Progettazione del Software (Teoria)

# Modello Waterfall

Il tradizionale modello Waterfall:

* identifica le fasi e le attività
* collega una fase alla successiva in modo unidirezionale
* non si può retrocedere a una fase precedente
* standardizza gli output di ogni fase

# Studio di fattibilità

Produce un documento che descrive preliminarmente un problema, gli scenari descrivendo le possibili soluzioni, i costi di progettazione e le finestre di tempo dedicate ad ogni alternativa.

In pratica lo studio di fattibilità è soggetto ai termini dentro ai quali consegnare il progetto: tempo e costo.

Punti da analizzare:

* per quale ambiente deve essere impiegato il software
* identifica i requisiti
* definizione delle specifiche per il software
  + requisiti di interazione con l’utente
* produce il RASD (Requirements Analysis and Specification Document) che deve essere:
  + preciso
  + completo
  + coerente
  + capibile
  + modificabile

# Design

Definisce l’architettura del software attraverso i componenti (moduli), le relazioni e le interazioni tra questi componenti.

Lo scopo del design è supportare lo sviluppo concorrente (condiviso) per separare le responsabilità e produce il Design Document.

# Programmare e test delle unità

Ogni modulo viene implementato utilizzando un linguaggio di programmazione a scelta e viene testato isolatamente dal programmatore del modulo. Il programma deve includere la sua documentazione.

# Integrazione e sistemi di test

I moduli sono integrati in (sotto)sistemi e ogni (sotto)sistema viene testato.

Questa fase e la precedente possono essere integrate in uno schema di implementazione incrementale.

Il test completo del sistema è necessario per verificare tutte le proprietà.

Alcune volte abbiamo un alpha test e un beta test.

# Mantenimento

Include diversi tipi di cambiamento del codice: correzione ed evoluzione.

* mantenimento correttivo e richiede il 20% delle risorse
* mantenimento adattivo e richiede il 20% delle risorse
* mantenimento perfettivo e richiede il 50% delle risorse

# Raccolta di errori

La raccolta di errori sistematica scopre circa il 50%-70% degli errori.

Più grande è il progetto più è facile trovare errori e diventa più costoso correggerli.

Nonostante la risoluzione dei problemi ciò può causarne ulteriori.

I sistemi di grandi dimensioni tendono a stabilizzarsi su un certo livello di errori tollerabili.

# Evoluzione

E’ necessario preparare il programma in modo tale che sia facile modificarlo in seguito a causa di evoluzioni nell’ambito in cui è utilizzato. Esempio: cambio valuta.

# Cambiamenti software

Per ottenere una buona progettazione bisogna prima modificare il design e in seguito cambiare le implementazioni. E’ necessario anche applicare la modifica a tutti i documenti del codice.

Il codice deve essere semplice da modificare.

La manutenzione non si può mai anticipare e programmare e questo in genere è causa di disastri.

# Waterfall

Nella progettazione a cascata è necessario ci sia trasparenza e continua comunicazione con il committente del progetto in modo tale da avere momenti di verifica per ogni fase di progettazione, così da non avere l’effetto “black box” dove si comprende se il software è corretto e soddisfa i requisiti solo alla fine dell’intera stesura del codice causando costi esagerati e tempo sprecato.

# Lo Standard ISO9126

La qualità è definita in termini di sei caratteristiche

* **Funzionalità**: riguarda il che cosa deve fare una applicazione. L’insieme di attributi che riguardano l’esistenza di un complesso di funzioni e delle specifiche proprietà. Le funzioni sono quelle che soddisfano i requisiti stabiliti o impliciti.
* **Affidabilità**: l’insieme di attributi che riguardano la capacità del prodotto software di mantenere il livello di prestazioni con condizioni e limiti di tempo fissati.
* **Usabilità**: l’insieme di attributi che riguardano lo sforzo necessario all’uso del prodotto e alla valutazione individuale di tale uso relativamente a un insieme di utenti prefissato o implicito.
* **Efficienza**: l’insieme di attributi che riguardano la relazione esistente tra il livello delle prestazioni del prodotto software e la quantità di risorse necessarie nell’ambito di condizioni fissate. Tipiche risorse: spazio di memoria, tempo e occupazione della banda.
* **Manutenibilità**: l’insieme di attributi che riguardano lo sforzo necessario ad eseguire modifiche. Correttiva, perfettiva o adattiva.
* **Portabilità**: l’insieme di attributi che riguardano la capacità del prodotto software di essere trasferito da un ambiente all’altro. Differenze tra gli ambienti: organizzativa, software e hardware.

Ogni caratteristica è definita in termini di sotto caratteristiche.

**Sotto caratteristiche**

* si possono stabilire limiti di accettabilità per ciascuna misura
* oppure, si possono utilizzare somme pesate per tutte le misure in modo tale da giungere a un valore univoco per una sotto caratteristica di qualità

**Caratteristiche**

* si possono utilizzare somme pesate per tutte le sotto caratteristiche per giungere a un valore univoco per una caratteristica di qualità

**Qualità**

* si possono utilizzare somme pesate per tutte le caratteristiche per giungere a un valore univoco per la qualità

Classi e oggetti

Una classe è un contenitore che ha attributi, che vengono anche detti campi o variabili, e i metodi che possono essere funzioni o procedure.

A cosa servono le classi? Le classi fanno parte del Object Oriented Paradigm ovvero che tutto quello che fa parte di un software può essere ricondotto a un oggetto in modo più o meno astratto.

Con una classe in genere si cerca di celare la progettazione e la sua implementazione. In sostanza importano solo i valori che restituisce e come richiamarla.

Per i dati si tende a utilizzare private, mentre per le funzioni in genere si indica public. (modificatore di visibilità)

Questo tipo di paradigma ci offre dei vantaggi:

1. Ogni programmatore deve pensare alla implementazione della singola classe;
2. Possiamo modificare il codice solamente di quella specifica classe e il tempo usato per la manutenzione è minore.

Il modificatore abstract indica che è stato definito un metodo, ovvero cosa prende in ingresso e cosa restituisce in uscita, ma non il corpo il metodo (il codice). Mentre se il metodo viene implementato classicamente non c’è bisogno di indicare che è completa l’implementazione (perché ho indicato tutto nel codice del metodo).

I dati possono essere di due tipi:

1. Astratti ovvero che sono complessi (per esempio classe);
2. Concreti (carattere, intero, long, ovvero le variabili primitive ma non quelle per riferimento come String).

Anche una classe definisce un tipo, cioè una aggregazione di dati! Anche i metodi vengono considerati dei dati perché si possono richiamare tramite nome.

Differenza tra classe e oggetto:

La classe è il progetto dell’oggetto mentre l’oggetto è l’istanza, cioè quello che è stato creato all’interno della classe.

N.B. Non tutte le classi possono essere istanziate 🡪 Le classi astratte non hanno metodi implementati e quindi non possono essere istanziate. Le classi astratte sono una impalcatura. (es. modelli di word… Sono una impalcatura che deve essere implementata).

I metodi più comuni sono: get, set, utility, costruttore personalizzato, costruttore di default.

Il get e il set vengono usati quando hai dei tipi di dati astratti (complessi) oppure quando non hai l’accesso diretto al dato. Se noi abbiamo da esempio da inserire il codice fiscale. Noi non possiamo modificarlo, però possiamo leggerlo tramite il get. In questo caso il set lo potrebbe utilizzare lo Stato.

utility: per esempio prendi un oggetto e lo trasformi in stringa o prendi una stringa e lo trasformi in un oggetto

costruttore personalizzato: è quello che è stato definito e implementato dallo sviluppatore e serve solamente per l’oggetto creato.

costruttore di default: è definito dalla classe madre e in genere non ha parametri. Quando viene definita una classe essa è sottoclasse di Object o della classe madre che è quella direttamente superiore.

N.B. Quando chiami il costruttore chiama ricorsivamente tutti i costruttori fino a Object.

Identificatori:

Il metodo finalize l’implementazione è opzionale e serve per indicare delle porzioni di codice da eseguire che viene eseguito all’eliminazione della classe.

this indica letteralmente questa classe mentre se faccio this.x indico il parametro x della classe stessa.

static indica che la classe definita nel codice sorgente è già “compilata”, mentre se non è stato usato l’identificatore bisogna generare ogni volta la classe.

Ereditarietà:

Quando tu hai due classi molto simili, per non scrivere due volte lo stesso codice, si utilizza l’ereditarietà così da evitare errori frequenti. In questo modo viene indicata una classe con dati in comune e da quella le sottoclassi ereditano questi dati. La classe principale ha delle operazioni di base e le sottoclassi hanno le stesse operazioni e aggiungono funzionalità extra e/o ridefinire delle funzioni.

Sintassi:

class SottoClasse extends SuperClasse //(semantica extends 🡪 eredita da)

{

…

}

Si usa super per richiamare le funzioni della classe precedente (madre).

Le classi astratte possono avere figli e si consiglia di implementare il prima possibile questi metodi astratti.

L’ereditarietà multipla è una fonte di problemi per via del fatto della copia di tutti i parametri dei metodi. Per questo in Java si è aggirato questa problematica con le interfacce.

Una classe può avere solo una classe madre, ma può avere più interfacce.

Le interfacce sono come le classi ma tutti i loro metodi sono astratti e possono essere ricondotti a delle specie di contratti. L’interfaccia è di per sé incompleta, ma quando viene ereditata le sue funzioni devono essere per forza implementate, mentre le variabili definite all’interno di una interfaccia sono tutte final ovvero non possono essere modificate (sono costanti).

Design

Questa attività serve per definire l’architettura del software definendo i sistemi in componenti e interazioni che possono essere definiti in due livelli di astrazione. Il livello più alto è per esempio quello del sistema operativo dove ho le interfacce, poi ho il livello delle macro categorie di sistema e così via….

Dobbiamo conoscere quali sono i criteri per decomporre il sistema in moduli.

Documentazione: la documentazione è un catalogo di moduli e relazioni che sono sinonimo di componenti e interazioni.

Esempio: componenti possono essere client e server, mentre le relazione possono essere le azioni che possono eseguire fra di loro ( il client contatta il server e il server risponde).

I meccanismi descrivono come l’architettura è strutturata, mentre lo stile descrive le differenti caratteristiche tra un’architettura e l’altra. Per ogni livello di design bisogna lasciare del tempo per poter ragionare sul nuovo livello.

Un modulo è la parte di sistema che fornisce vari servizi (funzioni). I servizi di un modulo sono l’interfaccia, quest’ultima definisce il contratto tra il modulo e l’utente.

Parole chiave dei diagrammi use case:

* USES (usa): un modulo usa i servizi esportati da un altro;
* IS COMPONENT OF (è componente di): descrive un gruppo di moduli come se fosse un modulo di un livello più alto;
* INHERITS (eredita): viene usato per sistemi Object Oriented.

Esistono dei principi: massimizzare la coesione e minimizzare le relazioni, nascondere le informazioni (incapsulamento = ciò che non deve essere modificato/utilizzato da altri, non deve essere visto).

Concetti chiave design:

* Decomposizione;
* Astrazione;
* Incapsulamento;
* Modularità;
* Estensibilità;
* Gerarchia.

Questi concetti servono a gestire la complessità del software, migliorarne la qualità e facilitare il riuso di codice.

(per la sezione design è bene guardare anche gli esempi per comprendere a pieno.)