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 sintattica, in grado di indirizzare il *design project* sulla strada meno tortuosa e articolata possibile, proseguendo per la realizzazione dei requisiti funzionali.

Nonostante, il fulcro del tema proposto stabilisce quali siano i *parametri* affinchè i costrutti utilizzati fino ad ora possano essere ritenuti validi. 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 escutivo espresso dovrà restituire un valore atteso pari al 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

includono la complessità temporale, lo spreco di memoria o il tempo di esecuzione.

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 impone l'esclusione di accessi non autorizzati di molteplici utenti e di garantire che i dati possano essere estratti in maniera congrua e in linea con le direttive dell'applicativo.

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 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 gli utenti finali. Generalmente sono interessati alla semplicità d'uso del software oppure alla correttezza del prodotto, denigrando la leggibilità del codice oppure la stesura strutturale dell'applicativo.

Tuttavia il piccolo trafiletto 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 di attività interne, per migliorare le prestazioni oppure per correggere difetti.

Definizione Flexibility
Per **flexibility** si intende l'intento in cui sia possibile modificare il sistema software per variazioni 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 adoperati.

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

Fino ad 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 e 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 spessore.

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

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

Oltre ai *modelli qualitativi* spesso si attuano framework affinchè si riesca nell'intento di attribuire un criterio valorizzativo ad un sistema software. L'insieme di strumenti di analisi della qualità del software sono attuati affinchè sia esaudibile il desiderio principale di un qualunque sviluppatore, cioè rendere il prodotto finale *riutilizzabile* e *duttile al 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 descrittiva 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 ciò 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 eseguire, rappresenta una prima fase di una qualsiasi implementazione. Infatti, proseguendo con l'aggiunta e la modifica di ulteriori elementi, essa potrebbe essere viziata da comportamenti negativi, la cui manutenzione diviene sempre più difficile, raggiungendo apici che richiedano 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 ad interrompere la propria operatività per piccole variazioni attuate. Spesso, nuove problematiche sono poste all'interno di sezioni che non hanno collegamenti diretti con aree che abbiano subito 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 punto 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.
- *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 dei requisiti funzionali sia inefficiente. Un esempio è dato dalla fase di testing del codice sorgente, in cui se pochi file appartenenti all'insieme richiedano ore 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 possibili modifiche. Nonostante, potrebbe consistere in un *trade-off* tra la spesa sostenuta e l'introito ricevuto, poichè sovrastimati i reali requisiti funzionali.
- *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. Tipicamente avviene in situazioni in cui non sia adottato 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. Certamente una costante simile potrebbe rappresentare un punto di arbitrio tra una maggiore o minore *readability*, la quale essendo una qualità interna, potrebbe consistere in 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 rispetto dei *design smells*.

Dipedencies

La causa principale della maggior parte dei *design smells* è dovuta ad una errata gestione delle **dipendenze**. Il teorico *Beck*, la cui figura è ricordata poichè fu il primo a sviluppare *metodologie agili*, pone la massima attenzione sulla manipolazione delle *dipendenze*, dove egli

stesso afferma che la chiave dei problemi individuati in un sviluppo software consiste in una erronea amministrazione delle *dipedencies*.

Definizione informale
Una *dipendenza* indica che le modifiche attuate ad un elemento possono causare cambiamenti per un altro elemento del modello.

Come da definizione, una *dipedenza* è un cammino che diffonde cambiamenti, causando un effetto a cascata su buona parte degli elementi del sistema software. Una correlazione simile tende a propagare la necessità di apportare modifiche potenzialmente sull'intera struttura su cui fonda lo sviluppo software.

Trattandosi di concetti legati alla struttura logica del sistema, piuttosto che rifersi ad una concreta realizzazione, si adottano principi per interventi che tendino a scartare la presenza massiccia di *dipendenze*.

SOLID

SOLID rappresenta l'insieme dei principi più utilizzati e rinomati per la gestione delle *dipendenze*. Approcciare a principi simili consente di mantenere la semantica e le sintassi dell'architettura progettuale di elevato livello, manipolando al meglio le possibili dipendenze. Le stesse *dipendenze* non devono essere considerate come l'artefice della disfatta di un'implementazione software, anzi la loro presenza è pressoché fondamentale per la stesura di una soluzione. Occorre quindi individuare il giusto approccio affinchè siano correttamente gestite e adoperate.

SOLID consiste nell'acronimo di cinque principi modellativi, i quali possono essere riassunti come segue:

- *Single responsibility principle*, ammette che una classe, posta all'interno della soluzione, deve avere un unico *asse di cambiamento*. Per cui per asse di cambiamento si intende che la modifica di un'unica classe deve appartenere ad una singola ragione. Ciò si collega al concetto di *responsabilità*; se una classe ha più di una responsabilità, di fronte a modifiche di una di esse potrebbero essere inibite le capacità manifestate. Tutto ciò si traduce nella percezione di un *design smell*, ossia *fragility*.
- *Open-closed principle*, cita che una classe dovrebbe essere aperta ad *estensioni*, ma non a modifiche. Una mancata consecuzione del principio potrebbe provocare *errori a cascata*, ossia data la presenza di dipendenze, un errore all'interno di un elemento causa cambiamenti comportamentali di altri elementi del modello. A titolo semplificativo, la questione posta si potrebbe riassumere come segue: aggiungere una funzionalità alla classe non causa fraitendimenti, mentre se l'azione dovesse richiedere la modifica di metodi già implementati, allora si viola il *principio OCP*.

  La soluzione è data da una caratteristica su cui fondano linguaggi OOP, l'*ereditarietà*. Qualora sia necessario modificare metodi già implementati, si eredita la classe esistente e si aggiungono le funzionalità desiderate. *OCP* rappresenta l'artefice che ha portato alla formulazione del processo denominato *refactoring*, i quali in comune accordo, propon-

gono di ristrutturare l'architettura del sistema per cambiamenti delle classi, piuttosto che adoperare *controlli di flusso*, come l'uso degli *IF* concatenati.

- *Liskov substitution principle*, descrive che la relazione fra classi in una gerarchia deve essere composita da sottotipi. *LSP* è molto affine rispetto al *principio di sostituibilità*, il quale impone che ogni istanza di una *sottoclasse* sia interpellabile dalla *superclasse*. Tale sostituibilità è necessaria ma non sufficiente, poichè rispettare *layer di compatibilità* non garantisce la soddisfazione del principio. La violazione principalmente è data da una deviazione comportamentale, in cui è favorita la struttura sintattica ma non la *behavioral compatibility*.

  Definizione informale
  Se il processo *p(o1)* è una funzione valida per oggetti *o1* della classe T, allora la funzione *p(o2)* deve essere valida per oggetti della classe S dove S è una sottoclasse di T.
- *Interface segregation principle*, illustra come la dipendenza di una classe ad un'altra dovrebbe dipendere dall'interfaccia più piccola possibile. Si ricorda che una dipendenza esprime come un'unica modifica provochi cambiamenti anche per altri elementi del modello. Detto ciò l'obiettivo del principio prevede di attribuire un livello referenziale sensato, senza attuare un numero sempre più crescente di *entità astratte*, come interfacce, le quali porterebbero ad un accrescimento di layer strutturali sempre più articolato e complesso.
- *Dependency inversion principle*, pone un tema aggiuntivo rispetto all'introduzione effettuata tramite *ISP*. Stabilisce che *high level classes* non possano dipendere da *low level classes*. Per *HLC* si intendono tutte quelle entità generali, le quali hanno pochi riferimenti al dominio; mentre per *LLC* si contraddistinguono quei elementi che contengono le funzionalità che saranno poi utilizzate, ossia adoperate dall'utente finale. Per cui tramite il principio è valorizzata la struttura gerarchica delle classi, il quale richiede che siano ben manipolate le dipendenze successive all'implementazione.

Una *layer architecture* pone come scopo principale la semplificazione della struttura del sistema, ma da canto suo, un'errata gestione, garantisce un numero crescente di dipendenze. Proprio per questo si adoperano *astrazioni*, grazie alle quali modifiche di funzionalità non saranno vincolanti per ulteriori elementi. Concludendo, è buona pratica ovviare a dipendenze dirette tra *HLC* e *LLC*, mediante il costrutto delle *interface*, in grado di contraporre uno scudo capace di isolare cambiamenti tra livelli.