento ad f
- Un oggetto in un'istanza di C

6.3.1 Incidenti ferroviari

```
final String outputDir =
    ↪ ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Questo tipo di codice è chiamato incidente ferroviario perchè sembrano due treni che si sono scontrati e si sono accartocciati fino a diventare una cosa sola.

Questo stile di programmazione è da evitare, sarebbe meglio dividere le chiamate così

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Se questo insieme viola la legge di Demetra dipende se ctxt, Options e ScratchDir sono oggetti o strutture dati. Se sono oggetti, le loro strutture interne dovrebbero essere nascoste e quindi la conoscenza dell'interno è chiaramente una violazione della legge. Se sono strutture dati senza comportamenti, allora esporranno naturalmente le loro strutture interne, e quindi la legge non si applica.

6.3.2 Ibridi

Certe volte vengono create classi che sono anche strutture dati, presentano varaibili pubbliche e private, accessori di ogni tipo e funzioni significative. Sono da evitare, è un caso di "feature envy", ovvero metodi che chiamano funzioni o attributi di altre classi più della propria.

6.3.3 Nascondere le strutture

Se comunichiamo con un oggetto dovremmo dirgli di fare qualcosa, non dovremmo chiedere i suoi dettagli interni.

Ad esempio, nel caso delle chiamate di prima, ottenevamo il percorso assoluto di un file, un errore gigantesco. In più cosa dobbiamo farci? Se volessimo creare un nuovo file non sarebbe meglio fare così?

```
BufferedOutputStream bos =
    ↪ ctxt.createScratchFileStream(classFileName);
```

In questo modo i dettagli sull'implementazione sarebbero nascosti.

6.4 Data Transfer Object (DTO)

un DTO è una struttura dati molto utile, specialmente quando si comunica col database o si devono elaborare dei dati prima di presentarli. I loro campi sono pubblici e utilizzabili.

Sono comuni anche i ”bean”, oggetti con variabili private ma con getter e setter.

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
    String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
    }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }
```

```
  public String getZip() {
    return zip;
  }
}
```

6.4.1 Active records

Sono tipi speciali di DTO. Hanno strutture dati con variabili pubbliche o accessibili ma solitamente hanno metodi come salva e trova. Spesso sono traduzioni dirette del database.

Capitolo 7

Gestione errori

La gestione errori è importante ma se offusca la logica allora c'è un problema.

7.1 Usare le eccezioni invece del return

Una volta, quando non c'era una gestione delle eccezioni come ora, si usavano dei flag per segnalare gli errori. Al giorno d'oggi possono essere sollevate eccezioni che rendono la vita molto più semplice e, soprattutto, possono rendere il codice più comprensibile.

7.2 Scrivere il blocco Try-Catch-Finally prima di tutto

Il blocco catch deve permettere l'uscita dal metodo in uno stato consistente, indipendentemente da cosa sia successo nel try.

7.3 Usare eccezioni non controllate

Le eccezioni controllate violano il principio Open/closed perchè creano più uscite. Se si solleva un'eccezione e viene mandata tre livelli sopra allora si dovrà inserire nell'intestazione di tutti i metodi della catena.

7.4 Offrire un contesto con le eccezioni

Le eccezioni dovrebbero offrire abbastanza informazioni da consentire di comprendere cosa le abbia scatenate. Lo stacktrace è utile ma non è l'unica fonte, anche un messaggio significativo è ottimo.

7.5 Definire le classi di eccezioni in termini dei bisogni del chiamante

Ci sono molti modi per classificare gli errori. Possiamo classificarli dalla loro fonte o per il loro tipo. La cosa, però, realmente più importante in questo contesto è "come sono catturati"?

Ora vediamo un esempio di pessima classificazione in una libreria:

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
reportPortError(e);
  logger.log("Device response exception");
```

```
} finally {…

}
```

Questo blocco di codice contiene molte ripetizioni. In questo caso possiamo racchiudere le eccezioni in un tipo generico che però contenga una descrizione dell'errore.

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {…

}

public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }…

}
```

Rinchiudere in un wrapper le eccezioni derivanti da una libreria

di terze parti può essere molto comodo. In più permette di non dover dipendere dalle scelte di design di qualcun altro.

7.6 Non ritornare null

Mai e poi mai ritornare null. Rendere un programma null safe è essenziale per evitare la maggior parte degli errori.

Usare un optional o, ad esempio, una emptyList, sono metodi per poter permettere al flow del programma di non incontrare una NullPointerException.

Capitolo 8

Limiti

8.1 Usare codice di terze parti

Speso nella scrittura del codice da poter implementare da terze parti si pensa all'uso più generico possibile che può essere fatto mentre l'utente ha bisogno di un uso specifico. In cosa si traduce questo? Nel dover pensare a come rendere specifico questo codice per evitare problemi. Ad esempio

```
Map sensors = new HashMap();
Sensor s = (Sensor)sensors.get(sensorId );
```

Il codice non è perfettamente leggibile, il typecast è una pezza messa alla generalità della mappa. Potrebbe essere risolto rendendo specifica la mappa

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId );
```

Ma in questo caso non verrebbe risolto un problema essenziale: Map fornisce più funzioni di quelle che vogliamo, tra cui clear, attivabile da chiunque. Come possiamo allora nascondere questo problema implementativo? Semplice, mascherandolo in una classe apposita che si occupi di typecast, gestione degli accessi ecc.

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }
  //snip
}
```

In questo modo l'utente non deve preoccuparsi di certi dettagli implementativi, di dover stare attento a cosa venga ritornato ecc. ma invece può usare in maniera naturale la classe Sensor.

8.2 Esplorare e comprendere i confini

Quando ci si trova a implementare una libreria di terze parti si possono passare giorni a leggerne la documentazione ma non è detto che ciò che si pensa faccia la libreria e ciò che fa veramente siano la stessa cosa. Per questo motivo, prima di generare bug complicati, sarebbe meglio creare delle suite di test per testare le funzioni richieste alla libreria e vedere se il comportamento atteso è quello effettivo o no. Questo metodo, per quanto tedioso, può risparmiare molto tempo più avanti.

8.2.1 Conoscere log4j e slf4j

Per fare un esempio, uno potrebbe pensare "beh, se inizializzo il logger e dico di loggare sono a posto" e invece no

```
@Test
public void testLogCreate() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.info("hello");
}
```

Questo codice restituisce un errore in quanto il Logger ha bisogno di un appender. Allora viene aggiunto ciò

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  ConsoleAppender appender = new ConsoleAppender();
  logger.addAppender(appender);
  logger.info("hello");
}
```

Ma a quanto pare ha bisogno di un output stream, allora aggiungiamo

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.removeAllAppenders();
  logger.addAppender(new ConsoleAppender(
    new PatternLayout("%p %t %m%n"),
    ConsoleAppender.SYSTEM_OUT));
  logger.info("hello");
}
```

E ora funziona! Ora però rimuovendo l'output stream continua a funzionare ma non rimuovendo il pattern, strano. Studiando ancora la configurazione scopriamo che semplicemente il Logger non era configurato, cosa che non è altamente intuitiva. Ancora un po' di ricerca e raggiungiamo questa classe di test che rappresenta la conoscenza acquisita

```
public class LogTest {
  private Logger logger;

  @Before
  public void initialize() {
    logger = Logger.getLogger("logger");
    logger.removeAllAppenders();
    Logger.getRootLogger().removeAllAppenders();
  }

  @Test
  public void basicLogger() {
```

```
    BasicConfigurator.configure();
    logger.info("basicLogger");
  }

  @Test
  public void addAppenderWithStream() {
    logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n"),
      ConsoleAppender.SYSTEM_OUT));
    logger.info("addAppenderWithStream");
  }

  @Test
  public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n")));
    logger.info("addAppenderWithoutStream");
  }
}
```

8.3 I test di apprendimento sono ottimi

Questi test, oltre a darci un'ottima conoscenza la prima volta, ci permettono di tenere sotto controllo anche eventuali aggiornamenti. Infatti ad ogni nuova release ci basta far partire nuovamente i test per controllare che tutto sia come prima. In questo modo abbiamo anche un ottimo rientro sull'investimento.

Capitolo 9

Unit test

9.1 Le tre leggi del test driven development

1. Non scriverai codice in produzione fino a quando non avrai scritto il codice per testarlo
2. Non scriverai più di un test che debba fallire e la non compilazione è un fallimento
3. Non scriverai più codice di quanto non sia strettamente necessario per passare il test

9.2 Mantenere i test puliti

Non bisogna minimamente pensare che i test non debbano essere scritti con meno cura rispetto al codice originale.

9.3 I test abilitano la flessibilità del codice

Quando modifichiamo del codice possiamo avere paura di introdurre bug inattesi. L'avere degli unit test pronti per controllare che non vengano introdotti bug inaspettati è ciò che ci permette

di rendere il nostro codice flessibile, riutilizzabile, mantenibile, senza avere preoccupazioni aggiuntive.

9.4 Test puliti

La caratteristica più importante di un test è che sia leggibile. Ad esempio, guardiamo questi test:

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root,
      ↪ PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));
  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();
  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void
    ↪ testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
  throws Exception
{
  WikiPage pageOne = crawler.addPage(root,
      ↪ PathParser.parse("PageOne"));
  crawler.addPage(root,
      ↪ PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));
  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
```

```
  WikiPageProperty symLinks =
      ↪ properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);
  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();
  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
  crawler.addPage(root, PathParser.parse("TestPageOne"),
      ↪ "test page");
  request.setResource("TestPageOne");
  request.addInput("type", "data");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();
  assertEquals("text/xml", response.getContentType());
  assertSubString("test page", xml);
  assertSubString("<Test", xml);
}
```

Guardiamo ad esempio PathParser. Trasforma la stringa in un'istanza usata dal crawler. QUesta trasformazione è inutile per quanto riguarda il test.

Ora vediamo un miglioramento:

```
public void testGetPageHierarchyAsXml() throws Exception {
```

```
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");
  submitRequest("root", "type:pages");
  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>",
        ↪ "<name>PageTwo</name>",
        ↪ "<name>ChildOne</name>"
  );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy()
    ↪ throws Exception {
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");
  addLinkTo(page, "PageTwo", "SymPage");
  submitRequest("root", "type:pages");
  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>",
        ↪ "<name>PageTwo</name>",
        ↪ "<name>ChildOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");
  submitRequest("TestPageOne", "type:data");
  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

è molto più semplice da leggere, il pattern build-operate-check rende ovvia la struttura del test: ogni test è diviso in tre fasi, nella prima si costruiscono i dati di test, nella seconda si opera su di essi, nella terza si testano.

9.5 Doppio standard

Ci si deve sempre ricordare una cosa: l'ambiente locale e l'ambiente di produzione hanno esigenze e capacità completamente differenti. Ad esempio in produzione il codice deve essere efficiente, deve richiedere le risorse minime necessarie e deve essere rapido. Sull'ambiente locale, dove i test saranno fatti girare, non ci sono spesso questi limiti. Per questo quando scriviamo codice dobbiamo sempre chiederci "dove verrà usato?".

9.6 Un assert per test

Un test non deve testare vari stati male, ne deve testare uno bene, un po' come per le funzioni che devono effettuare solo una cosa. Il vantaggio di ciò è la leggibilità.

9.7 Un solo concetto per test

Una regola ancora migliore, da cui la precedente deriva come ovvia conseguenza, è che ogni test dovrebbe testare un solo concetto alla volta.

9.8 F.I.R.S.T.

Fast: i test dovrebbero essere veloci da eseguire, questo perchè se sono lenti non si è portati ad eseguirli spesso, il che può portare a non risolverli velocemente

Indipendenti: i test non dovrebbero dipendere dagli altri, ogni test dovrebbe essere atomico e non avere effetti collaterali per gli altri

Ripetibile: dovrebbero essere eseguibili in ogni ambiente

Auto validante: Il test dovrebbe avere un output booleano, ovvero lo passa o non lo passa

Tempestivo: i test dovrebbero essere scritti appena prima che il codice venga scritto

Capitolo 10

Classi

10.1 Organizzazione

Le classi, secondo la convenzione Java, seguono l'ordine descritto a pagina 49.

10.1.1 Incapsulamento

Le variaibli e i metodi ausiliari dovrebbero essere privati, al massimo protetti se ne abbiamo bisogno all'interno della suite di test.

10.2 Le classi dovrebbero essere piccole

Una classe con, ad esempio, 70 metodi pubblici, forse è un po' troppo, arriva a livello di God class, ovvero una classe con fin troppe responsabilità.

10.3 Principio di singola responsabilità

Ogni classe dovrebbe avere una sola responsabilità e un solo motivo per cambiare. Ad esempio

```
public class SuperDashboard extends JFrame implements
    ↪ MetaDataUser {
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

In questo caso la classe ha due ragioni per cambiare:

1. controlla le informazioni che devono essere aggiornate ogni volta che il software viene rilasciato
2. gestisce i componenti Java Swing

Questa versione è decisamente meglio

```
public class Version {
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

Questo rende le classi più indipendenti, più modellabili e permette di gestire meglio l'intero progetto.

10.3.1 Coesione e dipendenza

Per spiegare la differenza, la coesione rappresenta la forza del modulo, quanto esso possa fare e come venga effettuato. I legami (o dipendenza) è come i vari moduli interagiscono tra di loro. Idealmente noi vorremmo alta coesione e passa dipendenza.

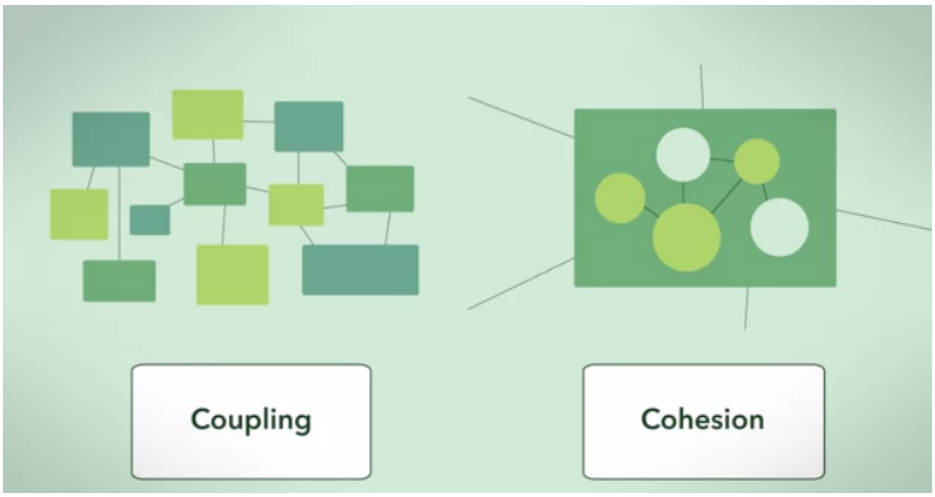

Figura 10.1: Legami e coesione

All'interno del modulo con alta coesione abbiamo un'alta dipendenza tra le classi, per mantenerla efficacemente possiamo dividere il programma in molte piccole classi che abbiano una singola responsabilità e che si richiamino tra di loro.

10.4 Organizzare per un cambiamento

Quando scriviamo una classe possiamo anche non implementarla subito perchè magari non abbiamo ancora i dettagli implementativi o perchè serve solo come placeholder per il momento. Ad esempio questa classe per la gestione SQL:

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
```

```
  private String valuesList(Object[] fields, final Column[]
      ↪ columns)
  private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

Quando ci troveremo ad implementarla, però, potremo ragionare sul suo design. Sembra una classe che effettivamente segua il principio di responsabilità all'inizio ma poi, quando guardiamo meglio, sembra che ci siano dei metodi privati che vadano in realazione solo con il metodo select, è quindi un piccolo subset di comportamenti.

In più leggere questa classe richiede tempo, è voluminosa e ha svariate funzioni. Se invece prendessimo un approccio diverso, ovvero più frazionato ma che rispetti appieno il principio di singola responsabilità?

```
abstract public class Sql {
  public Sql(String table, Column[] columns)
  abstract public String generate();
}

public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[]
      ↪ fields)
  @Override public String generate()
  private String valuesList(Object[] fields, final Column[]
      ↪ columns)
}
```

```
public class SelectWithCriteriaSql extends Sql {
  public SelectWithCriteriaSql(
  String table, Column[] columns, Criteria criteria)
  @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
  public SelectWithMatchSql(
  String table, Column[] columns, Column column, String
      ↪ pattern)
  @Override public String generate()
}

public class FindByKeySql extends Sql
  public FindByKeySql(
  String table, Column[] columns, String keyColumn, String
      ↪ keyValue)
  @Override public String generate()
}

public class PreparedInsertSql extends Sql {
  public PreparedInsertSql(String table, Column[] columns)
  @Override public String generate() {
  private String placeholderList(Column[] columns)
}

public class Where {
  public Where(String criteria)
  public String generate()
}

public class ColumnList {
  public ColumnList(Column[] columns)
  public String generate()
}
```

Ora abbiamo, invece di una singola classe con metodi diversificati e metodi usati solo da alcuni di questi, un gruppo di classi che rispettano sia il principio di singola responsabilità sia quello dell'open close. In questo modo è anche più testabile, meno prono

ad errori di diffusione.

Capitolo 11

Sistemi

11.1 Separare il costruire un sistema dall'usarlo

11.1.1 Separazione del Main

Nel Main possiamo costruire i vari moduli che poi verranno passati alla nostra applicazione, in questo modo il flow è molto più controllabile.

11.1.2 Factory

Le factory possono essere utilizzate per creare oggetti quando richiesto ma senza doverne inserire la costruzione all'interno del codice principale.

11.1.3 Dependecy injection

Alla base di ciò c'è l'inversione di controllo, ovvero muove le repsonsabilità secondarie da un oggetto ad un altro il cui scopo è gestire ciò. Un oggetto non dovrebbe occuparsi del gestire le sue dipendenze. Ad esempio, durante l'inizializzazione non si chiede per le dipendenze ma sarà il sistema a fornirle direttamente.

Capitolo 12

Emergenza

Secondo Kent Beck ci sono quattro regole che permettono di scrivere un software progettato correttamente e il cui uso possa emergere nella lettura.

1. Far girare tutti i test
2. Non deve contenere duplicazioni
3. Esprime le intenzioni del programmatore
4. Minimizza il numero di classi e metodi

12.1 Regola numero 1: Far girare tutti i test

Un sistema che passa tutti i test è un sistema testabile e che lavora come richiesto.

Troppe dipendenze rende difficile creare test.

12.2 Regole 2-4: refactoring

Ora che abbiamo un codice testabile possiamo raffinarlo senza la paura di non notare un errore che spacchi tutto.

12.3 Niente duplicazioni

Ogni duplicazione di codice è un errore per due motivi:

- è uno spreco cognitivo
- se un bug insorge da una parte e viene corretto solo lì, le altre rimarranno scoperte

In certi casi la duplicazione non è evidente fino a quando non si ragiona:

```
int size() {}
boolean isEmpty() {}
```

è uno spreco in quanto si può tranquillamente scrivere

```
boolean isEmpty() {
  return 0 == size();
}
```

Il design pattern template è un metodo usato tipicamente per rimuovere le duplicazioni ad alto livello:

```
public class VacationPolicy {
  public void accrueUSDivisionVacation() {
    // code to calculate vacation based on hours worked to
        ↪ date
    // ...
    // code to ensure vacation meets US minimums
    // ...
    // code to apply vaction to payroll record
    // ...
  }

  public void accrueEUDivisionVacation() {
    // code to calculate vacation based on hours worked to
        ↪ date
    // ...
    // code to ensure vacation meets EU minimums
```

```
    // ...
    // code to apply vaction to payroll record
    // ...
  }
}
```

12.4 Espressività

Tutto ciò che è stato detto fino ad ora sono consigli su come rendere il codice espressivo:

- scegliere nomi espressivi
- creare metodi brevi e semplici
- usare nomenclatura standard se esiste, ad esempio strategy
- scrivere unit test comprensibili

12.5 Classi e metodi minimali

Cercare di tenere tutto di lunghezza ragionevole permette di capire l'atomicità delle azioni effettuate al loro interno, in questo modo è anche più semplice seguire il flusso, soprattutto con nomi corretti e autoesplicativi.

Capitolo 13

Concorrenza

La concorrenza è sempre difficile, sia scriverela bene e senza che si rompa. Ci sono una serie di raccomandazioni per cercare di renderla più semplice.

13.1 Miti ed errori concettuali

13.1.1 La conccorenza migliora sempre le performance

Non sempre, solo se c'è tanto tempo d'attesa che i thread possono condividere

13.1.2 Il design non cambia quando si scrive un programma concorrente

Un algoritmo classico è molto differente se creato in maniera concorrente. La divisione di cosa debba essere fatto e quando ha solitamente un forte effetto sul design

13.1.3 Comprendere i problemi di concorrenza non è importante quando si lavora ad esempio su di una rete

In realtà sapere cosa potrebbe bloccarlo è essenziale

13.1.4 Cose vere

- la concorrenza crea dell'overhead in performance e codice
- La concorrenza è complessa per ogni problema
- i bug derivanti dalla concorrenza non sono spesso ripetibili, quindi sono spesso ignoranti come casi isolati
- la concorrenza spesso richiede un cambio fondamentale di design

13.2 Sfide

Immaginiamo che questo pezzo di codice sia condiviso tra due thread:

```
public class X {
  private int lastIdUsed;
  public int getNextId() {
    return ++lastIdUsed;
  }
}
```

Non sappiamo in che ordine due thread toccheranno lastIdUsed, questo perchè le elaborazioni di due thread possono seguire differenti percorsi. Potremmo trovarci quindi in una situazione nella quale due thread toccano allo stesso momento lastIdUsed e così generano un problema di coerenza.

13.3 Principi per la difesa dalla concorrenza

13.3.1 Principio di singola responsabilità

- Il codice legato alla concorrenza ha il suo ciclo di sviluppo, cambiamento e perfezionamento
- Il codice legato alla concorrenza ha le sue sfide che sono diverse e più difficili rispetto a quelle date da codice non concorrente
- Il numero di modi nei quali un codice legato alla concorrenza scritto male può fallire rende più complicata la sua analisi.

Raccomandazione: mantenere il codice legato alla concorrenza separato da altro codice.

13.3.2 Limitare la condivisione delle risorse

Il primo trucco da adottare è vedere se c'è un accessore che permetta al compilatore di comprendere che una risorsa deve essere soggetta ad operazioni atomiche, ovvero non interrompibili (e.g. synchronized in Java). In più si dovrebbe cercare di ridurre il numero di utilizzi di sezioni critiche in quanto:

- prima o poi queste sezioni di codice verranno aggiornate e quando anche una sola verrà dimenticata il codice si romperà
- alto rischio di duplicazione di codice
- Sarà difficile trovare la fonte degli errori

L'incapsulazione è un'ottima soluzione.

13.3.3 Usare copie dei dati

Una possibilità per evitare problemi di concorrenza è copiare i dati da passare ai vari thread in modalità sola lettura oppure passare i dati a tutti i thread, permettere di modificarli indipendentemente e poi avere un thread che si occupi della discriminazione dei risultati per poi unirli.

13.3.4 I thread dovrebbero essere indipendenti

Ogni thread dovrebbe avere i suoi dati non condivisi e lavorare indipendentemente dagli altri.

13.4 Conoscere le librerie

Le librerie possono offrire soluzioni thread safe. In java vanilla ci sono java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.

13.5 Conoscere i modelli di esecuzione

Per capire come eseguire certi lavori prima dobbiamo capire i rischi della concorrenza:

13.5.1 Produttore-consumatore

Uno o più thread che producono lavoro lo mettono in una coda o in un buffer. Uno o più thread acquisiscono il lavoro dalla coda e lo completano. La coda tra il produttore e il consumatore è una risorsa bloccata, ovvero il produtore deve attendere che si liberi dello spazio in coda prima di scrivervi e il consumatore deve attendere fino a quando c'è qualcosa da consumare in coda.

Risorse bloccata	Risorse di una grandezza limitata o valori usati in un ambiente concorrente. Ad esempio le connessioni ad un DB
Mutua esclusione	Un solo thread può accedere ai dati condivisi o una risorsa condivisa alla volta
Starvation	Un thread o un gruppo di thread non possono procedere per un tempo eccessivo. Ad esempio, permettere sempre i thread più veloci per primi può portare a non lasciare tempo a quelli più lenti
Deadlock	Due o più thread si attendono a vicenda per terminare
Livelock	Due thread provano a portare avanti il lavoro ma trovano sempre l'altro thread "sulla strada"

13.5.2 Lettore-scrittore

Quando sia ha una risorsa condivisa che serve primariamente come fonte d'informazione per i lettori ma che occasionalmente è aggiornata dagli scrittori, la quantità di dati scritta un problema. Enfatizzare il flusso dati in entrata può provocare starvation e l'accumulazione di informazioni non aggiornate.

Il problema è bilanciare lettura e scrittura in modo che i dati siano sempre aggiornati per i lettori e che gli scrittori possano aggiornarli. Una classica strategia è attendere che non ci siano lettori per permettere allo scrittore di andare avanti.

13.5.3 La cena dei filosofi

Immaginiamo dei filosofi attorno ad un tavolo. Una forchetta è alla sinistra di ogni filosofo e c'è una ciotola di pasta al centro del tavolo. I filosofi continuano a pensare fino a quando non hanno fame. Quando sono affamati prendono una forchetta da un qualsiasi lato e mangiano. Un filosofo non può mangiare a meno che non stia tenendo due forchette. Se un filosofo al proprio fianco sta usando le posate deve attendere. Una volta che il filosofo ha mangiato rimette le posate a posto.

Questo è un classico esempio di deadlock, livelock. Un modo

per risolverla è l'uso di un semaforo per le risorse.

13.6 Attenzione alle dipendenze tra metodi sincronizzati

Se più thread condividono le stesse risorse è possibile che ci sia un problema di design.

Ci saranno volte che si sarà costretti ad usare risorse condivise, in quel caso si possono tenere a mente certi principi:

- Lock basato sul client: il client blocca la catena di chiamate
- Lock basato sul server: All'interno del server si crea un metodo che blocchi il server.
- Server adattivo: Crea un intermediario che si occupa del blocco

13.7 Mantenere le sezioni sincronizzate piccole

Se una sezione sincronizzata fosse troppo grossa diventerebbe un collo di bottiglia. Immaginiamo che più persone debbano passare da una fessura, una sola alla volta: se la fessura è corta allora il tempo di attesa sarà minimo, se invece è troppo lunga il tempo si estende, bloccando tutte le altre persone. Uguale per i thread con le sezioni sincronizzate.

13.8 Scrivere bene del codice di terminazione è difficile

Una cosa è scrivere un sistema che debba lavorare per un tempo indefinito, un'altra scrivere un codice che si spenga in maniera aggraziata.

Alcuni thread magari sono in attesa di segnali che non arrivano, oppure processi genitori che attendono tutti i figli ma alcuni di questi sono in deadlock e quindi il processo genitore aspetterà per sempre[1].

13.9 Testare codice multithreaded

Una checklist possibile per testare il codice in maniera sicura:

1. Trattare i bug sporadici come possibili problemi di multithreading
2. Testare prima il codice non multithreaded
3. Rendere il codice multithreaded inseribile
4. Rendere il codice multithreaded aggiustabile
5. Testarlo con più thread che processori
6. Testarlo su più piattaforme
7. Creare test che possano forzare i fallimenti

13.9.1 Trattare i bug sporadici come possibili problemi di multithreading

Non bisogna ignorarli pensando che siano casi isolati. Per quanto possano essere rari e per quanto possa essere difficile scatenarli, non bisogna darsi per vinti.

13.9.2 Testare prima il codice non multithreaded

Se non funziona il codice non threaded allora perchè dovremmo concentrarci sul renderlo multithreaded?

[1] Chi l'avrebbe mai detto che un dramma familiare potesse bloccare un programma dallo spegnersi

13.9.3 Rendere il codice multithreaded inseribile

Scrivere il codice in modo che sia configurabile:

- Un thread, più thread, variati in base all'esecuzione
- Il thread interagisce con qualcosa che può essere reale o un doppio per il test
- Eseguire il test con casi veloci, lenti, variaibli
- Configurare i test in modo che possano essere eseguiti per un numero di interazioni

13.9.4 Rendere il codice multithreaded aggiustabile

Trovare il giusto equilibrio di thread solitamente è un lavoro di svariate prove. Il trucco è trovare un modo di misurare le performance.

13.9.5 Testarlo con più thread che processori

QUando il computer passa da un task all'altro possono accadere cose, per questo è meglio incentivare questo cambio tramite la creazione di più thread di quanti gestibili.

13.9.6 Testarlo su più piattaforme

DIfferente sistema operativo, differente sistema di gestione dei thread. Testarlo su più sistemi permette di capire quali siano le parti incriminate.

13.9.7 Creare test che possano forzare i fallimenti

In certi casi bisogna forzare dei casi limite. ome fare? Ci sono ad esempio Object.wait(), Object.sleep(), Object.yield(), Ob-

ject.priority() che possono modificare il flusso dei thread. Ci sono due opzioni qua:

- scriverlo a mano
- automatizzarlo

Scriverlo a mano

Questa è la via da scegliere quando si vuole testare un pezzo specifico di codice. Ad esempio:

```
public synchronized String nextUrlOrNull() {
  if(hasNext()) {
    String url = urlGenerator.next();
    Thread.yield(); // inserted for testing.
    updateHasNext();
    return url;
  }
  return null;
}
```

La chiamata a yield() cambia l'esecuzione e probabilmente ne causa il fallimento. Se il codice salta non è però a causa dello yield. CI sono svariati problemi con questo approccio:

- devi trovare i punti appropriati
- come si fa a sapere quali chiamate fare e dove?
- lasciare questo codice in produzione rallenta tutto
- è un approccio shotgun: potresti non trovare i difetti o no ma le probabilità non sono dalla tua

Automatizzato

Si possono usare framework appositi per programmare queste chiamate nel codice. Ad esempio con un solo metodo

```
public class ThreadJigglePoint {
  public static void jiggle() {
  }
}
```

Che venga chiamato in più punti

```
public synchronized String nextUrlOrNull() {
  if(hasNext()) {
    ThreadJiglePoint.jiggle();
    String url = urlGenerator.next();
    ThreadJiglePoint.jiggle();
    updateHasNext();
    ThreadJiglePoint.jiggle();
    return url;
  }
  return null;
}
```

Ora si ha un semplice aspetto che sceglie se fare niente, andare in sleep o prendere la precedenza.

Oppure potremmo avere due implementazioni: una in produzione che fa niente e una per il test che genera numeri a caso e sceglie cosa fare. Se si esegue il test migliaia di volte i risultati varieranno.

C'è un programma di IBM chiamato ConTest che fa esattamente ciò.

Capitolo 14

Code smells ed euristica

14.1 Commenti

14.1.1 Informazione inappropriata

In certi casi le informazioni non andrebbero incluse all'interno dei commenti ma, ad esempio, dei servizi di versionamento. I commenti che diventano obsoleti con l'evoluzione del codice tendono a rimanere se sono inseriti direttamente nel codice.

14.1.2 Commento obsoleto

Se un commento non è più attuale è meglio cancellarlo o modificarlo.

14.1.3 Commenti ridondanti

Stesso esempio a pagina 44

14.1.4 Commenti scritti male

Un commento sgrammaticato è il peggior commento che ci sia.

14.1.5 Codice commentato

Meglio cancellare parti di codice tramite sistema di versionamento e non commentandole, diventano solo un peso per la struttura poi.

14.2 Ambiente

14.2.1 La build richiede più di uno step

Fare una build dovrebbe essere una singola operazione triviale.

14.2.2 I test richiedono più di uno step

Lanciare un test dovrebbe essere un'operazione che richiede, come per la build, una singola istruzione, che sia un comando una shell o un bottone in un'IDE.

14.3 Funzioni

14.3.1 Troppi argomenti

Stessa spiegazione vista da pagina 29.

14.3.2 Argomenti di output

Stessa spiegazione vista da pagina 32.

14.3.3 Argomenti di guardia

Stessa spiegazione vista da pagina 29.

14.3.4 Funzioni morte

Le funzioni che non vengono mai chiamate dovrebbero essere scartate.

14.4 Generali

14.4.1 Più linguaggi in un solo file sorgente

Il fatto che si possa fare non vuol dire che sia una soluzione intelligente o chiara a chi legge.

14.4.2 Comportamento ovvio non implementato

Ogni funzione dovrebbe implementare ciò che un programmatore può aspettarsi dal suo nome.

14.4.3 Comportamento errato sui limiti

Il problema di certi programmi spesso è che non viene considerato correttamente cosa potrebbe andare storto nelle situazioni limite.

14.4.4 Violare i limiti di sicurezza

I limiti ci sono per un motivo, semplicemente decidere di ignorarli per rendere più semplice la fase di testing è un errore che può costare caro.

14.4.5 Duplicazione

Vedere pagina 80.

14.4.6 Codificare al livello di astrazione sbagliato

L'astrazione è ciò che ci permette di separare i vari livelli concettuali di un programma. Violarla ci impedisce di ragionare al meglio su cosa vogliamo fare, in quale caso e cosa vogliamo dire al successivo programmatore.

14.4.7 Troppe informazioni

Le interfacce non dovrebbero offrire troppe funzioni da cui dipendere. Quindi solitamente si impara a limitare cosa si espone nelle interfacce. Nascondere i dati, nascondere le funzioni di utilità, nascondere le costanti e i temporanei.

14.4.8 Codice morto

Il codice non eseguito è codice inutile. Il codice inutile appesantisce. Cancella il codice morto, merita il suo eterno riposo.

14.4.9 Separazione verticale

Le variabili e le funzioni dovrebbero essere definite vicino a dove verranno usate per la prima volta.[1].

[1]Dissento su questa visione dell'autore, dipende da come uno è abituato ma lo standard è dichiarare le variabili, costruttori, metodi pubblici, metodi di utilità, getter e setter

14.4.10 Inconsistenza

Sempre perchè il codice deve essere facilmente interpretabile, se da una parte abbiamo usato un nome per una variabile legata ad una cerca classe, allora anche dalle altre parti chiamiamo sempre allo stesso modo quella variabile e non usiamo quel nome per altre cose.

14.4.11 Clutter

Commenti che non danno informazioni, costruttori non implementati, perchè bisogna lasciare tutte queste cose che non danno informazioni? Via.

14.4.12 Dipendenza artificiale

Non inseriamo variabili o altro in posti a cui non appartengono logicamente, come ad esempio forzare un'enumerazione perchè vogliamo che funzioni in una maniera specifica in una sottoclasse.

14.4.13 Feature envy

Una classe interessata alle variabili e ai metodi di un'altra classe è una classe non pensata correttamente.

14.4.14 Intenti non chiari

Se non si capiscce quale sia l'intento del codice semplicemente leggendolo, allora qualcosa non va. Ad esempio questo codice è da bruciare, se scrivi codice così cerca aiuto:

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
```

```
}
```

14.4.15 Responsabilità male posizionate

Ogni tanto mettiamo giustamente certe funzioni come statiche in altri casi no. Ora, se lavoriamo su di una libreria come Math è giustissimo, se vogliamo creare delle classi da estendere, rendere polimorfiche e quindi modificando l'implementazione, allora forse è meglio evitare.

14.4.16 Static dove non dovrebbe essere

Se un metodo non opera su di una singola istanza allora va bene che sia statico. Un problema è quando non si considera il possibile polimorfismo di una funzione, in quel caso non può essere statica in quanto sarebbe chiamata col nome della classe.

14.4.17 Usare variabili di spiegazione

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

Le variabili key e value sono variabili di spiegazione, ovvero si mostra in maniera diretta cosa rappresentino, il valore viene racchiuso in una variabile leggibile e chiara.

14.4.18 I nomi delle funzioni dovrebbero dire cosa fanno

Il nome di una funzione non dovrebbe lasciare dubbi sul cosa faccia, cosa prenda in ingresso e cosa verrà restituito alla fine.

14.4.19 Comprendere l'algoritmo

Prima di terminare il proprio lavoro perchè "funziona" è meglio essere sicuri di aver compreso cosa il codice da noi scritto stia facendo, il perchè funzioni e capire perchè questa soluzione è corretta. L'approccio migliore è prima arrivare ad un "funziona" e poi fare refactoring da lì.

14.4.20 Rendere fisiche le dipendenze logiche

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee>
      ↪ employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
  }

  private void printAndClearItemList() {
    formatter.format(page);
    page.clear();
  }

  private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
```

```
    item.tenths = e.getTenthsWorked() % 10;
    page.add(item);
  }

  public class LineItem {
    public String name;
    public int hours;
    public int tenths;
  }
}
```

Se un modulo dipende da un altro, la dipendenza dovrebbe essere fisica, non solo logica. Il modulo dipendente non dovrebbe fare assunzioni (ovvero dipendenze logiche) riguardo al modulo da cui dipende. Dovrebbe chiedere esplicitamente al modulo le informazioni che gli servono.

Questo codice, ad esempio, ha una dipendenza logica che non è stata resa fisica: PAGE_SIZE. Perchè questa classe dovrebbe conoscere la dimensione della pagina? Possiamo rendere fisica la dipendenza creando un metodo chiamato getMaxPageSize(), questo perchè magari un'altra implementazione potrebbe avere un valore differente.

14.4.21 Preferire il polimorfismo a if/else o switch

Proposta di regola: non ci possono essere più di uno switch per un certo tipo di selezione. I casi in quello switch devono creare degli oggetti polimorfici che prendono il posto di altri switch nel resto del sistema.

14.4.22 Seguire convenzioni standard

Ogni team dovrebbe rispettare le norme generali dell'industria. Ad esempio, si dovrebbe specificare dove dichiarare le istanze, come chiamare i metodi, le classi e le variabili.

14.4.23 Sostituire i numeri magici con costanti

Ad esempio, il numero 86400 dovrebbe essere nascosto tramite una costante SECONDS_PER_DAY. Se si stanno stampando 55 linee per pagina, allora avremo la costante LINES_PER_PAGE.

14.4.24 Essere precisi

Non prendersi del tempo per pensare ai casi limite, alle ragioni per scegliere una variabile o un'altra, cosa potrebbe restituire una query, è un errore.

14.4.25 Struttura sopra convenzione

Le convenzioni sono comode però a seconda dell'ambito può esserci fin troppa licenza, ad esempio su come nominare i campi di uno switch controllo il trovarsi a dovere dare nomi precisi per una classe astratta.

14.4.26 Condizioni incapsulate

Meglio estrarre le funzioni che spiegano l'intendo di una condizione:

```
if (shouldBeDeleted(timer))
```

è meglio di

```
if (timer.hasExpired() && !timer.isRecurrent())
```

14.4.27 Evitare le condizioni negative

```
if (buffer.shouldCompact())
```

è meglio di

```
if (!buffer.shouldNotCompact())
```

14.4.28 Le funzioni dovrebbero fare una sola cosa

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

è molto meno comprensibile rispetto a

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
if (e.isPayday())
  calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

In questo caso le funzioni sono ben separate e ci permettono di controllare in una maniera più naturale ognuna di esse.