Ingegneria dei software

kanopo

12 aprile 2022

Indice

Elenco delle figure

Elenco delle tabelle

1 T1 - Software Development Process

L'Ingegneria del Software non è solo scrivere codice è piuttosto un concetto di risoluzione dei problemi del mondo reale sfruttando il software.
I requisiti sono sempre più stringenti: tempi brevi, sistemi complessi, molte features.
Un buon software deve avere ottime **maintenability, dependability, efficiency, acceptability**.
I problemi e le soluzioni sono sempre complesse ma il software permette massima flessibilità. È un sistema discreto.
Le sfide principali sono rappresentate da **eterogeneità, delivery, trust**.
La fase di problem solving si suddivide in analisi e sitesi.

1.1 Cos'è l'ingegneria del software?

L'Ingegneria del software è un **insieme di tecniche, metodologie, strumenti** che aiutano nella produzione di software di alta qualità dati un budget, una scadenza, e delle modifiche continue.
La sfida principale è quelli di aver a che fare con complessità elevate e ad un aumento delle responsabilità, dato che un ingegnere del software non deve solo scrivere codice, ma piuttosto lavorare con competenza e confidenzialità, attenendosi ad un'etica.

1.1.1 Processo del software

Dopo una rappresentazione astratta, si procede con un set di attività strutturate: specifiche dei **requirements, design, implementazione, validazione, evoluzione.**

1.2 Modelli di sviluppo del software

Distinguiamo tra **plan-driven** e **agile** development. Nel primo si pianificano i requisiti e solo in seguito si sviluppa il software. Nel secondo si sviluppa il software un pezzo alla volta, a stretto contatto con il cliente per dei feedback.

1.2.1 Modello a cascata

Modello plan-driven, le specifiche e lo sviluppo sono separati.
I pro:

- ottima documentazione
- manutenzione semplice

I cons:

- specifiche congelate dopo la pianificazione iniziale
- cliente poco coninvolto
- tempi lunghi

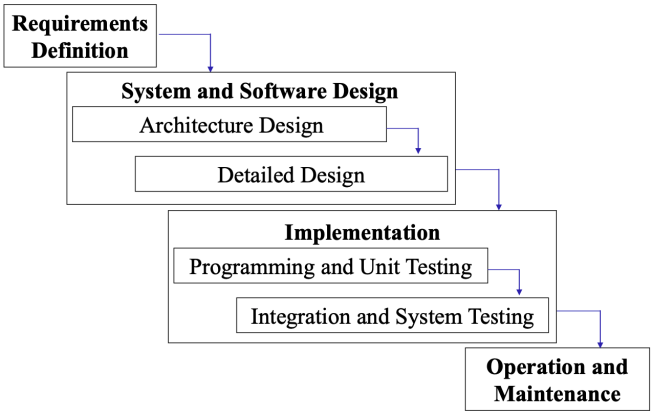

Figura 1: Modello a cascata

1.2.2 Modello a spirale

Diverse fasi che susseguono a spirale, la gestione del rischio viene gestita tramite prototipazione che permette di testare i prodotti.
I pro:

- prevenzione dei rischi
- completezza della documentazione
- flessibilità
- elevata usabilità
- buon design
- facilità di manutenzione
- ridotto costo di sviluppo

I cons:

- modello costoso
- riservato ad esperti e a progetti costosi

Il prototipo è un'implementazione limitata del sistema, rappresentanti solo alcuni aspetti alla volta.

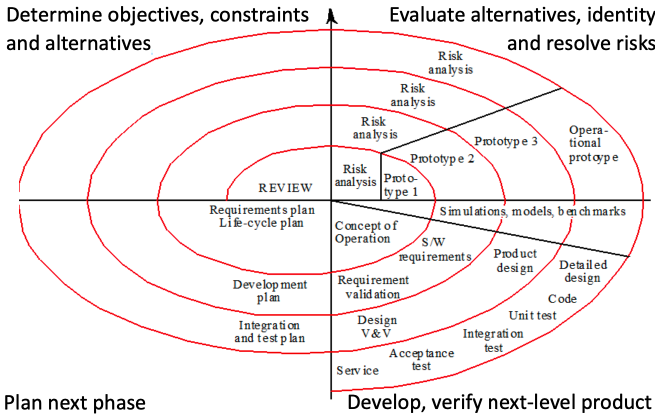

Figura 2: Modello a spirale

1.2.3 Sviluppo incrementale

Modello che si suddivide in fasi:

1. raccolta dei requisiti
2. versione iniziale
3. fase di design
4. fase di implementazione
5. produzione di versione finale

I pro:

- naturale presenza di prototipi ad ogni aggiunta di features
- basso rischio di fallimento
- qualità di testing in base alla priorità

I cons:

- scarsa visibilità d'insieme
- sistemi mal strutturati
- skill speciali necessarie

Adatto a progetti piccoli o parti di progetti grandi.

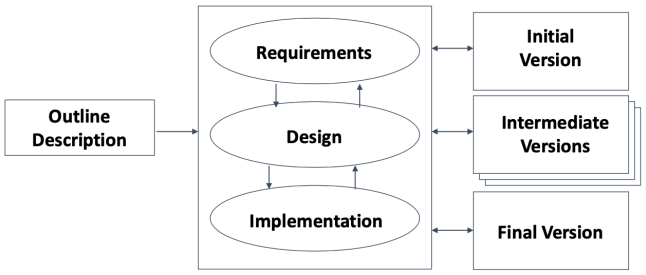

Figura 3: Modello a incrementale

1.2.4 Sviluppo guidato dai test

Vengono prima scritti i test e poi l'implementazione mettendo le difficolta in primo piano.
Rende il debug più semplice.
Si aggiunte il nuovo test e poi la features cosi da veriicare che tutti i test precedenti siano validi.

1.2.5 Sviluppo agile

Questo modello di sviluppo è basato sul concetto di delivery del prodotto continuo avendo delle features in continua evoluzione.
Il cliente è al centro dello sviluppo, il team si autoorganizza e le features vengono aggiunte man mano.
Essendoci la mancanza di planning il team deve essere esperto per non perdersi.
Spesso si creano documentazioni sbagliate o incomplete.

1.2.6 Extrem programming

L' XP viene scelte quado i requisiti cambiano velocemente, i team sono ridotti e affiatati(pair programming per esempio).
Tipo di programmazione agile, basato sul design semplice, release minori, refactoring continuo, alta semplicità.

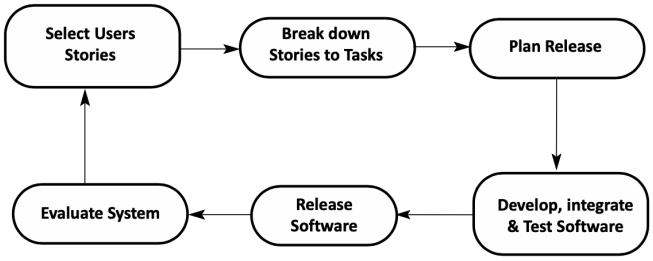

Figura 4: Modello a extrem programming

1.3 Software riusabile

È comodo lavorare per **microservizi atomici**, di modo da usarli in seguito se si affronta un problema simile, un'esempio sono le API, è sempre meglio riusare qualcosa che riscriverlo da zero.
Questo "*mindset*" riduce i temi e i costi di sviluppo nel luno periodo.
Il contro è il fatto che i microservizi non sono fatti espressamente per una task specifica e si va a sacrificare qualche requisito.

2 T2 - Coding, Debugging, Testing

Lo stile di coding è importante perchè un programma viene scritto una volta e letto spesso da altri!
Stare a ttenti ai layout, nomi, commenti, ecc...

2.1 Legge di Ambler per gli standard

Più uno standard è adottato e più sarà facile farlo usare al proprio team.
Non perdere tempo adietro a standard scemi.
Se hai dei dubbi usa gli standaard di google che sicuramente ne sanno più di me e te. **standard di google**

Solitamente si hanno degli standard aziendali e questi standard vanno a chiarire aspetti come:

- nomenclatura
- formattazione
- formato
- contenuto

2.2 Coding practices

- indentazione
- whitespaces
- naming, commenting

Fondaentale è spiegare i compiti delle classi, funzioni e procecssi complessi.
Utilizzare uno standard permette ai collechi di non dover riscrivere il codice. Rendendo più semplice le iterazioni del codice successive.

2.3 Gestione degli errori

Si fa un distinguo fra il prima, durante e dopo:

- prrevenzione
- rilevamento
- recupero

Per debuggare, bisogna riconoscere l'errore, isolare la fonte, identificarne la causa, trovare un fix, applicarlo e testarlo.

2.4 Testing

Permette di trovare errori ma non la loro assenza, **il tester non dovrebbe essere il programmatore**.

2.4.1 Unit testing

Test di singoli componenti.

2.4.2 Integration testing

L'intero sistema è visto come insieme di sottositemi, l'obbiettivo è quello di testare tutte le interfacce e le interazioni fra i sottositemi.

2.4.3 System testing

test dell'intero sistema per vedere se rispetta i requisiti funzionali.

2.4.4 Functional testing

Verifica delle funzionalità del sistema, test basati sui requisiti del progetto.

2.4.5 Performance testing

Test in situazioni estreme(stress testing, volume testing, recovery testing, penetration testing).

2.4.6 Acceptance testing

Il sistema è pronto per la produzione??
Test scelti ed effettuati dal cliente.(alpha e beta test)

3 T3 - System Modelling and UML

Rappresentazione astratta del sistema e dei problemi, si dedvono introdurre i componenti essenziali mediante una noazione consistente.
Il system modelling deve essere **predictive**(prima del development), **extracted**(da un sistema esistente) e **prescriptive**(definire regole per l'evoluzione del software).

3.1 UML

UML è semplice, espressivo, utile, consistent, estensibile.

4 T4 - Requirements engineering

Gli scopi dell'ingegnerizzazione dei requisiti è **identificare** i servizi necessari e i constrait, **definire** offerta e contratto, **ottenere** tutte le informazioni necessarie al design.
Si cerca di ottenere requirements:

- validi(reali necessità)
- non ambigui
- completi
- comprensibili
- consistenti
- prioritizzati
- verificabili
- modificabili
- tracciabili

4.1 Classificazione dei requirements

4.1.1 Requisiti funzionali(features di sistema)

Descrivono funzionalità di sistema o di servizio, come input di dati, output, operazioni svolte, workflow, autorizzazioni.

4.1.2 Requisiti non funzionali(features di sistema)

Descrivono i limitti di parti del sistema e del suo sviluppo.
Specificano criteri per giudicare l'operato del sistema.

4.1.3 Requisiti del dominio(features di sistema)

Derivato dal campo di utilizzo del software.

4.1.4 Requisiti volatili(natura statica/dinamica)

- mutable requirements = cambiano nel tempo(tasse, nomative, ecc)
- emergent requirements = cambiano quando il cliente capisce di più il sistema
- consequential requirements = emergono con l'informatizzazione del sistema che non lo era
- compatibility requirements = emergono dal dover interfacciare il sistema con altri sistemi

4.2 Rischi

I rischi della stesura dei requisiti possono essere:

- imprecisioni
- conflitti tra requisiti

4.3 Documento di specifica dei requirements

Questo documento specifica i requisiti del sistema, includendo una definizione e una specifica.
Prende il nome di system specification se include direttive su harware e software.
Software Requirements Specification (**SRS**) se include solo specifiche software.
Un **SRS** deve includere:

- introduzione
- descrizione generale
- features
- requirements

Il linguaggio naturale usato per stilare questo documento implica ambiguità, per questo si ricorre a una struttura ben definita per evitare ambiguità.

5 T5 - Requirements ecgineering and UML

5.1 Use case diagram

In questo diagramm vengono inclusi tutti i casi di utilizzo del sistema, da parte dei vari attori che rappresentano gli utenti.
Il **System boundary** divide l'interno dall'esterno del sistema.

5.2 Class diagram

Diagramma che rappresenta le classi tramite:

- gli attributi(pubblici con +, privati con - e # per i protetti)
- le funzioni delle classi

5.2.1 Associazione

Quando una classe svolge il ruole di variabile all'interno di un'altra classe, questa connessione deve essere rappresentata, si usano:

- cardinalità
- direzione
- constraint
- ruoli

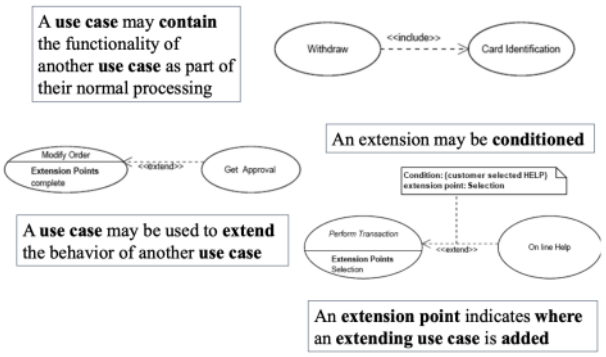

Figura 5: Associazioni

5.2.2 Gneralizzazione e nesting

La generalizzazione indica **eretidarietà**, nesting indica che una classe è nestata nella classe dove arriva l'operatore.

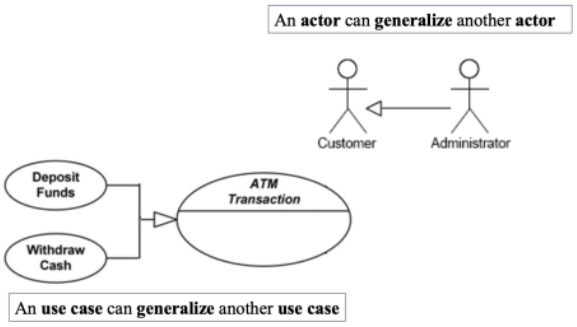

Figura 6: generalizzazione e nesting

5.2.3 Dipendenza e realizzazione

- **dipendenza** = relazione debole tra client e supplier
- **realizzazione** = relazione tra specifica e implementazione

5.2.4 Aggregazione e composizione

L'aggregazione rappresenta un elemento composto da altri elementi minori.

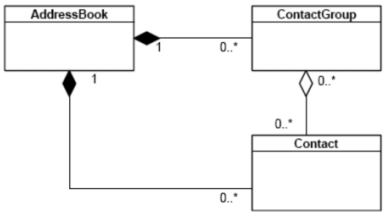

Figura 7: Aggregazione e composizione

5.3 Sequence diagram

Usa un timeline verticale per rappresentare l'interazione tra le classi.
Ci possono essere comunicazioni sincrone e asincorne.

5.4 Activity diagram

Diagramma che rappresenta un'operazione eseguita nel sistemasa con rappresentazione dell risorse utilizzate(io, disk, banda, ecc)

5.5 Robustnes diagram

Diagramma UML semplificato che appresena gli use cases verificandone la correttezza, completezza e requisiti.

6 T6 - Feasability and requirements elicitation

Per ottenere le informazioni necessarie dobbiamo identificare le fonti, acquisire i dati, verificarli e sintetizzarli.
Le varie persone interessate al progetto possono essere suddivise cosi:

- stakeholder: interessato nel progetto
- developer: produce con pochi errori
- project managment: budget, scadenze

- investor: velocizzazione del progetto
- cliente e utente: usabilità e workflow

6.1 Tecniche di elicitation

(elicitation = tirare fuori le informazioni)

6.1.1 Document analysis

Analizzare i documenti e insieme agli stakeholder verificare la correttezza dei dati.

6.1.2 Observation of the work environment

Apprendere informazioni importanti osservando il lavoratore.

6.1.3 Questionario

Determinare i fatti e leopinioni con un questionario.

6.1.4 Interviste

Decidere chi intervistare e acquisire i dati.

6.1.5 Scenari e casi di utilizzo

Esempi IRL(In Real Life) di come il sistema verrà usato.

6.2 Attività di supporto all'elicitation

6.2.1 Brainstorming

composto dalle fasi storm(spara idee) e calm(analizza idee)

6.2.2 Focus group

simile al brainstorming

6.2.3 Prototipi

Esecuzone di una task e si analizza lasituazione facendo emergere problemi e soluzioni.

7 T7 - Use cases

Gli use cases sono scenari plausibili che sfruttano i diagrammi UML.
Ogni requirements deve essere mappato da almeno un use case.

7.1 Componenti principali

Lo use case deve definire lo stato precedente del sistema, l'ordine degli eventi, le alternative, situazioni eccezionali e i risulati. Deve menzionare gli attori che prendono parte al sistema, i problemi di design, i diagrammi di relazione.
Ecco delel guidelines:

- non pensare all'implementazione
- essere pessimistici
- elencae gli scenari funzionanti
- elencare tutti i possibili use cases
- utilizzare un formaato standard
- utilizzare verbi appropriati
- documentare bene le situazioni eccezionali
- non rappresentare singoli step come use cases

Name	The Use Case name. Typically the name is of the format <action> + <object>.
ID	An identifier that is unique to each Use Case.
Description	A brief sentence that states what the user wants to be able to do and what benefit he will derive.
Actors	The type of user who interacts with the system to accomplish the task. Actors are identified by role name.
Organizational Benefits	The value the organization expects to receive from having the functionality described. Ideally this is a link directly to a Business Objective.
Frequency of Use	How often the Use Case is executed.
Triggers	Concrete actions made by the user within the system to start the Use Case.
Preconditions	Any states that the system must be in or conditions that must be met before the Use Case is started.
Postconditions	Any states that the system must be in or conditions that must be met after the Use Case is completed successfully. These will be met if the Main Course or any Alternate Courses are followed. Some Exceptions may result in failure to meet the Postconditions.
Main Course	The most common path of interactions between the user and the system. 1. Step 1 2. Step 2
Alternate Courses	Alternate paths through the system. AC1: <condition for the alternate to be called> 1. Step 1 2. Step 2 AC2: <condition for the alternate to be called> 1. Step 1
Exceptions	Exception handling by the system. EX1: <condition for the exception to be called> 1. Step 1 2. Step 2 EX2 <condition for the exception to be called> 1. Step 1

Figura 8: Tamplate per i casi di utilizzo

8 T8 - Requirements

L'analisi dei requisiti permette di raffinare e strutturare i requisiti in modo da renderli più chiari, precisi e formali.

8.1 Analysis classes

Il concetto è quello di astrarre le entità del problema.

8.2 Classes discovering techniques

8.2.1 Noun verb analysis

Analisi che sfrutta i contenuti delle specifiche del progetto. Necessita di completezza del documento molto alta.

8.2.2 Use case driven approch

Approccio che sfrutta fli scenari degli use cases.

8.2.3 Common class patterns

Analisi basata sulla teoria della classificazione generica degli oggetti.
Fornisce linee guida, ma non un processo sistematico per ottenere le classi.
Introduce errori di mal'interpretazioni.

8.2.4 CRC cards

Sono usate in specifiche sessioni di brainstorming, generalmente si parte dagli use cases e s icreano delle card con:

- class name
- responsabilities
- collaborators

Non è un metodo sistematico, si usano le CRC cards come validazione.
Gli achievement di questo metodo sono:

- Verifica la correttezza dello use case
- verifica la correttezza delle associazioni
- verifica la correttezza delle generalizzazioni

- trovare le classi omesse
- scovare opportunità di refactoring

8.2.5 Mixed approch

Migliore:

1. Le classi iniziali provengono dalla conoscenza del dominio
2. si sfrutta come guida il common class pattern
3. per aggiungere altre classi si usa la noun verb analysis
4. per verificare il lavoro si usano gli use cases
5. per il brainstorming si usano le CRC cards

9 T9 - Requirement validation and managment

9.1 Valdation

La validazione consiste nella verifica della correttezza dei requisiti, con obbiettivi la completezza e i costi.

- consistenza
- realismo
- verificabilità

9.2 Requirements managment

Si verificano gli errori, conflitti e inconsitenze.
Soddisfare il cliente e mantenere i costi bassi:

- identificazione dei requisiti
- processo di modifica
- policy di tracciabilità
- supporto del case tool

9.2.1 Case tool support

non ho voglia di spiegarlo.

9.2.2 Stati dei requirement

tutto questo ambaradam è molto simil ad un issue tracker come git.

- proposal
- approved
- rejected
- implementation
- verified
- deleted

Si fanno sostanzialemnte dei test con un version control e si fa il roll back in caso di problemi.

9.2.3 Tracciabilità

Anche questo discorso viene gestito bene da un software tipo git.

9.2.4 Traciability planning

Elemeti fondamentali del planning:

- tipi di stakeholder
- informazioni richieste
- dove e da chi raccogniere info
- dove e come mantenere le info
- dove e come cercare le info

Ci sono delle limitazioni per il plannign:

- numero di requirements
- lifetime stimato
- livello di maturità dell'azienda
- dimensione progetto
- altri vincoli dell'utente

12 T12 - Object constraint lenguage

Fa parte delle specifiche UML, e segue uno standard di IBM, è un linguaggio formale usato per descrivere i modelli Object oriented.

12.1 Model type

sono classi, subclasses, classi di associazione, interfacce ed enumerazioni.
Hanno proprietà e attributi.

12.2 Operazioni, espressioni, constraint

I constraint sono dei valori booleani che veificano una condizioni, servono a porre limitazioni sul tipo di valore che può avere una classe.

- Migliorano la documentazione
- più precisione
- comunicazione senza ambiguità

Vari tipi di constraint:

- invarianti(quando l'istanza è a riposo)
- precondizionali
- postcondizionali
- guard(vera prima e dopo)

13 T13 - Design process

Processo che permette di capire come implementare il sistema.
Si parte dai requirements e si tiene conto dei design general princilas.

13.1 Linee guida

- esibire organizzazione modulare che fa uso intelligente di controllo tra componenti
- partizione in componenti logici
- descrivere sia dati che procedure
- arrivare ad interfacce che riducono la complessità

Un buon design deve implementare tutti i requirement espliciti del modello di analisi, e tutti quelli impliciti desiderati dal cliente. Dev'essere leggibile e comprensibile, sia per chi scrive codice, sia per chi lo testa e supporta. In generale, dovrebbe dare un'immagine generale del software, indicandone i dati, i domini funzionali e behavioral dal punto di vista dell'implementazione.

13.2 Stadi del design process

1. comprendere il problema
2. identificare una o più soluzioni
3. processo iterativo ed incrementale

13.3 Design as series of decisions

Alcuni tradeoff possono essere:

- funzionalità vs usabilità
- costo vs robustezza
- efficienza vs portabilità
- velocità di sviluppo vs funzionalità
- costo vs riutilizzabilità

Gli step per una decisione pesata sulle priorità sono:

1. Elencare le alternative possibili
2. elencare i pro e i contro di ogni alternativa(rispetto agli obiettivi e alle priorità)
3. determinare se alcune alternative impediscono altri obiettivi
4. scegliere l'alternativa che permette di raggiungere più obiettivi
5. aggiustare la priorità per decisioni future

13.3.1 Approcci

Approccio **top-down**: prima si fa il design ad alto livello del sistema, poi si prendono le decisioni man mano che si sviluppa il sistema.
Approccio **botton-up**: prima si decidono le utilities di basso livello riutilizzabili, poi il modo di integrarle insieme.
Approccio **mixed**: si usa top-down per scegliere la struttura ed il bottom-up per i componenti riutilizzabili.

13.3.2 Attività, rischi e obiettivi

La fase di design ha tre fasi:

- design dell'architettura
- user interface
- class design

Un buon design è **flessibile, dettagliato, ben documentato e buona possibilità di managment**.

14 T14 - Design concepts

Un sistema sftware ha diversi possibili problemi:

- rigidità
- fragilità
- immobilità(difficle riusare i componenti)
- viscosità(più semplice l'hack del mantenimento del design originale)

14.1 Software design concepts

14.1.1 Abstraction

L'astrazione permette di focalizzarsi su parti importanti del software senza perdersi nell'implementazione. Permette la descrizione di un sistema come struttura a livelli.

14.1.2 Refinement

processo top-down iterabile che ad ogni passo trasforma istruzioni generiche in istruzioni più specifiche.

14.1.3 Nascondere le informazioni

L'information hiding è strettamente legato a:

- astrazione
- coupling
- coesione

14.1.4 Modularità

Un metodo di design può essere detto modulare solo se supporta:

- decomposability(ridurre in modulli più piccoli)
- composability(accorpare in un unico modulo)
- understandability(un modulo è comprensibile senza sapere gli altri)
- continuity(la modifica in un modulo modifica solo lui)
- protection(bug in un modulo?? affligge solo qul modulo!)

I principi per il design modulare sono: linguistic modular units (i moduli devono corrispondere alle unità del linguaggio, come pacchetti o moduli), poche interfacce, poco scambio di informazioni tra moduli, interfacce esplicite (se due moduli comunicano, dev'essere ovvio), information hiding(tutte le informazioni di un modulo dovrebbero essere private, se non specificatamente dichiarato il contrario, e per l'accesso bisogna utilizzare interfacce).

14.1.5 Coesione

La coesione misura la chiusura di una relazione tra elementi di un componente/classe. Una coesione forte è desiderabile perché semplifica le correzioni, le modifiche, le estensioni, riduce il testing e promuove il riutilizzo. Abbiamo più livelli di coesione (dal più basso al più alto):

- coincidental(elementi senza relazioni ma uniti per convenienza)
- logical(elementi che fanno cose simili)
- temporal(elementi attivi nello stesso tempo)
- procedural(elementi che compongono una singola sequnza di controllo)
- communicational(elementi con stesso input e stesso output)
- sequential(l'output di uno è l'input dell'altro)
- functional(elementi che soppriscono ad un funtiona lrequirement)
- object(operazioni tra oggetti)

14.1.6 Coupling

Misura l'interconnessione tra moduli. Quando è debole, i moduli sono fortemente indipendenti. Livelli di coupling dal più debole al più forte:

- no direct(nessuna dipendenza)
- data(solo dati necessari)
- stamp(passati una lista di argomenti ma solo alcuni vengono usati)
- control(si passano flags e altri paramentri)
- externl(legato a device o device drivers)
- common(uso di dati globali)
- content(modificadi dati dell'altro modulo)

Con coupling elevato i componenti sono difficili da comprendere autonomamente, le modifiche causano bug in altri moduli e il riuso di moduli fa schifo.
Con coupling basso aumentano i costi di performance ma lo sviluppo ne gode.

15 T15 - Objects oriente design principles

15.1 Principi dell'Object Oriented Design

15.1.1 SRP - Single Responsability Principle

Una classe deve avere una sola ragione di cambiare.
Le modifiche ai requirements si mappano sulle responsabilità.
Avere tante classi con responsabilità separate semplifica e rende il design flessibile.

> *In programming, the Single Responsibility Principle states that every module or class should have responsibility over a single part of the functionality provided by the software.*

15.1.2 OCP - Open Closed Principle

Un modulo deve essere aperto verso l'esterno ma chiuso all modifica.
Meglio implementare cose in classi derivate piuttosto che modificae una classe.

> *In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification. If you have a general understanding of OOP, you probably already know about polymorphism. We can make sure that our code is compliant with the open/closed principle by utilizing inheritance and/or implementing interfaces that enable classes to polymorphically substitute for each other.*

15.1.3 Liskov Substituotion Principle

Le classi derivate devono adempiere alle funzioni ereditate dalle classi padre altrimenti se si usa una classe derivata al posto delal padre si generano errori.

> *More generally it states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

15.1.4 ISP - Interface Segregation Principles

Avere tante interfacce client-specific è meglio di una generica.

> *In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. Put more simply: Do not add additional functionality to an existing interface by adding new methods. Instead, create a new interface and let your class implement multiple interfaces if needed.*

15.1.5 DIP - Dependency Inversion Principles

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello ma dovrebbero dipendere da astrazioni.

15.2 Principi di package cohesion

15.2.1 REP - Release/reuse Equivalency Principle

Per riusare un componente non si fa copia incolla(perchè si perdono le modifiche alla libbreria) ma si inserisce il codice mediante una release della libreria e poi si modifica un'istanza.

15.2.2 CCP - Common Closure Princple

Le classi che cambiano insieme devono rimanere insieme.

15.2.3 CRP - Common Reuse Principle

Le classi che non sono reutilizzabili insieme non dovrebbero essere pacchettizate insieme.

15.3 Package Coupling Principle

15.3.1 ADP - Acyclic Dependencies Principle

No cicli nel grafico delle dipendenze.

15.3.2 SDP - Stabel Dependencies Principle

Ogni volta che un pacchetto cambia, tutti quelli che dipendono da lui devono essere validati.

Ovviamente tu preferisci i pacchetti stabili

15.3.3 SAP - Stable Abstraction Principle

Il pacchetto perfetto è tanto stabile quanto astratto.
I pacchetti astratti dovrebbero essere responsabili e indipendenti, mentre quelli concreti dovrebbero essere irresponsabili e dipendenti.

15.4 Attività per un buon design

- Dividi e conquista(scompore tutto in processi semplici da fare uno alla volta)
- Aumenta la coesione(Un sistema con alta coesione tiene insieme quello che deve stare insieme e lascio fuori il resto)
- Ridurre il coupling(meno interdipendenze tra moduli)
- Aumenta l'astrazione(meglio nascondere i dettagli e ridurre la complessità)
- Aumentare la riusabilità
- Riutilizzare design e codice esistente
- Programmare con l'idea di flessibilità(modifiche future)
- Anticipare l'obsolescenza(pensare alle tecnologie e pianificare la gestione del sistema)
- pensare alla portabilità
- pensare alla testabilità
- sviluppare con l'idea di difendere il software dal'utente.

16 T16 - Design Patterns

Un design pattern è una soluzione ricorrente ad un problema comune, che può essere usata in modi diversi. È dipendente da linguaggio ed implementazione. I design patterns hanno vari obiettivi:

- Codificare un buon design
- dare nomi espliciti alle strutture
- catturare e preservare le info sul design
- facilitare il refactoring

Due forme possibili: **Alexandrian** e **GoF**

16.1 Creational patterns

Creazione flessibile di oggetti(rendendo il sistema indipendente dagli oggetti creati).

16.1.1 Factory method

Scopo: il metodo factory fornisce un'interfaccia per creare oggetti in una subclass, pemettendo di modificare gli oggetti creati.
Problema: Immaginiamo di dover creare un'applicazione che gestisca un servizio di logistica. La prima versione utilizza solo trasporto su ruota, quindi la maggior parte del codice sta nella classe Truck. Dopo un pò, l'app diventa popolare. Decidi di implementare anche il trasporto via mare. Aggiungere la classe Ship richiederebbe modifiche in tutto il codice. Aggiungerne altre, pure.
Soluzione: Il metodo factory permette di sostituire la creazione di nuovi oggetti (con **new**) con chiamate a un metodo factory.
Gli oggetti geenrati così sono dei **prodotti**.
Però i prodotti devono avere l'interfaccia in comune.

16.1.2 Abstract factory

Capito fin li
Scopo: L'abstract factory permette di creare famiglie di oggetti related senza specificare le classi concrete.
Problemo: Immaginiamo di voler creare un simulatore di negozio di arredamenti. Il codice consiste di classi rappresentanti 1) una famiglia di prodotti (sedia, divano, tavolo) 2) varianti di questa famiglia. Serve un modo di creare oggetti in modo da matcharli con altri arredamenti della stessa famiglia. Non vogliamo inoltre modificare il codice esistente quando aggiungiamo nuovi prodotti o famiglie.
Soluzione: Dichiaro le interfacce di ogni prodotto, le varianti dei prodotti sono sottoclassi.
Poi dichiaro l'abstract factory(interaccia con lista di metodi) dove ogni metodo ritorna una lista di prodotti astratti.
Poi estendo l'abstract ai prodotti concreti.

16.1.3 Builder

Scopo: permette di costruire oggetti un pezzo alla volta. **Problema**: Se un oggetto ha bisogno di tanti paramentri di inizializzazione avrà un costruttore giga grosso. **Soluzione**: si esegue prima un builder "generico" per poi dare gli altri valori all'oggetto con passaggi successivi.

16.1.4 Prototype

Scopo: Copia gli oggetti esistenti in un'oggetto definito. **Problema**: Immaginiamo di avere un oggetto, e volerne creare una copia esatta. Come fare? Farlo dall'esterno non è ideale: ci perderemmo tutti i dati privati! Inoltre, per fare il duplicato dobbiamo conoscere la classe concreta dell'oggetto.
Soluzione: Si delega il processo id clonazione all'oggetto effettivo così da non perdere i dati privati. Bisogna però implementare il metodo *Clone()*.

16.1.5 Singleton

Scopo: permette di assicurarsi che una classe abbia una sola istanza. **Problema**:

- si assicura che una classe abbia una sola istanza
- fornisce un punto di accesso globale alla risorsa

Soluzione:

- rendere il costruttore di default private
- creare un metodo getInstance che fa da builder

16.2 Structural patterns

16.2.1 Adapter

Scopo: permette ad oggetti con interaccia incompatibili di comunicare.
Problema: Immaginiamo di star creando un'app di monitoraggio del mercato in borsa. L'app scarica i dati in XML da un'API. Ad un certo punto, vogliamo integrare un'altra API, che però restituisce dati in JSON. Potremmo cambiare questa libreria, ma sarebbe un casino.
Soluzione: Creo un'adattatore che converte le interfacce.

16.2.2 Bridge

Scopo: permette di dividere un grande numero di classi in due gerarchie separate.
Problema: Ipotizziamo di avere una classe Shape con due subclasses: Circle e Square. Vogliamo estendere questa gerarchia per incorporare i colori, quindi inizialmente proponiamo di creare sottoclassi RedCircle, BlueCircle, RedSquare, BlueSquare. È palese che questo approccio non sia sostenibile per il futuro.
Soluzione: Semplicemente, facciamo diventare il colore una proprietà dell'oggetto Shape. In questo modo, switchamo da ereditarietà a composizione.

16.2.3 Adapter vs Bridge

Entrambi sono utilizzati per nascondere dettagli implementativi. L'adapter fa lavorare insieme componenti incompatibili, mentre il bridge permette ad astrazioni ed implementazioni di variare indipendentemente.

16.2.4 Composite

Scopo: permette di comporre oggetti in strutture ad albero e lavorare su queste strutture come oggetti individuali.
Problema: computazionalmente una merda a meno che il software non sia a sua volta strutturato ad albero.
Soluzione:Il composite pattern suggerisce di lavorare su prodotti e scatole attraverso un'unica interfaccia, che dichiara un metodo per calcolare il costo totale. In un prodotto, restituirebbe il costo del prodotto. In una scatola, restituisce la chiamata ricorsiva a tutti i contenuti. *Avete già capito dove voglio arrivare.* In questo modo, basta chiamare il metodo di calcolo sulla scatola grande e verrà chiamato ricorsivamente su tutti i nodi. *Spaventosamente elegante.*

16.2.5 Decorator

Scopo: Il decorator permette di agganciare nuovi comportamenti ad oggetti, inserendoli in oggetti wrapper che contengono questi nuovi behaviors.
Problema: non si possono coprire tutti casi con l'ereditarietà(per lo meno se sei umano).
Soluzione: Diventa una task di aggregazione più che di eretidarietà.
Si crea un oggetto wrapper che contiene i metodi del taghet e li chiama, così l'oggetto di base si interfaccia al wrapper per tutte le casistiche(tipo notifiche di facebook, whatsapp, sms, telegram, . . .).

16.2.6 Facede

Scopo: Fornisce un'interfaccia semplice per una libreria, framework o altre cose.
Problema: Immaginiamo di dover far funzionare il nostro codice con un set molto grande di oggetti che appartengono ad un sofisticato framework. Dovremmo, prima di tutto, inizializzare tutti gli oggetti, tenere traccia delle dipendenze, eseguire metodi nell'ordine corretto, e via dicendo. Il funzionamento del software diventerebbe quindi molto dipendente da questo framework.
Soluzione: la facede fornisce una semplice interfaccia al sottositema complesso.

16.2.7 Flyweight

Scopo: permette di ridurre lo spazio di memoria necessario al salvataggio di oggetti.
Problema: Immaginiamo di dover creare un videogioco sparatutto. Decidiamo di implementare un sistema di particelle complesso, in modo che proiettili, missili, detriti siano gestiti da oggetti. Avremmo quindi una struttura in cui ogni particella occupa molto spazio in memoria:

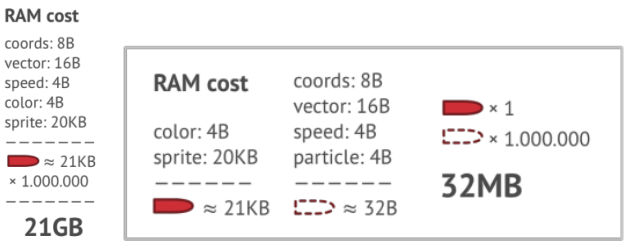

Figura 9: flyweight

Soluzione: Salviamo tutte le informazioni comuni in una classe a parte.

16.2.8 Proxy

Scopo: Permette di avere un sostituto ad un oggeto, potendo eseguire operazioni prima e dopo.
Problema: Perché dovremmo controllare l'accesso? Per esempio, se avessimo un oggetto che usa molte risorse di sistema ma che non utilizziamo sempre, sarebbe utile avere un lazy loading, ossia un'istanziazione solo quando necessario.
Soluzione: Creiamo quindi una classe proxy che abbia la stessa interfaccia dell'oggetto desiderato. In questo modo, agli oggetti che lo richiedono, lo passiamo al posto dell'oggetto vero e proprio. Il proxy crea quindi l'oggetto vero solo dopo la prima richiesta.

16.3 Behavioral patterns

A volte è necessario eseguire richieste agli oggetti senza conoscenze sull'operazione richiesta o il destinatario della stessa. Un esempio possono essere le librerie grafiche per bottoni e menù, che non implementano l'azione ma passano la richiesta. I behavioral patterns si occupano di questo: gestiscono l'assegnamento di

16.3.1 Chain of responsabilities

Scopo: permette di passare richieste attraverso una catena di handlers.
Problema: Immaginiamo di lavorare a un sistema di ordini online. Vogliamo restringere l'accesso al sistema, in modo che solo gli utenti autenticati possano ordinare. Vogliamo inoltre degli amministratori, che possano vedere tutti gli ordini. Decidiamo che questi controlli devono essere fatti in sequenza: se l'autenticazione fallisce, non è necessario fare altre verifiche. Col tempo, si aggiungono altre verifiche: un sanitizing dei dati, una verifica degli IP, un motore di caching. Ad ogni verifica aggiunta, il codice diventa un porcile.
Soluzione: Trasformiamo quindi queste verifiche in handlers: ognuna avrà un metodo check, ed un campo per il prossimo handler della catena.

16.3.2 Command

Scopo: trasforma una richiesta in un oggetto a se stante.
Problema: Immaginiamo di lavorare ad un software di text editing. Vogliamo creare una toolbar che contenga tasti per le operazioni più importanti, con una classe Button. Cominciamo a creare subclasses per ogni bottone necessario: OKButton, SaveButton, ApplyButton, ed ogni tasto che si aggiunge, la nostra voglia di vivere cala. Dev'esserci un modo più furbo.
Soluzione: Estraiamo i dati della richiesta tramite un oggetto dall'interfaccia Command, che avrà come implementazioni i vari tipi di comando possibili, aventi un solo metodo execute. In questo modo, ogni bottone chiamerà l'execute del suo Command, ma non solo: i comandi potrebbero, ad esempio, anche essere chiamati da scorciatoie da tastiera!

16.3.3 Interpreter

Scopo: Mappa un dominio ad un linguaggio, il linguaggio ad una grammatica, e la grammatica ad un design object-oriented gerarchico.
Problema: Se il dominio è caratterizzato da un linguaggio, il problema può essere risolto tramite un interprete.
Soluzione: Modelliamo il dominio con una grammatica ricorsiva ad albero, in cui ogni regola è un nodo o una foglia.

16.3.4 Iterator

Scopo: passa dentro ad una collection senza leekare la logica.
Problema: Creiamo un nostro oggetto di collection per salvare dati, con una struttura cazzutissima ed ultra segreta, che ci permetterà di diventare milionari grazie all'intelligentissimo metodo di esplorazione dei dati che ho. Abbiamo ora un problema: non vogliamo esporre le logiche con cui esploriamo la collection.
Soluzione: usiamo un iterator con i metodi get next e get prev per prendere gli oggetti.

16.3.5 Mediator

Scopo: riduce la coesione tra oggetti. Riduce la comunicazione fra gli oggetti perchè parlano attraverso lui(mediatore).

16.3.6 memento

Scopo: Permette di salvare e ripristinare lo stato degli oggetti senzaz rivelarne i dettagli implementativi.

16.3.7 Observer

Scopo: È un pattern che permette di definire un meccanismo di iscrizione per notificare

16.3.8 State

Scopo: permette ad un oggetto di alterare il suo comportamento in base allo stato.

16.3.9 Strategy

Scopo: pattern che definisce una famiglia di algoritmi, inseriti in classi separate e renderne gli oggetti intercambiabili.

16.3.10 Template

Scopo: permette di definire lo scheletro di un algoritmo nella sua superclasse, poi nelle sotto classi lo si agghinda.

16.3.11 Visitor

Scopo: Separa gli algortimi dagli oggetti su cui opera.

17 T17 - Architectural design

Permette di comprendere l'organizzazione ed il design della struttura generale di un sistema di software.
È il collegamento tra design e requirements engineering.
Identifica i componenti strutturali e le loro relazioni.
L'architettura è un modello di alto livello che descrive gli aspetti critici del sistema ed è comprensibile alla maggior parte degli stakeholders, permettendo la valutazione delle proprietà di sistema prima della sua esistenza.
Fornisce strumenti e tecniche per costruire il sistema.

17.1 Software architecture

Software architecture sono gli aspetti rilevanti dell'architettura.
Essa è una traccia che identifica componenti, interazioni, interconnessioni ed è definita in modo formale o informale. È fondamentale avere un vocabolario condiviso e ricco.

17.2 Elementi architetturali

Un subsystem è un sistema unico, le quali operazioni sono indipendenti dai servizi di altri subsystems. Un modulo è un componente di sistema che fornisce servizi ad altri componenti, ma non può essere considerato un subsystem.

17.2.1 Componenti

Essi sono unità di computazione o data stores: clients, servers, databases, filtri, layers. Possono essere semplici o composti(subsystem). Un architettura consistente in più componenti compositi è un sistema di sistemi.

17.2.2 Connettori

Sono elementi architetturali che modellano le interazioni tra componenti e le regole che le governano. Le interazioni si dividono in semplici (chiamate procedurali, variabili condivise) e comples- se/semantically rich (client-server, database access, async, piped data streams)

17.2.3 Configurazioni

Sono grafi connessi di componenti e connettori che descrivono strutture architetturali identificanti connettività, proprietà concorrenti/distribuite, aderenza ad euristiche/stili. I componenti compositi sono configurazioni.

17.3 Architectural design process

Tre fasi:

- System structuring(decompongo in sottosistemi e identifico la comunicazione fra sotto sistemi)
- Control modelling(scegliere un modello di controllo tra le parti)
- Modular decomposition(decompongo i subsystem in moduli)

17.3.1 Box and line diagrams

Sono diagrammi semplici ed informali che mostrano i subsystem e le loro relazioni. Mancano di semantica, non mostrando i tipi di relazioni o le proprietà. I requisiti per la semantica dei modelli dipendono da come il modello è usato.

17.3.2 Software architecture views

- logical view(astrazioni chiave del sistema)
- process view(mostra interazioni dei processi)
- development view(decomposizione del software)
- physical view(hardware)

Gli step per questo design sono capire i requirements, definire l'architettura, rappresentare e comunicare l'architettura, valutare l'architettura. Si utilizzano implementazione, miglioramento e manutenzione.

17.3.3 Non functional requirements

- performance
- security(architettura a strati)
- safety(trovare features con possibili falle di sicurezza)
- availability(includere componenti rindondanti)
- maintenability(usare componenti sostituibili e di cui si possa fare il fine-grain)

17.3.4 Euristica dei subsystem

Bisogna assegnare oggetti identificati nello stesso use case, allo stesso subsystem. Creare un subsystem dedicato per oggetti atti al movimento di dati tra subsystems. Minimizzare il numero di associazioni tra subsystem boundaries. Tutti gli oggetti nello stesso subsystem devono essere funzionalmente collegati.

17.3.5 layering e partitioning

Sono tecniche utili ad ottenere un basso coupling, che permettono di suddividere un sistema in subsystems. Il layering divide il sistema in strati che forniscono servizi allo strato più alto. Così, uno strato dipende solo da livelli più bassi e non conosce quelli più alti. Il partitioning divide un sistema in subsystems indipendenti.

17.3.6 Architecture reuse

I sistemi nello stesso dominio spesso hanno architetture simili: nascono quindi dei pattern/stili che catturano l'essenza di un'architettura.

17.4 Architectural patterns

Sono descrizioni di una buona pratica di design, testata in diversi ambienti, ed includono informazioni sul loro utilizzo. Sono rappresentati da tabelle e grafici.

17.4.1 Model-View-Control

Utilizziamo l'MVC quando abbiamo più modi di vedere ed interagire coi dati, e non abbiamo conoscenza dei futuri requirements dell'interazione coi dati. I dati e le rappresentazioni possono così cambiare indipendentemente, e abbiamo più views degli stessi dati. Aumenta però la complessità del codice. Abbiamo tre componenti:

- Model(tiene e gestisce i dati)
- View(Presenta i dati all'utente)
- Control(risponde agli input e interagisce con i modelli)

17.4.2 Layered Architecture

Qui, ogni layer fornisce dei servizi a quello sopra. Il più basso contiene i core services. Si utilizza quando dobbiamo creare nuove parti sopra sistemi esistenti, e lo sviluppo è basato su vari team che lavorano a funzionalità diverse. Risolve anche i requirements di sicurezza multi-level. È un ottimo sostituto dei layer di implementazione, supporta la ridondanza, ma ha problemi di performance e rende difficile separare nettamente i layers.

17.4.3 Repository

La repository è accessibile a tutti i componenti, che non interagiscono tra loro ma solo con la repository, che gestisce tutti i dati. Viene utilizzato quando abbiamo grandi volumi di informazioni e l'inclusione di dati nella repo triggera un'azione. Ha componenti indipendenti, propagazione dei cambiamenti, data management consistente, ma rischio di failure, comunicazione inefficiente e difficoltà di distribuzione della repo.

17.4.4 Client-Server

Utilizziamo client-server quando i dati devono essere accessibili da più posizioni, ed il carico è variabile (distribuzione dei server). Facilita la distribuzione di server e funzionalità, ma ogni server è un punto di failure, abbiamo performance imprevedibili, e problemi di gestione quando i server hanno diversi proprietari.
Thin e Fat client Un thin client implementa l'interfaccia grafica, delegando al server ed alla rete le operazioni più pesanti. Un fat client, invece, esegue l'applicazione localmente, ed è quindi più complesso e potente. Gli aggiornamenti vanno installati su tutti i clients.

17.4.5 Pipe and filter

Il processo dei dati in un sistema è organizzato tramite una sequenza di componenti di processing, ognuno dei quali esegue una tipologia di trasformazione dei dati, i quali scorrono da un componente all'altro. Si utilizza in applicazioni di data processing nelle quali gli input sono processati in stadi separati per generare output. Un esempio sono i compilatori. Si prestano bene al riutilizzo ed ai business processes. Hanno evoluzione semplice e implementazione sequenziale/parallela. Devono però avere un formato di data transfer condiviso, e codificare/decodificare i dati.

17.5 Application architectures

I sistemi di applicazioni possono essere raggruppati per il tipo di business. Siccome i business hanno molto in comune, i loro sistemi applicativi tendono ad avere un'architettura comune che riflette i requirements. Un'architettura generica può essere configurata e adattata per creare un sistema che ha specifici require- ments. Le application architectures sono un punto di partenza per il design, utilizzabili come checklist, come modalità organizzativa, come mezzo di riutilizzo, come vocabolario.

17.5.1 Centralied vs Decentralized

Il design centralizzato rende facili le modifiche nella control structure, penalizzando però le performance del singolo control object. Quello decentralizzato divide le responsabilità, funziona bene con l'Object- Oriented e permette di dividere il carico. Per scegliere, osserviamo i sequence diagrams ed i control objects, verificandone la partecipazione. Se il sequence diagram assomiglia a una forchetta, centralizzato, se assomiglia a una sedia, decentralizzato. C'è scritto veramente? I sequence diagrams sono derivati dagli use case.

17.5.2 Procedure vs event driven

Nel procedure driven, il controllo è nel codice del programma e l'utente ha poco controllo. Nell'event driven, il controllo risiede in un dispatcher che chiama funzioni via callbacks; l'utente ha qui molto controllo.

17.5.3 Vantaggi di una system architecture esplicita

- Comunicazione tra stakeholder(discutere del sistema)
- analisi del sistema
- riutilizzo in larga scala
- definizione di strutture di divisione del lavoro

Il design dell'architettura è uno dei primi passi della metodologia agile, perché il refactor futuro è molto costoso.

18 T18 - User Interface Design

Gli utenti di un sistema spesso giudicano più l'interfaccia delle funzionalità: un'interfaccia realizzata male può causare errori o far disinstallare il software. Nel design, bisogna considerare i fattori umani: memoria a breve termine limitata, errori, capacità differenti, preferenze di interazione diverse. Dobbiamo quindi matchare le skills, esperienze ed aspettative degli utenti, considerandone però anche i limiti e le possibilità di errore. È utile basarsi su un set di design principles:

- User familiarity(interfaccia user oriented)
- consistenza(consistenza dell'insieme)
- minimal surprise(un determinato comano funziona in maniera conosciuta dall'utente)
- recoverability(resiliente agli errori dell'utente)
- user guidance(sistemi di aiuto)
- real word mapping(layout famiglaire per le informazioni)
- consistency(futures simili nello stesso posto)
- less is more(feature meno importanti non devono essere in mezzo ai coglioni)
- anticipation(nascondere features inaccessibili)
- customization(per utenti avanzati features fighe)
- transparency(l'interfaccia non deve coprire altri contenuti)
- contiguity(testi esplicitanti vicino a icone grafiche)
- memory load(ricordare all'utente i dettagli)
- user control(identificare il responsabile delle azioni)
- speak user's lenguage(istruzioni comprensibili)

Alcuni tipi di componenti:

- windows
- icons
- menus e pulsanti
- pointing
- graphic element

Direct manipulation L'utente si sente in controllo del computer, ci mette poco ad imparare, ottiene feedback immediati. Però, la derivazione di un information space model è difficile, la navigazione può essere complessa e quindi richiedere molte risorse.
Menu I comandi sono presentati in una lista, lo sforzo di scrittura è minimale, gli errori improbabili. È possibile fornire aiuto contestuale. Però, le azioni complesse (and/or) sono difficili da rappresentare. I menù sono adatti a poche opzioni ed utenti non esperti, perché gli esperti preferiscono i comandi testuali.
Form Il form permette semplice data-entry, facilità di apprendimento, possibilità di verifica. Però richiede molto spazio a schermo e causa problemi quando l'utente non richiede esattamente ciò che è presente a schermo.

Command language I comandi permettono uno sviluppo rapido, di complessità arbitraria ed interfacce minimali. Però, gli utenti dovranno ricordare il linguaggio: non è adatto ad utenti saltuari. Faranno inoltre errori, e sarà richiesta la possibilità di scrittura.
Linguaggio naturale Esso è accessibile ad utenti inesperti ed estensibile facilmente. Però, il vocabolario è limitato e confinato a domini specifici. La tecnologia non è del tutto adatta a rendere queste interfacce accessibili ad utenti principianti (questa slide probabilmente è stata scritta nel 2004, quando Siri e Google Assistant non esistevano. Un aggiornamento non sarebbe male eh), ma gli utenti esperti odiano dover scrivere molto. È necessario poter scrivere.

18.1 Design process

Per il design è utile sviluppare un prototipo low-fidelity, che sia semplice ed economico anche nella modifica, in modo da far capire le caratteristiche principali senza concentrarsi sulle piccolezze estetiche. Bisogna spiegare le convenzioni agli utenti ed è difficile mostrare i comportamenti. Il processo di design consiste in:

1. analisi e comprensione delle attività dell'utente
2. produzione di prototipi su carta e confornto con gli utenti
3. design del prototipo dinamico e valutazione
4. design prototipo eseguibile, valutazione e implementazione

18.1.1 Information presentation

L'information presentation è costituita dallo studio di come presentare le informazioni all'utente, direttamente o trasformandole. L'MVC è una modalità che supporta più presentazioni di dati. Dobbiamo così porci più domande:

- l'utente è interessato alle info e alle loro relazioni?
- quanto rapidamente variano?
- l'utente deve poter rispondere ai cambiamenti?
- manipolazione diretta?
- testo o numeri?

Sfruttiamo i colori per supportare le attività dell'utente in maniera intelligente e consistente, stando attenti anche agli accostamenti.
Messaggi di errore Una corretta rappresentazione degli errori è fondamentale. I messaggi devono essere educati, concisi, consistenti e costruttivi.

18.2 Valutazione della user interface

Dopo il lavoro di design, bisogna valutarlo vs. le specifiche di usability, seguendo parametri come learna- bility, speed of operation, robustness, recoverability, adaptability. Utilizziamo più tecniche di valutazione:

- expert review
- questioanri
- registrazioni video dell'utilizzo
- strumenti di raccolta delle info
- feedback degli utenti
- competitive usability testing

19 T19 - UML Diagrams for system design

19.1 Component diagram

Il component diagram mostra gli elementi concettuali del business process (in UML1 erano componenti fisici), che forniscono o utilizzano interfacce per l'interazione con altri costrutti del sistema.
Component Unità logica del sistema, ha interfacce definite ed è rappresentato da un rettangolo.
Interface Descrive un gruppo di operazioni usate o create dai componenti. Un cerchio intero rappresenta un'interfaccia creata, mezzo ne rappresenta una required.
Dependencies Le dipendenze sono indicate da frecce tratteggiate.
Ports Le porte sono rappresentate da quadrati sul bordo. Sono usate per supportare l'esposizione di un'interfaccia required/created.

19.2 Package diagram

Il package diagram serve per rappresentare la struttura dei package, dove i package vengon usati in progetti di grandi dimensioni per aumentare la modularità.
Qui vengono aggiunte anche le dipendenze nel disegno delle dipendenze.

19.3 Deployment diagram

Il deployment diagram mostra l'architettura di esecuzione del software, quindi processori, nodi, devices, le loro connessioni e la distribuzione dei file. Utilizziamo questi diagrammi per mostrare l'hardware e software di esecuzione.

20 T20 - Object Oriented Desing

L'Object Oriented Design ha lo scopo di aggiungere dettagli alla requirements analysis e all'architecture model, prendendo decisioni riguardanti l'implementazione. Gli obbietivi sono:

- riutilizzare le conoscenze del passato
- riutilizzare funzionalità disponibili
- svuluppare la definizione di robustezza e adattabilità
- nuove funzioni
- adattare ssitema esistente a nuovo ambiente

4 step fondamentali:

1. identificazione di soluzioni esistenti(con l'ereditarietà sfruttiamo componenti, soluzioi e design patterns)
2. interface specification(descriviamo bene ogni class interface)
3. object model restructuring(miglioriamo la comprensibilità e l'estensibilità)
4. object model optimization(miglioriamo le Performance)

20.1 Refactoring

Cambio della struttura interna senza cambiare il comportamento "visto" dall'esterno.
Migliora il design del software, velocizzandolo, rende il design più comprensibile, rende il debugging più facile, permette di programmare più velocemente. Teniamo in conto tre principi:

- non si aggiungo funzioni nel refactoring
- verificare l'esistenza di test ottimali prima del refactoring
- piccoli passi localizzati

20.2 Tecniche di riutilizzo

20.2.1 Ereditarietà

Due utilizzi:

- Descrizione delle tassonomie: utilizzata durante la requirements analysis, identifica oggetti del domi- nio che hanno una relazione, allo scopo di rendere l'analysis model più comprensibile.
- itemizeSpecifica delle interfacce: usata durante l'object design, identifica le signatures degli oggetti identificati, allo scopo di aumentare il riutilizzo, la modificabilità, l'estensione.

Distinguiamo, inoltre, tra inheritance (white-box reuse), e composition (black-box reuse).
Abbiamo più ragioni per le quali può essere necessaria la creazione di oggetti:

- nuovi compenenti della GUI
- implementazione di algoritmi
- classi controller e cooridnator

20.2.2 Delegation

Nella delegation, due oggetti sono coinvolti nella gestione di una richiesta del client: il receiver delega le operazioni al delegate, assicurandosi che il client non utilizzi il delegate in modo errato.

20.2.3 Contraction

Il goal della contraction è rendere le operazioni della superclass invisibili, implementando metodi nella superclasse ed overridandoli con metodi vuoti nella subclass.

20.3 package design

Il packaging design impacchetta il design in unità discrete che possono essere modificate, compilate, collegate, riutilizzate.
Due principi base: minimizzare il coupling, massimizzare la cohesion. Come fare?

1. partire con un'interfaccia per ogni subsystem
2. limitare il numero di operazioni dell'interfaccia
3. se l'interfaccia ha troppe operazioni, riconsiderare il umero di interfacce
4. se abbiamo poche interfacce, riconsideriamo il numero di subsystem

20.4 Information hiding

Per ottenere un buon information hiding, definiamo interfacce pubbliche, applicando il "need to know" principle: meno dettagli una classe deve sapere, più facile è cambiarla/non rovinarla con cambiamenti.

22 T22 - testing

22.1 Componenti del testing

22.1.1 Test plan

Il test plan è utilizzato per dimostrare che il software è privo di falle e si comporta come richiesto dai requirements.

22.1.2 Test specification

Documenta lo scopo di un test; in caso di test compositi, documenta la relazione tra le parti ed il whole test. Descrive le condizioni che indicano quando il test è completo. In generale, è un modo per valutare i risultati.

22.1.3 Test oracle

È un set di risultati predetti per un set di test, e si usa per determinare il successo del testing. MOlto complicato da creare.

22.1.4 Test cases

È un set di input al sistema. Un testing di successo è basato sulla scelta dei giusti test cases.

22.1.5 Test suite

Ovviamente bisogna testare il codice per farlo funzionare la prima volta, facendo ad hoc testing o costruendo una test suite.
Distinguiamo tra due tecniche di testing: glass box testing, in cui esaminiamo il codice, e black box testing, basato sulla conoscenza del risultato atteso.

22.2 Tipologia di testing

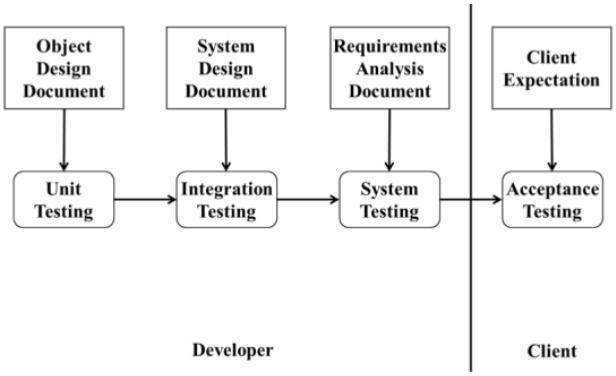

Figura 10: Tipologie di testing

22.2.1 Unit testign

Lo unit testing testa le singole unità che compongono il sistema, allo scopo di trovare falle in algoritmi, dati, sintassi.
Distinguiamo, anzitutto, tra static testing(analisi, code review) e dynamic testing (black-box, white-box).

22.2.2 Integration testing

L'integration testing testa un gruppo di subsystems, ed eventualmente l'intero sistema. È fatto dagli sviluppatori per testare le interfacce tra subsystems. L'intero sistema è visto come una collezione di subsystems, determinati durante il system e object design.
Driver È un componente che chiama l'unità testata e controlla i test cases **Stub/Double** è un componente che simula la presenza di un altro, rispondendo alle chiamate con dati falsi. Non deve comportarsi esattamente come chi sostituisce, ma deve fornire circa la stessa API. Ne distinguiamo 4 tipi:

- dummy
- stub(forza il sistema verso il path che vogliamo provare)
- mock(valori hardcoded)
- fake(implemetazioni alternativa)

22.2.3 System testing

Il system testing testa l'intero sistema, ed è fatto dagli sviluppatori allo scopo di determinare se il sistema rispetta i requirements funzionali e di performance.
I test cases sono centrati attorno ai requirements e alle funzioni chiave, ed il sistema è trattato come una black box. Gli unit test cases sono riutilizzati, ed i nuovi test cases vanno sviluppati.
Performance testing Il goal è tentare di violare i non-functional requirements, testando come il sistema si comporta quando è sovraccaricato.
Acceptance Testing Il goal è dimostrare che il sistema è pronto per l'utilizzo, con test e svolti scelti dal cliente. Distinguiamo tra alpha test, in cui si testa nell'ambiente di sviluppo, e beta test, in cui si testa nell'ambiente del cliente.

22.3 Object oriented tesiting

Object Class Testing Un test completo di una classe testa tutte le operazioni associate, gli attributi, tutti gli stati. L'ereditarietà rende questo test più complesso.
Ereditarietà, polimorfismo, dynamic binding

- Ereditarietà: i metodi ereditati devono essere ritestati nelle subclasses: il contesto delle superclasses potrebbe essere incompleto
- Polimorfismo: i parametri devono avere più di un set di valori e un'operazione deve essere implemen- tata da più di un metodo
- Dynamic binding: i metodi che implementano un'operazione sono sconosciuti fino al runtime

Continuous testing Il continuous testing consiste in build e relativo testing ogni giorno, in modo che il sistema sia sempre eseguibile.