Introduzione generale alla progettazione software e ruolo dello sviluppo agile

**Motivazione**

Numero elevato progetti interrotti 🡪Costi pianificazione sproporzionati 🡪Non rispetto costi e tempistiche 🡪Rischio elevato progetti grossi 🡪Difficoltà manutenzione 🡪Problemi di qualità🡪Difficoltà a coinvolgere committente e non soddisfatte🡪Contesto di un problema si modifica🡪Natura del SW🡪Miglior preparazione dello sviluppatore🡪Ruolo dello sviluppatore

**Caratteristiche Agili**

* Necessita di coinvolgere il cliente nelle fasi e software di valore fin da subito e in maniera continua
* Il team può prende decisioni, committenti e sviluppatori lavorano insieme
* La conversazione diretta per comunicare con il team, riflessione su come essere più efficaci
* I requisiti cambiano ma i tempi no, accogliamo i cambiamenti e ritmi costanti
* Sviluppare piccole parti alla volta, semplicità essenziale
* Rilascio frequente di versioni incrementate del prodotto, SW funzionante metro di misura sviluppo
* Concentrarsi su maggior priorità
* Integrare test
* Mantenere approccio collaborativo e ambiente di lavoro motivante
* Approccio diverso a dipendenza dei requisiti
* La continua attenzione all'eccellenza tecnica e al buon design esaltano l'agilità

**Punti di forza**

• Flessibilità nei confronti dei cambiamenti di requisiti • Deliverables regolari e pianificati • Coinvolgimento importante del committente • Fixed budget, fixed timeline • Rischio ridotto

**Modello a cascata**

Analisi Requisiti🡪 Design🡪Code🡪Integrazione🡪 Test🡪Deploy

Prima si analizzano tutti i requisiti 🡪Approccio rischioso e costoso 🡪Per requisiti che cambiano rapidamente è meglio l’agile 🡪“il fattore singolo più determinante di insuccesso”

**Valori**

• Seplicità •Comunicazione •Feedback •Coraggio

**Scrum**

Product Backlog 🡪 Sprint Backlog 🡪 Sprint 🡪 Deliverable

**Debito Tecnico**

Non inserire stime dei costi 🡪 non ripulire il codice 🡪 ad un certo punto bisognerà rimediare 🡪 più si aspetta, maggior sarà l’interesse e maggior probabilità di aggiungerne altro 🡪 importante essere vigili e gestire l’evoluzione 🡪 se non si combatte il debito tecnico si ottiene Legacy

**Come combatterlo?**

Uso frequente di refactoring, Il suo valore è quello di impedire l’accumulo di debito, permettendo una velocità di sviluppo costante e “all’infinito”

**Cliente**

Il cliente può vedere la funzionalità implementata ed eventualmente intervenire, Il software, utilizzabile, può andare in produzione molto presto

• Nel processo predittivo l’unico criterio di qualità è il confronto con il piano del progetto

• Nel processo adattivo, il piano va rivisto ad ogni iterazione. I rischi sono mantenuti ai minimi termini

**Sviluppatore**

* Team sviluppo ben formate ed integrate e sinergia (Fowler)
* Sviluppatore attivo nell’analisi, design, codifica e test
* Prende decisioni tecniche e stima il tempo
* Suddivisione compiti
* Stretto contatto con cliente (business expertise)

Introduzione al test di unità e allo sviluppo incrementale.

**Test** Aumenta fiducia nel proprio codice, aumenta produttività, facilita il refactoring e provvedere ad una qualità SW in minor tempo. Il costo iniziale rimane uguale per i test automatici mentre il costo incrementale nei test manuali.Fare test dettagliati e generali, richiama i test regolarmente, controllo correttezza. La realizzazione di test può anticipare il codice da sviluppare.

**Annotazioni** @Test @Before @BeforeEach @ After @AfterEach @Test(expected = ArithmeticException.class) @Ignore @Test(timeout=1000)

**Incapsulamento** 🡪 Permette di nascondere i dettagli implementativi di una [classe](https://it.wikipedia.org/wiki/Classe_(informatica)). Rendere privato tutte le funzionalità che non sono d’interesse per gli utenti

**Oggetto immutabile** 🡪 Semplice, può trovarsi un solo stato, thread safety, può essere condiviso, necessario un nuovo oggetto ad ogni valore

Integrazione continua

È una pratica di sviluppo, ogni membro ingloba il proprio lavoro il più frequentemente possibile, ogni integrazione viene verificata tramite test e build automatica

Aiuta a ridurre al minimo i problemi di integrazione, permette uno sviluppo più rapido, aiuta a ridurre i rischi di integrazione.

Regole

* **Un solo repository** 🡪 dev’essere possibile eseguire un check out su una macchina “vergine” ed poter subito eseguire un build
* **Build Automatico 🡪** Build significa trasformare i sorgenti in un sistema funzionante
* **Build Self Testing 🡪** verificare il funzionamento tramite la chiamata ai test nel processo di build
* **Integrazione ogni giorno 🡪** Commit frequenti
* **Build veloce** 🡪 feedback veloce
* **Clonare Ambiente 🡪** Avere a disposizione ambienti differenti
* **Visione dello stato 🡪** Visione grafica di quello che è lo stato attuale
* **Deployment automatico**

Benefici

* Minor rischio
* Trovare errori
* Deployment più frequement

Design Pattern

Sono modelli che descrivono soluzioni semplici ed eleganti ad uno specifico problema della OO

Tipo di controllo

Static Check (Java) 🡪 evita problemi

173

Refactoring

Senza il refactoring il software tende a decadere, molti cambiamenti nel codice portano ad una incomprensione del codice e della sua struttura. Un regolare refactoring porta ad un mantenimento di quest’ultimo. Inoltre è utile in una futura rilettura del codice. Migliora la comprensione del proprio codice e permette una miglior ricerca di bug. La legge di Le Blanc dice che se una roba (test, bug, leggibilità) non si fa subito non si farà mai

**Conclusione** : passare il tempo a mantenere pulito il tuo codice non è solo economico: è una questione di sopravvivenza professionale

**Entropia del software** : Ogni volta che si modifica un programma la sua complessità aumenta a meno che si tenga la crescita sotto controllo

**Accoppiamento**: Per accoppiamento o dipendenza, in informatica e ingegneria del software, si intende il grado con cui ciascuna componente di un software dipende dalle altre componenti.

**Coesione**: per quali e quanti compiti una classe (o un metodo) è stata disegnata

**Builder**

È una classe nella quale mette a disposizione una build per poter ritornare l’oggetto costruito corrispondente. Ricorda Computer, ComputerBuild.build(this)

**Singleton**

Permette di ottenere una sola istanza tramite un metodo statico getInstance.

**Adapter**

Adapter permette di utilizzare un adapter per collegare due o più interfaccie per lavorare solo su una. Ricorda Mp3 + VLC

**Composite**

Pensa ad un albero dove ogni intersecazione corrisponde ad un composite (il quale ha una lista contenente le foglie) e in printAll tramite un iteratore all’interfaccia comune (component) anche con le foglie le stampa. Pensa Manager 🡪 impiegati

**Decorator**

Permette di ottenere varie funzionalità nelle varie classi, tutte fanno parte di un interfaccia comune

**Observer**

La classe observable ha al suo interno add(Observer o), remove(Observer o) e notifyObserver() nel quale ad ogni richiamo di notifyObserver verrà chiamata la corrispettiva update della classe che implementa Observer

**Strategy**

L’interfaccia strategy permette di avere più implementazioni di un procedimento nel quale tramite una classe di appoggio (context) setto la strategia utilizzata e la eseguo. Nel main utilizzo il context

**Template**

Tramite una classe astratta , creo un metodo final nel quale farò eseguire quelle azioni. In modo che nel main quando richiamo game.play(), mi eseguirà start() ed end() (implementate nelle rispettive classi)

**Proxy**

Tramite una classe intermedia (Proxy) eseguiamo le operazioni tramite un'altra classe se soddisfano i requisiti. Le due hanno la stessa interfaccia.

**Abstract Factory**