# Software quality

## Introduzione

Obiettivi:

- Comprendere quando uno sviluppo software sia di elevata qualità
- Produrre software di elevata qualità in accordo con lo scopo della progettazione

In capitoli precedenti sono stati illustrati differenti processi di modellazione, che hanno come intento principale quello di fornire un'elevata metrica qualitativa, in grado di indirizzare il prossimo sviluppo nella strada meno tortuosa e articolata possibile, sperando di proseguire per la realizzazione dei requisiti funzionali.

Tuttavia spesso il punto su cui ci si focalizza, riguarda quali siano i principi fondamentali affinchè si possa ritenere che costrutti illustrati sino a ora siano sufficienti. Ad un primo impatto, potrebbero essere formulate della caratteristiche ritenute comuni per una qualsiasi implementazione, come la *leggibilità del codice*, non percepita dall'utente finale, comportamento adattivo del sistema di modellarsi dinanzi a problematiche, meglio definito come *meccanismo di risilienza*, oppure che abbia un atteggiamento *deterministico*, il processo esecutivo espresso dovrà corrispondere al processo atteso dal risultato finale.

Le particolarità descritte rappresentano solamente l'apice su cui fonda un *design goal*, per cui, innanzitutto, è bene soffermarsi sulla diversificazione delle *qualità*.

Semplicemente, si distinguono in *qualità esterne*, ossia tematiche che l'utente è in grado di percepire, sia che ritraggano vincoli funzionali o meno, e *qualità interne*, le quali promuovono l'organizzazione dello sviluppo software, informazioni che sono di passaggio tra sviluppatori e programmatori, per cui occorre che siano rispettate determinate regole pur di procedere verso il compimento dell'obiettivo posto.

Di seguito, sono riportate un elenco di qualità che possono caratterizzare un progetto software, di entrambe le categorie, di cui saranno descritti i concetti principali su cui si formulano.

*Qualità esterne*
*Definizione Correctness*
Per **correctness** si intende il grado in cui un sistema è esente da errori nelle sue specifiche, progettazioni e implementazioni.

*Definizione Usability*
Per **usability** si intende l'approccio in cui un utente possa comprendere ed imparare ad utilizzare e valorizzare un sistema.

*Definizione Efficiency*
Per **efficiency** si intende l'uso minimale delle risorse messe a dispozione del sistema, le quali spesso includono la complessità temporale, lo spreco di memoria oppure il tempo di ese-

cuzione.

*Definizione Reliability*
Per **reliability** si intende l'abilità di un sistema di conseguire e compiere i requisiti funzionali posti inizialmente, caratterizzato da una minima probabilità di fallimento.

*Definizione Integrity*
Per **integrity** si intende il grado in cui un sistema è capace di prevenire accessi non autorizzati o dannosi, ai propri programmi e dati collezionati. L'idea ipone l'inclusione di respingere accessi non autorizzati da molteplici utenti e di garantire che i dati possano essere estratti in maniera congrua e in linea con le direttive poste.

*Definizione Adaptability*
Per **adaptability** si intende la possibilità che un sistema possa essere riadattato, senza apportare modifiche, in applicazioni oppure ambienti che non siano specifici rispetto al fine per cui si era progettato e, successivamente, implementato il singolare processo esecutivo.

*Definizione Accuracy*
Per **accuracy** si intende il grado in cui un sistema è esente da errori, specialmente in relazione al rispetto dei risultati attesi. *Accuracy* si distingue da *correctness*, poichè dispone quanto un sistema conduca adeguatamente il proprio obiettivo, rispetto a come sia correttamente progettato e costruito.

*Definizione Robustness*
Per **robustness** si intende il grado in cui un sistema reagisca rispetto a input invalidi oppure in relazione a condizioni ambientali stressanti, illustrando la propria capacità nel produrre il risultato sperato.

Le *qualità esterne* sono le uniche caratteristiche che interessano ad utenti finali. Generalmente sono interessati alla semplicità d'uso del software, e non in relazione all'immediatezza di modifiche apportate da parte di sviluppatori; principalmente preoccupati nella correttezza del prodotto, e non sulla leggibilità del codice oppure in una buona stesura strutturale.

Tuttavia, il piccolo trafiletto posto indica qualità strettamente necessarie per tutti coloro che compiono una mansione simile.

*Qualità interne*
*Definizione Maintainability*
Per **maintainability** si intende la facilità con cui sia possibile modificare un sistema software per cambiamenti o aggiunta di capacità interne, migliorare le prestazioni oppure correggere difetti.

*Definizione Flexibility*
Per **flexibility** si intende l'intento in cui sia possibile modificare il sistema software per vari-

azioni poste al di fuori dei veri requisiti funzionali stilati ad una prima analisi.

*Definizione Portability*
Per **portability** si intende la semplicità con cui sia possibile modificare il sistema per operare all'interno di un ambiente differente rispetto a quello specificato.

*Definizione Reusability*
Per **reusability** si intende l'intento e la facilità con cui sia possibile adoperare parti di certe soluzioni software in altri sistemi.

*Definizione Readability*
Per **readability** si intende la facilità con cui sia possibile leggere e comprendere il codice sorgente, soprattutto a livello di dettagli articolati.

*Definizione Testability*
Per **testability** si intende il grado in cui si possa testare l'efficacia del sistema software, ossia se sia possibile verificare se rispecchia i requisiti funzionali.

*Definizione Understandability*
Per **understandability** si intende la semplicità con cui si possa comprendere il sistema sia a livello organizzativo che a livello strutturale.

Concludendo, la massimizzazione di certe caratteristiche inevitabilmente confligge con altre azioni di ottimizzazione delle qualità proposte. Individuare una soluzione migliore da un insieme di tematiche simili, rende molto difficile lo sviluppo software per un qualsiasi progetto.

## Modelli qualitativi

Sino a ora, sono state illustrate ed elencate un insieme di qualità che possano contraddistinguere un sistema software. Tuttavia, una descrizione del genere potrebbe essere poco incisiva, del tutto fuorviante, alludendo alla possibilità di poter imputare quali sviluppi abbiano rilevanza o meno. Solitamente, si tende a scartare l'ipotesi di assegnazione di un punteggio, poichè potrebbe essere poco adattiva rispetto al contesto narrato, per cui si adottano *layer qualitativi* in grado di stabilire prodotti software di valore.

I **modelli qualitativi** sono strettamente connessi allo schema SquaRE, evoluzione dello standard *ISO 9126*, il quale provvede a determinare tre *quality models*: **Trovare definizioni sui modelli ...**

- *Software product quality*,
- *Data quality*,
- *Quality in use*,

Oltre ai *modelli qualitativi* spesso si attuano framework affinchè si riesca nell'intento di attribuire criterio valorizzativo ad un sistema software. A titolo semplificativo, il committente dei requisiti funzionali potrebbe essere interessato ad uno sviluppo coeso che possa contraddistinguere passi di *manuntenzione*; tale effettività potrebbe essere utile per gli stessi

sviluppatori riprendendo codice sorgente leggibile e duttile. Per cui, in conclusione, tutto ciò che si desidera consiste che il software prodotto sia *riutilizzabile* e *progettato per il cambiamento*.

Come già descritto, la volontà primaria prevede la *modifiability* del codice sorgente progettato e implementato dagli sviluppatori. Nonostante, una richiesta simile pone le proprie fondamenta su principi, concetti e ideologie articolati e di difficile comprensione. Per cui, di seguito, sono presentate le tematiche fondamentali che modellano la logica legata alla *software quality*.

## Object-Oriented Principles

L'obiettivo impone la progettazione di sistemi software duttili al cambiamento, sia a livello di contesto che di requisiti funzionali posti. Per semplicità il discorso verte completamente su progetti legati al paradigma orientato agli oggetti, per cui occorre trovare il giusto compresso tra *incapsulazione*, *ereditarietà* e *poliformismo*, i quali se combinati possono garantire la massimizzazione della qualità del software.

Sono proposte brevi definizioni per completezza informativa delle caratteristiche illustrate poco fa:

*Definizione informale*
L'*incapsulamento* è la proprietà per cui un oggetto contiene al suo interno gli attributi e i metodi che accedono ai dati stessi.

*Definizione informale*
L'*ereditarietà* è la proprietà che consente di definire delle classi figlie che ereditano dalle classi padre attributi e metodi.

*Definizione informale*
Il *poliformismo* è la proprietà che permette al programma di fare uso di oggetti che espongono una stessa interfaccia, ma implementazioni diverse.

Terminando, oltre ai tre principi su cui si stabilizzano OOP, occorre soffermarsi anche sul *concetto astratto* che deve essere mantenuto, non tanto a livello alto per cui ovviando a caratteristiche superflue oppure ovvie, ma adoperando un'analisi in grado di rescindere su cosa possa essere ritenuto essenziale ai fini della qualità cardine, denominata *modifiability*.

## Design smells

Improntare un progetto verso una visione chiara e lineare di ciò che il sistema dovrebbe fare rappresenta una prima fase di una qualsiasi implementazione. Infatti, proseguendo con l'aggiunta e la modifica di ulteriori elementi, esso potrebbe essere viziato da comportamenti negativi, la cui manuntenzione diviene sempre più difficile, raggiungendo apici che possano richiedere una nuova progettazione dell'intero contesto. Per questa ragione sono introdotti i

**design smells**, ossia la qualità momentanea non è all'altezza delle aspettative.

Anche in questo contesto si evidenziano caratteristiche negative comuni, le quali possono essere riassunte come segue:

- *Rigidity*, è la tendenza che rende complicata la *modifiability* del software. Una progettazione è rigida qualora un'unica modifica causi una serie di sotto-variazioni in moduli dipendenti.
- *Fragility*, è la tendenza di un programma che smette di operare per piccole variazioni fatte. Spesso, nuove problematiche sono poste all'interno di sezioni che non hanno collegamenti diretti con aree che abbiano subuto modifiche. Tuttavia, la possibilità di risolvere problemi simili potrebbe rappresentare un'arma a doppio taglio, poichè man mano che si tenti nella risoluzione di pari passo aumenta la fragilità del sistema software.
- *Immobility*, una progettazione è immobile quando essa contiene in un unico fulcro un insieme di *dipendenze* utilizzate dalla maggioranza delle componenti del sistema. Tuttavia, l'intento di suddividere le *dipendenze* in sezioni differenti potrebbe essere molto pericoloso ai fini della progettazione effettuata fino ad ora.
- *Viscosity*, a sua volta la viscosità si suddivide in due forme. La *viscosità del software* rappresenta un valore attribuile a differenti effettività, le quali sono accomunate da due vie implementative; la viscosità è minima qualora modifiche apportate preservano la progettazione, mentre la viscosità è elevata qualora variazioni comportano a cambiamenti totali delle linee guida prefissate. Infine, la corrispettiva è definita *viscosità dell'ambiente*, la quale avviene qualora lo sviluppo dell'intero contesto è lento ed inefficiente. Una semplificazione è data dalla fase di testing del codice sorgente, in cui se pochi file appartenenti all'insieme richiedano ora di analisi e di controllo, gli stessi sviluppatori saranno tentati nell'apportare modifiche che richiedano il minor tempo possibile, ma questo non garantisce correttezza dell'implementazione, con un'elevata probabilità che manifesti errori in un futuro prossimo.
- *Needless complexity*, indica la presenza di elementi che attualmente sono del tutto inutili. Ciò avviene frequentemente quando gli sviluppatori tentano di anticipare cambiamenti rispetto a requisiti funzionali, facilitando l'adattamento a possibile modifiche. Nonostante, potrebbe consistere in un *trade-off* tra la spesa sostenuta e l'introito ricevuto, poichè sovrastimati i reali requisiti funzionali requisiti.
- *Needless repetition*, indica la cattiva abitudine di riadattare porzioni di codice già esistenti per propri processi esecutivi, la quale potrebbe essere disastrosa per sviluppi futuri. Tpicamente avviene in situazioni in cui non sia posto un corretto livello di astrazione, in cui attività ed elementi sono condivisi da moltitudini di classi; in questi casi occorre stabilire un *abstract layer* aggiuntivo, implementando *interfacce*.
- *Opacity*, è la tendenza che rende di difficile comprensione il progetto condotto. Il codice sorgente potrebbe essere scritto in maniera chiara ed espressiva, oppure in modo confusionario e convulsionario. Certamente una costante simile potrebbe rappresentare un punto di arbitrio tra una maggiore o minore *readability*, la quale essendo una qualità interna consiste un vantaggio per gli stessi sviluppatori.

Comunque sia, proseguire nell'implementazione e sviluppo di requisiti funzionali comporta da sè ad un peggioramento della qualità del codice, indipendentemente dal conseguimento

dei principi carratteristici.

## Dipedencies

...