# Clean Code

Robert C. Martin

Jacopo De Angelis

28 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.

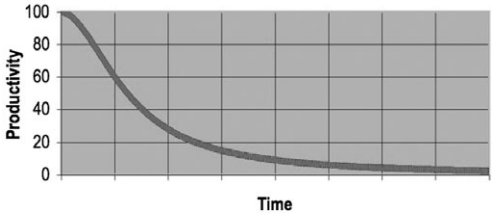


Figura 1.1: Produttività vs tempo

## 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 Piccoli