Tutorato Progettazione, Algoritmi e Computabilità

Analisi statica e dinamica del codice

Dott. Michele Beretta

Prof.ssa Patrizia Scandurra

Università degli Studi di Bergamo

2023 – 2024

Outline

Metriche del software

Qualità del software

Le metriche del software consentono la valutazione della qualità:

- Del prodotto software
- Dei dati
- Del processo di produzione

Qualità del software

La qualità di un prodotto software viene definita attraverso 8 caratteristiche:

- Idoneità funzionale
- Manutenibilità
- Affidabilità
- Usabilità
- Portabilità
- Efficienza
- Compatibilità
- Sicurezza

Qualità del software

Il monitoraggio della qualità può essere realizzato in diversi momenti, in funzione del tipo di qualità analizzata:

- *Qualità interna*: viene misurata in modalità statica durante la realizzazione.
- *Qualità esterna*: viene misurata in modalità dinamica durante il test del sistema.
- *Qualità in uso*: si intende come qualità percepita (in ambiente simulato o reale) durante l'uso del software (efficacia, efficienza, soddisfazione dell'utente, esenzione del rischio, copertura contestuale).

Rispettare gli standard di qualità

Fintanto che la dimensione del software è ridotta, è molto semplice per gli sviluppatori garantire la qualità, in quanto è possibile mantenere sotto controllo tutte le porzioni del codice.

Con l'aumento della dimensione del codice, invece, il software diventa:

- Difficile da testare
- Difficile da estendere
- Difficile da mantenere

Molto spesso, questo, porta ad avere un software che diventa sempre più *monolitico*, in cui si hanno dipendenze non necessarie tra qualsiasi pezzo di codice.

Rispettare gli standard di qualità

La soluzione migliore, per garantire un elevato standard di qualità del software, è quella di utilizzare diversi tool che permettono di individuare:

- Errori di progettazione
- Implementazione non ottimale
- Punti critici con potenziali bug (*code smells*)

Quando parliamo di interventi sui punti critici del software (senza alternarne le funzionalità) si parla di *refactoring*.

Rispettare gli standard di qualità

Code smell	Descrizione
Rigidity	Il sistema è difficile da modificare perché ogni cambiamento implica altri cambiamenti.
Fragility	Eventuali cambiamenti causano la rottura del sistema in tanti sotto-sistemi non collegati l'un l'altro.
Immobility	È difficile suddividere il sistema in tanti componenti riutilizzabili.
Viscosity	È la proprietà di un sistema, progettato male, in cui aggiungere cose corrette risulta più difficile dell'aggiungere cose errate.
Opacity	Il codice è difficile da leggere ed interpretare. In questo caso il codice non esprime bene quelli che sono i suoi "intenti"

Metriche

La qualità del software può essere misurata utilizzando una serie di metriche che rappresentano un insieme di caratteristiche **matematicamente misurabili** del prodotto software.

- **Ca: Afferent Coupling**
  Rappresenta il numero di classi al di fuori di una categoria (package) che dipendono dalle classi che si trovano all'interno.
- **Ce: Efferent Coupling**
  Rappresenta il numero di classi all'interno di una categoria (package) che dipendono dalle classi che si trovano all'esterno.

Metriche

- **I: Instability**

$$I = \frac{C_a}{C_a + C_e} \in [0, 1]$$

$I = 0$ indica massima stabilità, $I = 1$ indica massima instabilità.

- **A: Abstractness**

$$\frac{\#\text{ di classi astratte}}{\#\text{ di classi}}$$

$A =$ indica che si hanno solo classi concrete, $A = 1$ solo classi astratte.

Legame fra Abstractness e Instability

I due punti che rappresentano le condizioni migliori per una categoria, sono

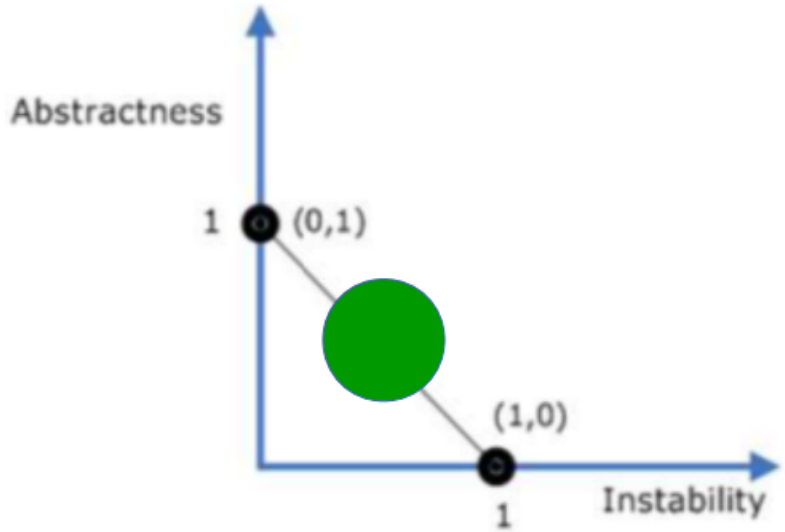
- $(0, 1)$ che rappresenta massima astrazione e minima instabilità
- $(1, 0)$ che rappresenta massima concretezza e massima instabilità

I punti sul segmento diagonale vengono detti **di equilibrio**.

Esempi:

- Categoria con $A = 0, I = 0$: massima concretezza, massima stabilità, ma risulta **rigida** e non può essere estesa a causa della sua poca modificabilità.
- Categoria con $A = 1, I = 1$: come al punto precedente.

Legame fra Abstractness e Instability



- Categoria con $A = 0.5, I = 0.5$: abbiamo una parziale estensibilità (essendo la categoria parzialmente astratta) con il vantaggio che le estensioni non sono soggette ad instabilità (essendo parzialmente stabili). La categoria è **equilibrata**.

STAN4J – Analisi statica del codice

Stan4j permette di effettuare un'analisi visiva del progetto nella sua interezza, e di esplorarne i vari moduli e pacchetti presenti per vedere di cosa essi sono composti e quali e quante dipendenze possiedono.

STAN4J

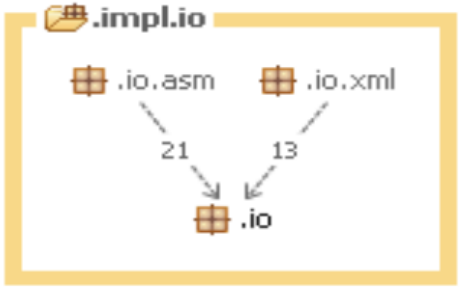
STAN4J supporta diverse metriche, tra cui:

- Metriche classiche viste nelle slide precedenti
- Estimated Lines of Code
- Cyclomatic Complexity
- Average Component Dependency, Fat and Tangled
- Etc.

STAN4J – Views

STAN4J include diverse views:

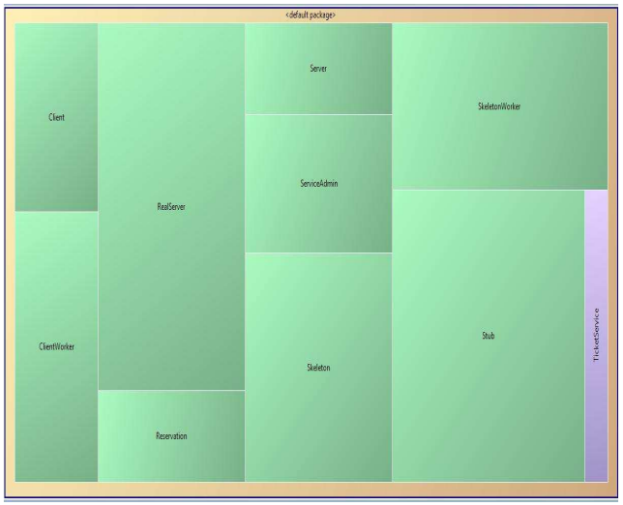
- **Composition View**: mostra i package che contengono il software sotto analisi.



- **Couplings View**: permette di analizzare un singolo elemento e di leggerne le dipendenze in entrata ed in uscita. I numeri sulle linee esprimono il numero di volte che una classe ne richiama un'altra.

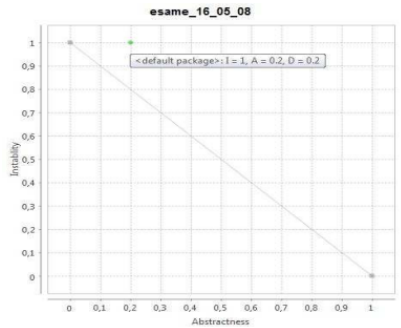
STAN4J – Views

- **Map**: consente di individuare le dipendenze attraverso un approccio visivo legato ai colori. Il codice rappresentato da uno dei blocchi dipende dal codice rappresentato dal blocco sottostante e ha una dipendenza bidirezionale dai blocchi collocati sul suo stesso livello. Selezionando il pacchetto in esame, la divisione sarà:
  - Il pacchetto selezionato (color ambra).
  - Gli elementi richiesti dal pacchetto selezionato (color verde).
  - Gli elementi che dipendono dal pacchetto selezionato (color blu/viola).



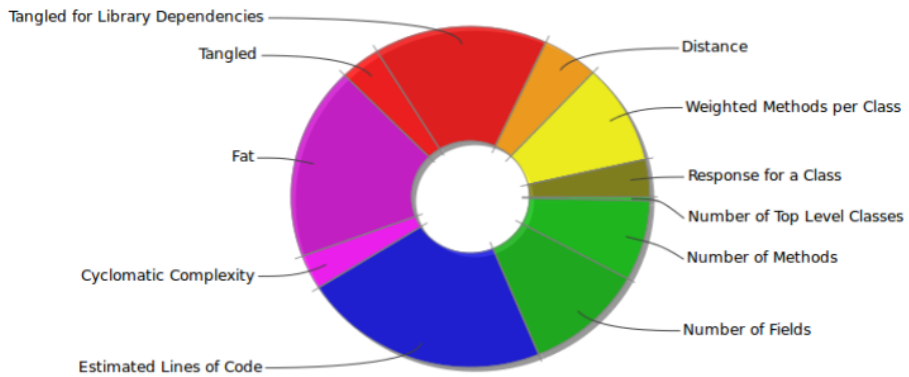
STAN4J – Views

- **Distance**: consente di valutare la distanza che il progetto (o i suoi elementi) ha dalla linea di stabilità nel grafico Instabilità-Astrazione.



- **Fat**: è calcolato come il numero di archi presenti nel grafo delle dipendenze tra tutti i metodi della classe. L'obiettivo di questa metrica è quello di limitare il più possibile la dimensione del codice riducendo il numero dei metodi presenti.

- **Pollution**: indica il grado di "inquinamento" del codice e la percentuale in base a questo grado, di tutte le metriche violate (vedi slide successiva).



STAN4J – Views

Metrica	Descrizione
Depth of inheritance treee	È la posizione della classe all'interno dell'albero di eriditarietà. Si consiglia di mantenere il numero dei livelli da attraversare per giungere alla radice al di sotto di 5
Response for class	È il numero di tutti i metodi implementati da una classe più il numero di metodi accessibili tramite un oggetto della classe. Si consiglia di mantenere questo numero basso. Maggiore è la RFC maggiore è lo sforzo per apportare modifiche
Afferent couplings	È il numero delle classi che utilizzano una classe
Efferent couplings	È il numero delle classi che una classe utilizza
Lack of cohesion of methods	Essa misura il numero di componenti collegati ad una classe. Un valore basso indica che il codice è semplice da comprendere e riutilizzabile. Si consiglia di splittare in più classi se il valore è maggiore di 2
Weighted methods per class	È il numero medio di metodi contenuti in una classe, non dovrebbe mai superare 14
Cyclomatic Complexity	Essa misura il numero di cammini linearmente indipendenti all'interno di una parte di codice. Più il numero è alto, più lo sforzo per testare le funzionalità del software sarà alto.
Distance	Rappresenta la distanza dalla linea ideale A+I=1. Identifica quanto una categoria è lontana dal caso ideale. Minore è la distanza del software dalla linea maggiore è la sua qualità.

Refactoring

Refactoring del codice

Eclipse mette a disposizione diversi strumenti per assistere e rendere semi-automatico il refactoring del codice. Questi strumenti sono disponibili dal menù a cui si accede tramite tasto destro sul codice sorgente e click sulla voce `refactor`:

- *Rename*: la classe viene rinominata e la modifica si propaga per tutto il codice.
- *Move*: cambia il package od il progetto a cui la classe appartiene, propagando la modifica.
- *Extract Interface*: estrae l'interfaccia della classe (conterrà i metodi della classe selezionata).
- *Extract Superclass*: definisce una superclasse della classe selezionata (conterrà i metodi della classe selezionata).

Refactoring del codice

- *Pull Up*: ridefinisce l'oggetto selezionato come istanza della superclasse.
- *Push Down*: ridefinisce l'oggetto selezionato come istanza della sottoclasse.
- *Introduce parameter*: aggiunge un parametro al metodo selezionato.
- *Introduce Factory*: introduce un metodo che restituisce un'istanza della classe chiamandone il costruttore al suo interno. È utile per implementare il pattern singleton.
- *Change method signature*: cambia la segnatura del metodo selezionato.

JUnit – Analisi Dinamica del codice

Unit testing

Lo **unit testing** permette di verificare singole parti (*unità*) di un codice sorgente.

Unità: può essere un singolo programma, una funzione o una procedura. Normalmente la più piccola unità testabile è il metodo. Lo unit testing si articola in diversi test case che vanno a testare le singole unità nel modo più indipendente possibile.

Nota

Lo unit testing viene eseguito dagli sviluppatori e non dagli utenti finali.

Unit testing

Lo **unit testing**:

- *Semplifica le modifiche*: un modulo che passa lo unit testing può essere in seguito modificato. Se dopo la modifica continua a passare lo unit testing il modulo continuerà a funzionare correttamente con il resto del programma.
- *Supporta la documentazione*: un test di unità rappresenta un esempio concreto di utilizzo dell'API del modulo.

Test Driven Development

Esiste una metodologia di sviluppo chiamata *Test Driven Development* che si basa su test d'unità e prevede:

1. Scrittura dei casi di test partendo dalle specifiche.
2. Esecuzione dei test (che inizialmente falliscono).
3. Scrittura del codice fino a quando tutti i casi di test passano.
4. Ricominciare dal punto 1.
5. Ogni volta che si rileva un difetto si riparte dal punto 1.

In questo modo il progetto ed il codice evolvono sotto la guida di test e scenari reali: lo sforzo per l'implementazione è maggiore ma si ha un prodotto di maggiore qualità e con una documentazione automatica.

JUnit

JUnit è lo strumento che permette di eseguire test di unità in Java, sfruttando la proprietà della **reflection**, secondo cui i programmi Java possono esaminare il loro stesso codice.

Tramite JUnit i programmatori possono:

1. Definire ed eseguire test e test-set.
2. Formalizzare in codice i requisiti delle varie unità.
3. Scrivere e debuggare il codice.
4. Integrare il codice tenendolo sempre funzionante.

Nota

JUnit è integrato in molti IDE. Noi sfrutteremo la versione 4 che sfrutta le annotation di Java per semplificare la creazione di casi di test.

Esempio 1

Un caso di test con JUnit consiste in una classe ausiliaria con metodi annotati con `@Test` in cui si controlla la corretta esecuzione del codice da testare tramite istruzioni come `assertEquals`.

Un esempio di codice che testa la moltiplicazione è il seguente:

```
public class Test {
    @Test public void testMult() {
        assertEquals(4, 2 * 2);
    }
}
```

Esempio 2

Vogliamo testare una classe `Counter` che rappresenta un contatore con le seguenti operazioni:

- Un *costruttore* che crea un contatore e lo setta a 0.
- Metodo `inc` che incrementa il contatore e restituisce il nuovo valore.
- Metodo `dec` che decrementa il contatore e restituisce il nuovo valore.

```
public class Counter {
    public Counter() {}
    public int inc() {}
    public int dec() {}
}
```

Esempio 2

Creiamo la classe `CounterTest`

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
    @Test public void testInc() {}
    @Test public void testDec() {}
}
```

Test di un metodo

Per testare un metodo con JUnit dobbiamo:

- Creare eventuali oggetti della classe sotto test.
- Chiamare il metodo da testare ed ottenere il risultato.
- Confrontare il risultato ottenuto con quello atteso.

Per fare questo, utilizziamo dei metodi assert di JUnit che permettono di eseguire il controllo e, nel caso in cui il controllo fallisca, JUnit cattura il fallimento e lo comunica al tester.

Nota

Le operazioni da 1 a 3 devono poi essere ripetute per tutti i metodi da testare.

Esempio 2

Implementiamo ora il metodo che consente di testare l'operazione di incremento:

```
public class CounterTest {
    @Test public void testInc() {
        Counter c = new Counter();
        assertEquals(1, c.inc());
    }
}
```

Metodi assert

Ci sono molti metodi assert:

- `assertEquals(expectedValue, actualValue)`: permette di controllare l'uguaglianza tra `expectedValue` e `actualValue`. Il confronto viene effettuato in automatico utilizzando l'equals di `Object`.
- `assertTrue(expression)` e `assertFalse(expression)`: controllare che expression sia vera o falsa.
- `fail()` e `fail(String message)`: permette di terminare con un fallimento permettono di forzato.
- `assertSame(expectedValue, actualValue)` e `assertNotSame(expectedValue, actualValue)`: permette di controllare l'uguaglianza (o la non uguaglianza) tra `expectedValue` e `actualValue` utilizzando == (o `!=`) per il confronto.

Metodi assert

Ci sono molti metodi assert:

- `assertNull(expression)` e `assertNotNull(expression)`: permettono di controllare che expression sia (o meno) `null`.

Alcuni assert permettono di passare come primo argomento una stringa `error_message` che rappresenta un messaggio di errore esplicito.

Quando usare `assert`

- Nel codice per
  - documentare una condizione che *deve* essere vera (`assertTrue`)
  - documentare una condizione che *non può essere mai raggiunta* (`assertFalse`)
- Nei test

Nota

Utilizzare un assert per controllare se un parametro assume un valore corretto non è una buona scelta.

`@BeforeClass`

In una classe che implementa un caso di test è possibile specificare un metodo che deve essere eseguito solamente una volta prima dei test. Il metodo deve essere annotato con `@BeforeClass`, deve essere `static` e `public`.

JUnit in Eclipse

Per scrivere un caso di test è necessario seguire questa procedura:

1. Scrivere la classe C da testare.
2. Tasto destro $C \rightarrow$ new $\rightarrow$ JUnit Test Case.
3. Selezionare JUnit4, e deselezionare `tearDown` e `setUp` che non sono necessari per piccoli progetti.
4. `next` $\rightarrow$ selezionare i metodi per i quali si vogliono creare dei casi di test.
5. Implementare i metodi di test.
6. Tasto destro sulla classe di test $\rightarrow$ Run As $\rightarrow$ JUnit Test.

Eccezioni nei test

È possibile verificare la presenza di una eccezione quando il metodo che si sta testando prevede una eccezione:

```
@Test(expected=Exception.class)
public void verificaEccezione() {}
```

Test parametrici

Alcune volte si potrebbe voler chiamare lo stesso metodo di test con dati diversi (*il codice che effettua il test non cambia, mentre i dati di ingresso e di controllo sì*).

Un esempio potrebbe essere il test di un metodo che incrementa il numero ricevuto per parametro, che potrei voler testare con input 0, 2 e 5, controllando che gli output siano rispettivamente 1, 3 e 6.

Voglio quindi fare in modo che invece di scrivere tre diversi casi di test, se ne abbia uno solo parametrico che riceve due valori: `input` e `expectedOutput`.

Test parametrici

Per ottenere questo risultato:

1. Si dichiarano tante variabili d'istanza quanti sono i parametri utilizzati nel test.
2. Si crea un costruttore del test che ha per parametri la *n*-upla identica alle variabili di istanza.
3. Si crea un metodo statico `@Parameters` che deve restituire una Collection contenente le *n*-uple di parametri con valori.
4. Si annota la classe di test con `@RunWith(Parameterized.class)`.

Test parametrici – Esempio

```
@RunWith(Parametrized.class)
public class ParameterTest {
    private int input;
    private int expectedOutput;

    public ParameterTest(int input, int expectedOutput
        this.input = input;
        this.expectedOutput = expectedOutput;
    }

    // (continues on the next slide)
```

Test parametrici – Esempio

```
    // (continued from previous slide)

    @Parameters
    public static Collection creaParametri() {
        return Arrays.asList(new Object[][]{
            {0, 1}, {2, 3}, {5, 6}
        });
    }

    @Test public void testParametrico() {
        assertEquals(
            this.expectedOutput,
            inc(this.input)
        );
    }
}
```

Copertura dei test

Eseguendo *unit testing* è essenziale valutare la copertura dei casi di test in quanto potremmo riuscire a passare con successo tutti i test ma questi potrebbero anche coprire solamente una piccola porzione del codice.

Teoricamente il valore della copertura di un insieme di casi di test dovrebbe essere il più alto possibile.

Attenzione

Copertura e correttezza del codice **non sono correlate**. Copertura al 100% non significa che il codice è corretto.

```
def add(a, b):
    return 4

assert add(1, 3) == 4
```

Tool per la copertura dei test in Eclipse: *EclEmma*.

Esercizio 1 – CoinBox

La classe `CoinBox` (vedi progetto Esercizio1) rappresenta un distributore automatico di prodotti che accetta solo monete da un quarto di dollaro.

Su richiesta, un `CoinBox` deve erogare un prodotto ogni mezzo dollaro (quindi ogni 2 monete) inserite dall'utente. Se al momento della richiesta la quantità di monete inserite non è sufficiente, il prodotto non viene erogato.

Quando il prodotto viene erogato, il quantitativo di monete inserite viene decrementato del costo del prodotto.

Implementare in JUnit i seguenti casi di test:

- `testInit`: `CoinBox` ha inizialmente un credito pari a zero.
- `testSingleVend`: inserendo due monete da un quarto di dollaro, `CoinBox` eroga il caffè
- `testNonEnough`: inserendo una moneta da un quarto di dollaro, `CoinBox` non eroga il caffè.

Esercizio 2 – Calcolatrice

La classe `Calcolatrice` (vedi progetto Esercizio2) implementa una calcolatrice. Il metodo `pow` contiene un errore.

Definire un caso di test JUnit che metta in evidenza il difetto (il test non deve passare), correggere l'errore nel codice e rieseguire il test (a quel punto deve passare).