Clean Code

Robert C. Martin

Jacopo De Angelis

30 luglio 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.

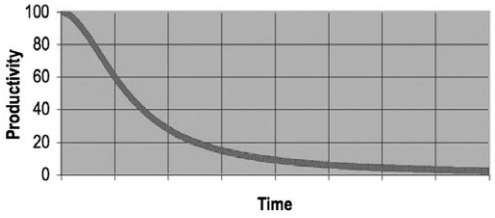

Figura 1.1: Produttività vs tempo

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