Ingegneria del Software

Cosimo Botticelli, Francesco Galasso

Chapter 1

Introduzione

Se questi appunti ti sono di aiuto considera una donazione. ;)

1.1 Concetti fondamentali di Ingegneria del Software

In questo corso ci occuperemo di sviluppare skill che vanno oltre la programmazione e che servono a sviluppare e manutenere un sistema software. Questo processo, e di conseguenza il corso, si dividerà nelle seguenti sezioni:

- M1: Concetti generali dell'ingegneria del software
- M2: Analisi e specifica dei requisiti
- M3: Progettazione architetturale e ad alto livello: System Design
- M4: Progettazione a basso livello: Object Design
- M5: Software Testing

Definizione di Software Secondo lo standard IEEE, il software è l'insieme di programmi, procedure e documentazione compresi in un sistema computerizzato.

Storia del Software Nel corso dei decenni il Software ha subito una evoluzione significativa. Dapprima esso era **arte**, ossia prodotto da una singola persona o piccolo gruppo di persone per utilizzo privato.

Poi è diventato **artigianato**, ossia applicazioni sviluppate da produttori specializzati su richiesta specifica di un cliente.

Infine, esso è divenuto **industriale**, ossia prodotto per un mercato più o meno grande che sia. Ciò ha causato un grande aumento di complessità, dimensioni e richiesta.

1.1.1 Programma vs Prodotto

Non sono da confondere **programma** e **prodotto software**.

In un programma, l'autore è anche il fruitore, mentre in un prodotto software no. Le conseguenze di ciò sono una necessità di documentazione e un approccio più formale al prodotto, in quanto questo avrà dimensioni maggiori e sarà destinato a un pubblico più generale.

Inoltre, la manutenzione di un prodotto software prende il significato di espansione, cambiamento, evoluzione, mentre in un prodotto hardware ciò non è possibile, in quanto la fase successiva al rilascio consiste di poche modifiche e perlopiù riparazioni.

Il prodotto software si suddivide a sua volta in:

- Prodotti generici: Sistemi stand alone da distribuire su un mercato più o meno grande, ma comunque generale. Ciò che l'utente paga è una licenza d'uso, e non il prodotto stesso.

- Prodotti specifici: Sistemi commissionati da un fruitore o macrosistema specifico. Il fruitore compra il software nel suo insieme, e non solo una licenza o permesso d'uso.

1.2 Principi dell'Ingegneria del Software

1.2.1 I principi

L'Ingegneria del Software si basa su alcuni importanti principi, quali:

- **Rigore:** Precisione di progettazione ed esecuzione

- **Formalità:** Più specifica del rigore, è fondamento matematico

- **Separazione di aspetti diversi:** Affrontare in fasi e modi diversi aspetti diversi di un problema più complesso

- **Modularità:** Suddivisione di un sistema complesso in parti più semplici. È importante mantenere un certo livello di indipendenza fra i moduli. Si distingue dalla separazione di cui sopra per l'accezione più pratica

- **Astrazione:** Si identificano gli aspetti cruciali di un determinato problema in un determinato momento, ignorando quelli superflui. Questo garantisce un certo grado di generalizzazione che idealmente dovrebbe essere massimizzato pur mantenendo una descrizione corretta del problema

- **Anticipazione del cambiamento:** Previsione dei cambiamenti a cui il sistema sarà soggetto. Se eseguito bene, il sistema potrà cambiare, crescere ed evolversi senza modificare la sua struttura fondamentale e senza generare ulteriori difetti
- **Generalità:** Tentare di risolvere il problema nella sua accezione più generale. Se eseguito correttamente ha come effetto collaterale l'anticipazione del cambiamento
- **Incrementalità:** Lavorare a un sistema complesso per passi successivi, dando priorità alle core funcionalities e sviluppando solo dopo i dettagli o funzionalità secondarie
- **Metodo (o Tecnica):** Strumento generale per risolvere determinate classi di problemi
- **Metodologia:** Insieme di principi, metodi ed elementi disciplinari che garantiscono l'efficacia del proprio procedere. È più generico del metodo e difatti lo contiene
- **Strumento (Tool):** Strumento per fare qualcosa in modo più preciso, corretto o immediato. È strettamente legato a skill tecniche e le complementa. Alcuni esempi di strumenti sono una IDE, uno strumento di creazione UML, LaTeX, etc
- **Procedura:** Una combinazione di *strumenti* e *metodi*, più dettagliato di un metodo, descrive precisamente la risoluzione di un problema
- **Paradigma:** Ancora più generale di una *metodologia*, è una filosofia o approccio a un qualcosa

1.2.2 Processo Software

Il processo software può essere formale, semi-formale o informale, ed è definito come l'insieme di organizzato di attività che sovrintendono alla costruzione del prodotto, e comprende *metodi*, *tecniche*, *metodologie* e *strumenti*.

Inoltre è suddiviso in fasi secondo uno schema di riferimento che chiamiamo **ciclo di vita del software**.

Definizione Il ciclo di vita del software (CVS) è il periodo che va dal concepimento del prodotto software al momento in cui lo stesso non è più mantenuto. Il modello del CVS è una caratterizzazione descrittiva o prescrittiva (in base a chi lo legge, se persona familiare o nuovo contributore) di come deve essere sviluppato.

1.2.3 Fasi del CVS

Le fasi generali di molti CVS sono:

- **Studio di fattibilità:** Serve a valutare se il progetto è fattibile, realistico e quali sono i suoi benefici

- **Analisi dei requisiti:** Analisi più specifica dei bisogni dell'utente, del da farsi e degli standard qualitativi. Il linguaggio usato è comprensibile dall'utente e comunque ad alto livello

- **Progettazione:** Scomposizione del sistema in moduli (architectural design) e specifica dei dettagli interni di ogni componente (detailed design). È in questa fase che si smette di chiedersi il **cosa** e si inizia a chiedersi il **come**

- **Programmazione e test di unità:** Codifica di ogni modulo e test dello stesso in isolamento

- **Integrazione e test di sistema:** Integrazione dei moduli e test del sistema nel suo insieme

- **Deployment:** Fase di distribuzione del Software all'utenza

- **Manutenzione:** Fase successiva al rilascio, di evoluzione e cambiamento del Software

1.2.4 Modelli di Ciclo di Vita del Software (CVS)

Modello a cascata (Waterfall)

Era un modello popolare negli anni '70, avente un approccio ingegneristico derivante dall'industria manifatturiera. Prevedeva la progettazione, la produzione di un prototipo e la messa in produzione.

I suoi **pro** sono stati la definizione di concetti utili successivamente utilizzati ed espansi in altri modelli, la semplicità di gestione del progetto e la facilità di comprensione del modello stesso.

I suoi **contro** invece sono stati la mancanza di interazione con il cliente, data dalla struttura modulare ma poco interconnessa delle fasi (una volta conclusa la progettazione difficilmente si modificava il progetto, e una volta mandato in produzione il prodotto difficilmente si tornava a rivedere il prototipo). Il cliente era al corrente di ciò che succedeva solo all'inizio e alla fine dell'intero progetto, la parte intermedia era determinata dalla correttezza di questa interazione iniziale e del progetto.

V&V e Retroazione (Feedback)

Ci sono stati molti modelli che si basavano sulla struttura di quello a cascata ma al contempo si proponevano di correggere alcuni suoi errori. Per esempio il modello **V&V e Retroazione (Feedback)** prevede un controllo alla fine di ogni fase. La natura di questo controllo può essere di diversi tipi (verification, validation o entrambi), e se uno o più di questi controlli falliscono, si torna al passo precedente per rivederlo.

La **verification (o verifica)** è la valutazione della verità della corrispondenza fra il prodotto e la sua specifica, mentre la **validation (o convalida)** consta dell'appropriatezza di un prodotto rispetto alla sua missione operativa.

In pratica la verifica testa se ciò che è stato fatto sia corretto, mentre la convalida testa se si è fatta la cosa giusta in primo luogo.

Il concetto di feedback può essere generalizzato, ed esso può far ritorno a una fase precedente arbitraria, e non necessariamente a quella immediatamente precedente.

Sviluppo evolutivo

Questo tipologia di sviluppo del software prevede la rivisitazione di fasi precedenti, anche molto precedenti, del progetto quando necessario. Un approccio del genere accomoda ottimamente il cambiamento dei requisiti e del progetto in corso d'opera, al contrario del modello a cascata, ma si rischia di perdere di vista il progetto nel suo insieme. Inoltre il prodotto potrebbe non avere una struttura solida, e dal punto di vista manageriale la gestione è complessa.

Trasformazioni formali

In questo modello di sviluppo, ogni passo è composto da descrizioni formali e può dare origine in modo quasi automatico al successivo. Un metodo del genere raramente è applicabile in contesti reali, ma può essere utile per lo sviluppo di sistemi critici come per esempio quelli orientati alla safety.

Sviluppo a componenti

Si basa sullo sviluppo e utilizzo di componenti già esistenti (off the shelf). Utile per sistemi sufficientemente generici, ma man mano che aumenta la specificità di un sistema, diminuisce la possibilità di utilizzare componenti già esistenti.

Iterazioni: Sviluppo incrementale

Questo è uno dei modelli che si basa sulle iterazioni del processo, e si pone l'obiettivo di rilasciare diverse e nuove funzionalità durante l'avanzamento dell'intero processo.

Uno dei vantaggi principali è che in caso di fallimento del progetto, il lavoro compiuto non va perduto, e si può dire la stessa cosa del compenso ricevuto.

Il processo è tale che quando si iniziano a sviluppare i sottosistemi (incrementi), le fasi alte e di definizione del progetto sono già ultimate. I sottosistemi, o incrementi, vengono sviluppati, messi in produzione, rilasciati e manutenuti secondo un piano di priorità, in modo tale che le prime funzionalità a essere rese disponibili sono quelle core, seguire da funzionalità via via sempre meno fondamentali. Questo ha due effetti:

- Il sistema è utilizzabile già da un periodo iniziale dell'intero processo di produzione, cosa che favorisce il feedback da parte dell'utenza man mano che vengono rilasciate nuove funzionalità
- Essendo le prime funzionalità a essere rilasciate quelle core, riceveranno molto feedback e nelle fasi avanzate del processo di produzione saranno molto stabili e testate, caratteristica importante delle funzionalità base di un sistema

Ne deriva che la fase di integrazione di nuovi sottosistemi è estremamente delicata ed importante.

"Release early, release often."

Modello a spirale

Un **meta-modello** che va a formalizzare il processo iterativo. Graficamente può essere rappresentato da una spirale divisa in sezioni, sulla quale ricadono le varie fasi dello sviluppo. In base alla sezione della spirale su cui la fase ricade, essa rientra in una diversa sottofase.

Strumenti: Prototipo

Mock-Up/Breadboards Uno degli strumenti a nostra disposizione è il prototipo. Questo è un'anteprima, e serve a dare all'utente un'idea del prodotto finale senza necessariamente spendere molte risorse sullo stesso. Il prototipo può essere di tipo **mock-up**, ossia una simulazione dell'interfaccia utente, oppure a **breadboards**, nel qual caso vengono testati i limiti e le capacità dei sottosistemi senza un'interfaccia utente.

Throw-Away/Esplorativo Un altro tipo di caratteristica della prototipizzazione che non va a escludere le precedenti (quindi ortogonale) è il suo essere di tipo **throw-away**, ossia un prototipo sul quale non si spendono molte risorse e che deve essere gettato una volta revisionato, o di tipo **esplorativo**, che è esattamente il contrario.

Il primo solitamente testa ed esplora caratteristiche del software non ancora ben comprese, mentre il secondo raffina ed esplora ulteriormente caratteristiche già abbastanza studiate. Così facendo si minimizza il rischio di spendere molte risorse su prototipi che si riveleranno totalmente da rivedere mentre si investe di più su quelli che potranno probabilmente essere riciclati e utilizzati nel sistema finale.

Gestione dei rischi: Compito del manager è anche minimizzare la possibilità che un determinato rischio si avveri, e nel caso, minimizzare il danno causato.

Ogni modello di CVS comporta dei rischi specifici, per esempio il modello a cascata comporta il rischio di perdere la direzione generale del progetto durante la fase intermedia dello sviluppo, ma il problema non sussiste se si sta sviluppando un sistema già ben conosciuto e con tecnologie note.

La decisione infine sta ai manager, che devono valutare quale tipo di CVS usare e quali tipi di rischi in generale valga la pena assumersi.

1.3 UML: Unified Modeling Language

UML è diventato lo standard de facto per la modellazione dei sistemi software.

1.3.1 Modelli

Modellare vuol dire costruire una astrazione della realtà, che nasconde i dettagli superflui senza trascurare quelli rilevanti, quindi avendone comunque una visione corretta.

In informatica, un modello è una astrazione, che ci permette di cogliere dettagli del codice e del sistema che non riusciremmo a comprendere guardando solo il codice (complice anche la crescita e la complessità dello stesso negli anni) o la realtà del problema.

Un modello è dunque una astrazione che descrive un sottosistema.

Una vista rappresenta invece alcuni specifici aspetti di un modello.

Una notazione è un set di regole testuali o grafiche per rappresentare una vista.

In un sistema, sia i modelli che le viste potrebbero sovrapporsi fra loro.

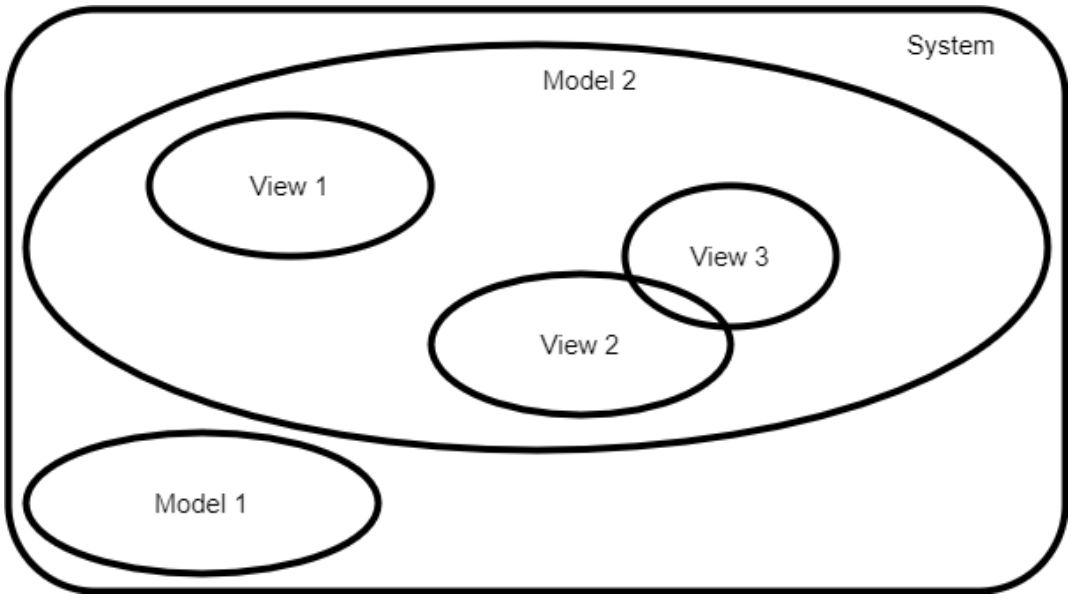

Figure 1.1: Rappresentazione grafica di modelli e viste nel contesto di un sistema.

1.3.2 Fenomeni e concetti

Un oggetto, per com'è percepito nel dominio del suo mondo, viene chiamato *fenomeno*, mentre il termine *concetto* può essere descritto come una tripla composta dal suo **nome**,

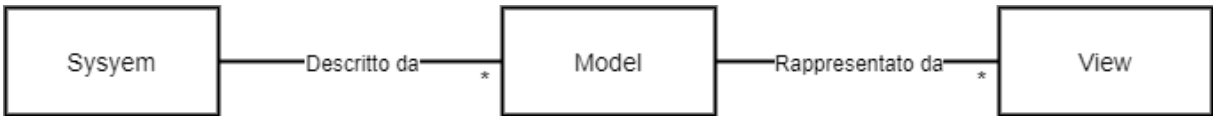

Figure 1.2: Un sistema può essere descritto da molti modelli, che a loro volta possono essere rappresentati da molte viste.

dallo **scopo**, ossia le proprietà che un fenomeno deve avere per appartenere a quel concetto, e i **membri**, ossia i fenomeni che appartengono a quel concetto.

L'**astrazione** è considerabile come l'atto di partire dai fenomeni e categorizzarli in concetti.

La **modellazione** è lo sviluppo di astrazioni che rispondono a certe caratteristiche di una classe di fenomeni, ignorando le caratteristiche invece superflue.

1.3.3 Componenti base di UML, come e quando utilizzarli

- **Use case diagrams:** Descrivono il funzionamento del sistema dal punto di vista del cliente.
- **Class diagrams:** Descrive la struttura statica del sistema, solitamente composta da oggetti, attributi e relazioni fra gli stessi.
- **Sequence diagrams:** Descrive il comportamento dinamico del sistema, che avviene tramite lo scambio di input e output fra il sistema stesso e attori od oggetti.
- **Statechart diagrams:** Una macchina a stati finiti che descrive il comportamento dinamico di un singolo oggetto, delineando gli stati che può assumere e come può assumerli.
- **Activity diagrams:** Modella il comportamento dinamico del sistema e in particolare il workflow.

Diversi tipi di domini In base al tipo di dominio in cui ci troviamo, dobbiamo assumere approcci diversi e anche il tipo di documento da produrre cambia.

Nel **demonio delle applicazioni**, ossia l'ambiente in cui il sistema opera, verrà sviluppata la parte di **Requirements Analysis**, mentre nel **dominio delle soluzioni**, che riguarda le tecnologie disponibili per sviluppare il sistema, andranno sviluppate le sezioni di **System Design** e **Object Design**.

Attori, classi, istanze Importante è la distinzione fra questi tre concetti.

Un **attore** interagisce con il sistema dall'esterno, e potrebbe essere o non essere rappresentato nello stesso.

Una **classe** è invece per definizione modellata all'interno del sistema. Essa è una astrazione che modella un'entità all'interno del dominio del sistema.

Infine, una **istanza** è appunto una istanziazione specifica di una classe.

Chapter 2

Requirements Analysis Document

Questo documento serve a dare allo sviluppatore e all'utente una visione univoca del sistema da sviluppare, astraendolo e prevedendone le funzionalità.

Si può cercare di diminuirne la complessità complessiva tramite **astrazione**, **decomposizione** (divide et impera) o **gerarchia** (layering).

2.1 Requirements Elicitation

Fare l'analisi dei requisiti significa identificare lo scopo del sistema, identificare cosa è nel sistema e cosa non lo è (cosa che si può vedere analoga all'identificare cosa è una classe e cosa è un attore).

Questo documento è inoltre poco formale, vicino al linguaggio naturale al fine di essere compreso dal cliente. Esso comprende scenari e casi d'uso. I primi sono veri e propri esempi concreti dell'utilizzo del sistema, mentre i casi d'uso utilizzano un linguaggio un po' più generico, seppur naturale. In generale, negli scenari vediamo delle istanze ("Piero esegue il login...", "la serie Lost viene aggiunta alla wishlist...", "si procede all'acquisto del gioco del Monopoly..."), mentre nei casi d'uso delle classi ("l'utente esegue il login...", "l'elemento viene aggiunto alla wishlist...", "si procede all'acquisto dell'articolo...").

Usiamo questi strumenti per descrivere in modo non ambiguo e chiaro al cliente la natura del sistema che andremo a realizzare. È importante che siano chiari e corretti perché verranno utilizzati come base per lo sviluppo di molte altre parti della documentazione.

I diversi tipi di Requirement Elicitation

- **Greenfield Engineering:** Non esiste un sistema attuale, si inizia la progettazione da zero.
- **Re-engineering:** Re-design o re-implementazione di un sistema esistente, per esempio causata dalla nascita di nuove tecnologie o nuove richieste di mercato.
- **Interface engineering:** Fornire i servizi di un ambiente esistente in un nuovo ambiente. Causato dall'evoluzione di tecnologie o richieste di mercato esistenti.

2.2 Problem Statement

Parte del documento solitamente prodotta dal cliente, contenente una descrizione del problema che il sistema deve affrontare e come affrontarlo. Un buon Problem Statemente dovrebbe contenere:

- **Situazione corrente:** Problema che deve essere risolto.
- **Scenari:** Descrizione di alcuni scenari in maniera discorsiva, preferibilmente scenari che rappresentano bene le funzionalità principali del sistema.
- **Requisiti:** Funzionali, non funzionali e pseudo-requisiti (vincoli).
- **Schedule:** Principali milestones e deadlines.
- **Ambiente target:** Descrizione dell'ambiente e dell'architettura in cui il sistema andrà ad operare.
- **Criteri di accettazione da parte del cliente.**

Dettagli situazione corrente. Una volta definito il problema, va definito se la soluzione andrà collocata nell'**application domain** (sviluppo di nuove tecnologie) o nel **solution domain** (cambiamento del sistema stesso). A volte una buona idea non può essere sviluppata per mancanza di tecnologie adatte, come per esempio internet che fu ipotizzato molto prima dello sviluppo della tecnologia necessaria per implementarlo. Altre richiedono lo sfruttamento più consapevole di tecnologie già esistenti.

Tipi di requisiti. I **requisiti funzionali** descrivono l'interazione fra l'attore e il sistema indipendentemente dall'implementazione, mentre quelli **non funzionali** descrivono aspetti percettibili dall'utente finale ma non direttamente legati alla parte funzionale, come per esempio il tempo di risposta.

I **vincoli**, o **pseudo-requisiti** invece sono imposti dal cliente o dall'ambiente in cui il sistema viene collocato. Quindi per esempio il linguaggio di implementazione.

2.3 Requirements Validation

In questa sezione del documento definiamo alcuni criteri di correttezza dei requisiti:

- **Correttezza:** Rappresenta la visione del cliente.
- **Completezza:** Descrive tutti i possibili scenari, compresi quelli eccezionali.
- **Consistenza:** Assenza di contraddizioni fra i vari requisiti.
- **Realismo:** I requisiti possono essere realizzati.
- **Tracciabilità:** Ogni funzione del sistema può essere ricondotta a un requisito.

2.4 Requirements Evolution

Una caratteristica dei requisiti è che essi cambiano molto e spesso. È utile dunque avere dei tools che aiutino a tenere traccia di suddetti cambiamenti.

2.5 Scenari

Abbiamo diversi tipi di scenari, fra cui:

- **As-is:** Usato per descrivere una situazione corrente, spesso utilizzato in progetti di re-engineering o in cui esiste un sistema funzionante su cui basarsi.
- **Visionary scenario:** Usati per descrivere un sistema futuro non attualmente esistente.
- **Evaluation scenario:** Test dell'utente secondo i quali il sistema dovrà essere valutato. Possono anche essere chiamati test scenarios.
- **Teaining scenarios:** Istruzioni step-by-step usati per allenare al sistema un utente novizio.

Gli scenari possono essere difficili da trovare. Talvolta l'utente stesso potrebbe non riuscire a verbalizzare i requisiti necessari che desidera vedere nel sistema ultimato. Inoltre scenari potrebbero essere aggiunti in futuro o essere soggetti a cambiamenti. In generale l'approccio all'acquisizione di scenari dal cliente deve essere di tipo esplorativo e dialettico.

2.6 Casi d'uso

Un caso d'uso è un flusso di eventi nel sistema che include l'interazione con un attore. Lo use case model descrive correttamente tutte le funzionalità del sistema.

I casi d'uso sono più generali degli scenari ma mantengono comunque un linguaggio discorsivo, almeno nel flusso di eventi.

2.6.1 Relazioni fra gli use case

Le relazioni fra gli use case possono essere **dipendenze**, a loro volta divise in **include** ed **extend**, oppure **generalizzazioni**.

- **Dipendenze:**
  - **Include:** Si usa per un complesso flusso di eventi che si vuole decomporre in flussi più semplici, i quali verranno eseguiti nella loro interezza in quello "padre". Può anche essere usato per riutilizzare parti abbastanza generiche di un flusso di eventi, permettendo a più use case di includerne uno stesso.

  - **Extend:** Questa relazione estende un flusso di eventi che già è completo, aggiungendo ulteriori funzionalità.

- **Generalizzazione:** Astrazione di un comportamento comune fra due casi d'uso. Il caso d'uso generale non ha di per sé un flusso di eventi, viene sempre istanziato come una delle specializzazioni.

2.7 Requisiti

2.7.1 Categorie di requisiti non funzionali

I requisiti non funzionali possono essere organizzati secondo il seguente modello:

- **Usabilità:** È la facilita con cui un utente novizio impara a preparare gli input e a interpretare gli output del sistema o componente.

- **Affidabilità:** Capacità di un sistema di eseguire le sue funzioni sotto certe condizioni e per un certo periodo di tempo. Questo può comprendere questioni di sicurezza per esempio (E2EE, etc). Recentemente rimpiazzato da **dependability**, che include anche **robustezza** (capacità di un sistema di un sistema di resistere a input non corretti) e **sicurezza**

- **Performance:** Misure quantificabili di efficienza del sistema, come throughput, tempo di risposta, etc.

- **Supportabilità:** Flessibilità al cambiamento e all'evoluzione del sistema.

2.7.2 Categorie di pseudo-requisiti

- **Requisiti di implementazione:** Uso di certi linguaggi o tecnologie.

- **Requisiti di interfaccia:** Riguardano sistemi esterni (per esempio legacy) e l'interazione con questi.

- **Requisiti sulle operazioni:** Vincoli sull'amministrazione del sistema e la sua gestione (come fare backup, etc).

- **Packaging requirements:** Requisiti riguardanti l'installazione del sistema da parte dell'utente (installazione, training, etc).

- **Requisiti legali:** Licenze, certificazioni, etc.

2.8 Object Modeling

Analizzando il flusso di eventi possiamo identificare gli oggetti e le loro operazioni principali. Individuare correttamente le operazioni e gli oggetti ai quali queste appartengono è una fase importante della progettazione.

Una fase fondamentale dell'object modeling è andare a trovare gli oggetti stessi nel flusso di eventi dei vari casi d'uso. Per fare ciò possiamo avvalerci di strumenti come per esempio l'analisi testuale di Abbott.

Inoltre possiamo identificare gli oggetti per classi, come per esempio.

- **Entity Object:** Rappresentano perlopiù informazioni nel database, ma possono all'occorrenza rappresentare anche informazioni non persistenti ma di cui il programma deve tenere traccia.
- **Boundary Object:** Rappresenta l'interazione fra l'utente e il sistema.
- **Control Object:** Rappresentano i task di controllo eseguiti dal sistema. In pratica instradano correttamente gli input dell'utente.

A seconda della necessità possono essere aggiunte anche ulteriori classi di oggetti, come per esempio i Manager Object.

2.9 Dynamic Modeling

L'analisi dinamica è utilizzata per descrivere il comportamento dinamico del sistema o degli oggetti, ma anche per esempio dell'interfaccia utente. Tramite questo tipo di analisi possiamo identificare classi ma soprattutto operazioni e stati delle stesse.

Sono due i tipi di diagrammi che possiamo usare in questo tipo di analisi.

- **Interaction diagram:** A loro volta suddivisi in **Sequence Diagrams**, utili per descrivere le interazioni fra gli oggetti di un sistema in base a una sequenza temporale, e **Collaboration Diagrams**, che mostrano le relazioni fra gli oggetti in maniera più evidente, ignorando però l'aspetto temporale.
- **Statechart Diagrams:** Sono macchine a stati finiti che descrivono gli stati di un determinato oggetto, specificando i possibili stati in cui può trovarsi e le transizioni tramite le quali può spostarsi fra gli stessi.

Modello Dinamico , per definizione è una collezione di statechart diagrams, uno per ogni classe che ha un comportamento dinamico rilevante. È utile per individuare e fornire metodi per l'object model.

Evento , uno **scambio di messaggi**, qualcosa che succede in un dato momento. Serve a inviare informazioni da un oggetto all'altro e possono essere raggruppati in gerarchie.

2.9.1 Sequence Diagrams

Un oggetto partecipante a un Squence Diagram è istanziato, quindi i Sequence Diagrams vanno a istanziare i più generici dasi d'uso. Essi possono inoltre essere suddivisi in:

- **Fork Diagram:** Il comportamento dinamico è molto centralizzato, solitamente in un oggetto che conosce tutti gli altri e scambia informazioni con essi.
- **Stair Diagram:** Il comportamento dinamico è più modulare e distribuito. Un qualsiasi oggetto può delegare informazioni e operazioni ad altri.

Nonostante il modello a scala sia generalmente preferibile per un'architettura a oggetti, uno non è assolutamente e sempre superiore all'altro.

2.9.2 Statechart Diagrams

Questo diagramma modella il comportamento del singolo oggetto, delineando gli stati in cui può trovarsi e le operazioni tramite cui li raggiunge, e rappresentando il tutto come una macchina a stati finiti.

Quando l'oggetto si trova in uno stato, un evento può far scattare una **transizione** e fargli raggiungere lo stato risultante. In assenza di un evento, la transizione scatta quando l'attività che l'oggetto stava eseguendo nello stato precedente viene portata a conclusione.

Tipi di operazioni I due tipi di operazioni che possono avvenire sono **activity** ed **action**. Una activity richiede tempo per essere completata, mentre una action è istantanea.

Statechart diagrams innestati Uno statechart diagram può contenere un sub-satechart diagram, i cui stati sono invisibili al super-statechart diagram. Così facendo descriviamo il comportamento di azioni non atomiche, ma in un diagramma separato. Questo ci permette di non perdere dettaglio ma al contempo di non aggiungere complessità superflua, in quanto nel diagramma di livello superiore rappresentiamo quelli di livelli inferiori come azioni atomiche.

2.9.3 Concurrency

Un'altra cosa che possiamo modellare è la concorrenza, di cui abbiamo due tipi.

- **Concorrenza di sistema:** Abbiamo vari statechart diagrams e ognuno viene eseguito in maniera concorrente.
- **Concorrenza interna agli oggetti:** Un oggetto può essere partizionato in sottoinsiemi di stati e partizioni indipendenti fra loro, e questi vengono eseguiti contemporaneamente. Ciò vuol dire che l'oggetto si trova in più stati nello stesso istante, uno per ogni sottoinsieme.

2.9.4 Modellazione dinamica dell'interfaccia utente: Navigation Path

Sono statecart diagrams, in cui gli stati sono le varie pagine o schermate dell'applicazione in cui l'utente può trovarsi, e il contenuto può essere un **mock-up** o una descrizione del contenuto di quella pagina. Le transizioni possono essere interazioni dell'utente con l'applicazione che causano il reindirizzamento a una nuova pagina o simili. (per esempio l'interruzione della connessione in un'applicazione web causerà la transizione a una pagina di errore)

Chapter 3

System Design Document

Lo scopo di questo documento è di spiegare in maniera comprensibile e semplice come la soluzione individuata (il design) può essere applicato. Per fare ciò dobbiamo adottare metodi adatti alle tecnologie a nostra disposizione (rendendo tuttavia il progetto più adatto possibile ad evolversi e adattarsi a sviluppi tecnologici futuri), ma anche tecniche ingegneristiche come il Divide & Conquer. Otteniamo ciò modellando il nostro sistema come un insieme di sottosistemi più semplici. Ciò che idealmente vogliamo ottenere è che i suddetti sistemi siano molto coesi ma debolmente accoppiati.

Il **System Design** si occupa inoltre di questioni come mapping hardware/software, concorrenza, boundary conditions, etc.

3.1 Design Goals

Abbiamo ovviamente una serie di obiettivi per il sistema, come per esempio portabilità, facilità di comprensione, performance, etc.

Questi design goals però non sono tutti uguali e indipendenti. Innanzitutto ogni **stakeholder** vuole ottenere i design goals per lui più appetibili, cosa che ovviamente non è fattibile per un gran numero di stakeholders molto omogenei perché un sistema che rispetta tutti i possibili design goals nella maniera ottima richiederebbe risorse e tempo infiniti.

Inoltre ci sono da considerare i cosiddetti **design trade-offs**. Questi sono collegamenti fra i vari design goals che fanno decrescere uno all'aumentare dell'altro. Alcuni esempi sono: **Functionality vs. Usability**, **Cost vs. Robustness**, **Cost vs. Reusability**, etc.

La scelta di uno dei due lati del trade-off dipende da molte cose, come per esempio budget, mercato, tipo di sistema da sviluppare, etc.

3.2 Decomposizione in sottosistemi

In UML, un **sottosistema** è identificato da un package, ed è un insieme di classi, associazioni, operazioni, eventi e vincoli collegati fra loro in qualche modo più o meno coeso.

Il **servizio** che un sottosistema offre sono le operazioni fornite dallo stesso, e sono specificate dall'**interfaccia** del sottosistema, che definisce i servizi da e verso il sottosistema ma non quelli interni allo stesso.

3.2.1 Layers e partizioni

La scelta dei sottosistemi da realizzare deve essere ponderata. Idealmente vogliamo che i vari sottosistemi siano internamente molto coesi ma poco accoppiati fra loro, allo scopo di diminuire la complessità, soprattutto relativamente a cambiamenti futuri.

L'**alta coesione** si ottiene quando le classi del sottosistema sono fortemente dipendenti l'una dall'altra ed eseguono task simili, mentre il **basso accoppiamento** fra i sottosistemi si ottiene con la bassa dipendenza fra gli stessi, e agevola la manutenzione.

Per ottenere questo delicato equilibrio possiamo avvalerci di **layers** e **partizioni**.

Partizioni Un sistema è partizionato quando è diviso verticalmente in sottosistemi, indipendenti fra loro o quasi, che forniscono un servizi con un livello uguale o simile di astrazione.

Due partizioni si conoscono a vicenda ma non in maniera profonda, e possono chiamarsi mutualmente.

Layers Un layer è una divisione orizzontale. Un sottosistema appartenente a un certo layer fornisce servizi a sottosistemi appartenenti a layers inferiori.

Di due layers A e B in cui A è più in alto, A chiama B a runtime, e A dipende da B a tempo di compilazione.

A tal proposito abbiamo due tipi di architettura:

- **Architettura chiusa (layering opaco):** Ogni layer può invocare operazioni solo dal layer immediatamente inferiore. Questo offre una alta manutenibilità e flessibilità, in quanto andando a modificare un layer solo quello immediatamente superiore potrebbe essere impattato.
- **Architettura aperta (layering trasparente):** Ogni layer può invocare operazioni da qualunque altro layer inferiore. Questo diminuisce la manutenibilità ma aumenta l'efficienza al runtime.

Oltre al paradigma a layer generico ce ne sono altri, fra cui:

- **Client/server:** Un server fornisce i servizi a un client in quanto provider, e non conoscendone l'interfaccia. Il suddetto client conosce l'interfaccia del server e ne usa i servizi. L'utente usa i livelli più alti dell'architettura, ossia il client.
- **Repository based:** È un particolare stile di client/server in cui c'è un sottosistema che funge da client e il repository che funge da serve. Quest'ultimo fornisce servizi per la gestione dei dati.

- **Model/View/Controller:** I sottosistemi sono **Model**, responsabile della gestione dei dati, **View**, responsabile della rappresentazione dei dati all'utente, e **Controller**, responsabile dell'instradamento dei dati inseriti dall'utente e della notifica delle modifiche al Model.

3.3 Concorrenza

La concorrenza in un sistema viene presa in considerazione per migliorare le performance.

Sappiamo che due oggetti sono inerentemente concorrenti se possono ricevere e processare eventi contemporaneamente senza interagire fra di loro, e in questo caso dovrebbero trovarsi quindi all'interno di due diversi thread di controllo. Al contrario, due oggetti che includono attività mutualmente esclusive dovrebbero trovarsi sullo stesso thread.

3.4 Mapping Hardware/Software

In questa attività decidiamo se realizzare i componenti come hardware o software, come il modello a oggetti può essere mappato su questi componenti, e ci chiediamo per esempio se è possibile aumentare l'efficienza distribuendo il calcolo su più processori, se abbiamo abbastanza spazio fisico di archiviazione, etc.

3.5 Data Management

Andiamo in questa sezione a individuare quali oggetti devono essere persistenti, cosa che può essere realizzata con **files** (semplici ed efficaci, a basso livello, ma richiedono programmazione aggiuntiva per interpretare i dati scritti), o tramite **database** (più portabile e potente ma meno efficiente).

3.6 Global Resource Handling & Security

Descrive i diritti di accesso per le varie classi di attori e le policies di sicurezza come per esempio autenticazione e crittografia.

Possiamo modellare l'accesso degli attori alle classi con una **matrice di accesso**: le righe rappresentano gli attori, mentre le colonne rappresentano le classi delle quali vogliamo regolare l'accesso. Un elemento della matrice è una lista di operazioni che possono essere eseguite da istanze di un attore su un'istanza della classe.

3.7 Decisioni sul controllo del software

Si sceglie il tipo di controllo: **Implicito** (non procedurale, linguaggi dichiarativi) o **esplicito** (linguaggi procedurali). Quest'ultimo può a sua volta essere **centralizzato** o **decentralizzato**.

Un controllo esplicito centralizzato si divide a sua volta in **procedure driven** o **event driven**

3.8 Boundary Conditions

Servono a regolare casi d'uso relativi a inizializzazione del sistema, shutdown e failure di vario tipo.

Chapter 4

Object Design Document

L'**Object Design** va ad aggiungere a ciò che abbiamo prodotto durante il RAD. Inoltre è in questo documento che si iniziano a prendere decisioni più vicine all'implementazione. Definiamo cose come le API, gli algoritmi che caratterizzano una classe, le strutture dati da usare, dettagli di ottimizzazione e tutto ciò che ha a che fare con l'implementazione.

Un sistema è fatto di **application objects** e **solution objects**. Durante l'analisi dei requisiti noi abbiamo colmato il requirements gap (a partire dal problema abbiamo individuato gli oggetti così come sono visti nell'ambito del dominio applicativo, e quindi come sono visti dal nostro cliente); nel System Design ci siamo occupati dell'architettura del nostro sistema (anche l'architettura fisica, vicina alla macchina, off-the-shelf components ecc.), l'infrastruttura che ci consente di implementare il nostro sistema. Nell'OD noi colmiamo il gap tra quanto fatto nell'AR e quanto fatto nel SD. Andiamo quindi a costruire del custom objects, degli oggetti del dominio delle soluzioni, e degli algoritmi che li vanno a caratterizzare.

Distinguiamo quattro tipi di attività all'interno dell'object design: **riuso**, **specifica**, **ristrutturazione**, **ottimizzazione**. Esse vanno eseguite come segue:

Dapprima ci occupiamo di riuso e/o specifica, sequenzialmente, contemporaneamente o iterando. Nel **riuso** andiamo a individuare componenti che possono essere riutilizzati, con o senza modifiche, siccome potrebbe capitare di non volere o poter effettuarne. Ciò è anche dato dall'origine di tali componenti; esse potrebbero certamente essere interne al progetto, ma anche appartenenti ad altri nostri progetti o addirittura progetti che non ci appartengono, nel qual caso si va a parlare di utilizzo di componenti off the shelf o servizi, per i quali è raramente possibile fare modifiche.

Durante l'attività di **specifica** andiamo a individuare componenti od operazioni mancanti, la loro visibilità, il tipo e le signature delle operazioni, vincoli ed eccezioni.

Queste due attività verranno sincronizzate e confluiranno nelle attività seguenti, che anch'esse potranno essere eseguite in qualsiasi ordine. Questo porta alla **ristrutturazione**, che comprendono la revisione dell'ereditarietà, il collasso delle classi (che possono essere collassate da classi a semplici attributi di altre classi per esempio) e la realizzazione delle associazioni. L'attività restante è lo studio dell'**ottimizzazione**.

4.1 Riuso

Lo scopo del riuso di codice e componenti è sfruttare conoscenze e funzionalità senza doverle riprogettare o reimplementarle. Può essere usato con:

- **Composition (black box reuse):** Ottengo funzionalità da componenti esistenti aggregandoli ad altri.
- **Inheritance (white box implementation):** Vedo cosa vado a riutilizzare nel dettaglio e ne conosco il funzionamento. Posso farlo tramite:
  - **Implementation o class inheritance:** Estendo funzionalità da una classe padre. Quest'ultima è quindi una classe che **esiste**, con alcune o tutte le operazioni già implementate, da cui vado a ereditare.
  - **Interface inheritance o subtyping:** Eredito da una classe *astratta* che specifica ma non implementa le sue funzionalità. Devo implementarle.

  Mi conviene usare la **class inheritance** quando c'è una relazione tipo-sottotipo fra le due classi. Questo vuol dire che se eredito l'implementazione di un metodo, la classe che sto realizzando deve avere nella sua specifica anche l'operazione che sto ereditando. Ciò non avviene per esempio se cerco di implementare uno `Stack` da una classe `List`. Se l'utente andrà a usare l'operazione `remove()` anziché `pop()`, questo potrebbe causare problemi. Mi conviene, in casi come questo, incapsulare la `List` all'interno della nuova classe, usando per esempio una variabile d'istanza di tipo `List`. Questo è possibile tramite la **Delegation**.
- **Delegation:** Abbiamo un oggetto receiver e un oggetto delegato. Il client chiama il receiver, il quale delega al delagato il compito di elaborare dati.

4.2 Design Patterns

Prima di occuparci del vero e proprio object design è bene conoscere almeno qualche design pattern.

Cosa è un design pattern Se andiamo a modellare in maniera rigida la realtà, andiamo a riflettere perfettamente quella di oggi ma non necessariamente bene quella di domani. Un **design pattern** descrive la soluzione a un problema che ricorre spesso. Può essere dunque utilizzato molte volte, in molti contesti e offre un livello di astrazione utile.

Pattern strutturali, comportamentali, creazionali Vedremo alcuni design patterns, che possiamo dividere in:

- **Pattern strutturali:** Sappiamo come si comportano a runtime, il loro comportamento è una diretta conseguenza della loro struttura. Vedremo i pattern **Adapter**, **Bridge** e **Proxy**.

- **Patten comportamentali:** Oltre alla definizione della loro struttura, essi hanno un comportamento che dobbiamo descrivere, cosa che spesso si fa con i sequence diagrams. Andremo a vedere i pattern **Command**, **Observer** e **Strategy**.

- **Pattern creazionali:** Automatizzano e astraggono la creazione di oggetti, rendendola trasparente all'utente o client o aumentandone l'efficienza. Vedremo i pattern **Abstract Factory** e **Builder**.

4.2.1 Composite Pattern

Se abbiamo il problema di dover modellare una struttura che può essere ricorsivamente ottenuta (quindi di cui non possiamo conoscere con certezza la profondità), ci avvaliamo del **composite pattern**.

Esso compone oggetti in strutture ad albero che permettono al client di decidere in maniera dinamica se l'oggetto corrente è una foglia o meno. Si può modellare in questo modo per esempio anche il CVS (una attività può essere atomica, quindi un task, o essere diviso in sottoattività a sua volta).

Un altro esempio dell'utilità di questo pattern è la modellazione grafica di menù, siccome una entry del menù potrebbe essere la radice di un sottomenù oppure un'opzione atomica.

4.2.2 Facade

Un sottosistema consiste di oggetti che rappresentano entità del mondo e un'interfaccia che mostra le informazioni utili all'esterno del sottosistema. Suddetta interfaccia può essere realizzata con un **facade**. Questo design pattern fornisce un'interfaccia di alto livello fornendo un accesso alle operazioni del sottosistema, ma senza necessariamente fornire accesso agli elementi e dettagli interni del sottosistema. Il vantaggio ovvio di questo design pattern è quello di diminuire l'accoppiamento, siccome l'accesso al sottosistema avviene in maniera indiretta, tramite un singolo oggetto, il quale è più manutenibile. Un altro effetto dell'utilizzo di questo deisgn pattern è l'aumento della comprensione delle funzionalità del sottosistema. Un pattern **facade** potrebbe essere poco desiderabile, per esempio nel caso di un sistema che ha bisogno di essere molto performante, quindi permettendo a ogni client di accedere alle funzionalità di ogni sottosistema, ma vengono a mancare comprensibilità e sicurezza.

4.2.3 Adapter e Bridge patterns

Quando andiamo a selezionare componenti per il riuso, essi possono essere componenti esistenti, presi da librerie o off the shelf. In questi casi possiamo adattarli ai nostri scopi andando a modificare l'API se abbiamo il codice sorgente, o usando un design pattern come **adapter** o **bridge**. Vado dunque a modificare l'interfaccia del componente che vado a usare per farla "matchare" con l'interfaccia dei componenti che devo realizzare, oppure uso uno schema di progettazione.

Le **somiglianze** fra adapter e bridge è che entrambi sono usati per nascondere i dettagli dell'implementazione sottostante. La **differenza** è che l'adapter consente a componenti che hanno interface diverse di lavorare insieme, mentre il bridge ci consente di realizzare diverse implementazioni da una stessa astrazione, e anche di farle evolvere e variare indipendentemente da essa. In questo modo il client è concentrato sull'interfaccia, l'implementazione può variare e di essa se ne occupa solo la classe intermedia. È utile quando si vuole realizzare un sistema estendibile, al quale aggiungere componenti che non sono ancora noti in fase di analisi o progettazione.

Adapter

Noto anche come **wrapper**, converte l'interfaccia di una classe in un'altra interfaccia che il client si aspetta, così da rendere le due classi compatibili tramite l'adapter anche se di base non lo sono. Esistono due tipi di adapter:

- **Class adapter:** Usa l'ereditarietà multipla per adattare un'interfaccia a un'altra.
- **Object adapter:** Più comunemente usato, serve ad adattare componenti scritte in un linguaggio di programmazione diverso da quello utilizzato, e si avvale di ereditarietà singola e delegation.

Funzionamento: Abbiamo una classe `Target` che vogliamo realizzare, ma abbiamo trovato una classe `Adaptee` che fa la stessa esatta cosa di `Target` ma ha un'interfaccia diversa. Utilizzo dunque un Adapter che avrà la stessa interfaccia d `Target` (e infatti potrei non implementare `Target` ma lasciarlo come interfaccia). `Client` avrà al suo interno un'istanza di `Target`, il tipo statico sarà quello di `Target`, ma il tipo dinamico sarà quello di `Adapter`, perché tra `Adapter` e `Target` vige una relazione di tipo-sottotipo. `Adapter` al suo interno dovrà avere come variabile d'istanza un'istanza di `Adaptee`, che userà per convertire il metodo di `Adaptee` (ExistingRequest) nel metodo di `Target` (Request). `Request()` sarà implementato nell'adapter andando a chiamare `ExistingRequest()` sulla variabile d'istanza di `Adaptee`.

Se `Adaptee` è un elemento scritto in un linguaggio diverso ovviamente sarà più difficile adattarlo. Comunque, il vantaggio dell'adapter è che se trovo poi una soluzione migliore da adattare, posso implementarla nell'adapter (all'interno del quale confino tutte le modifiche) senza andare a cambiare tutto nel client.

Bridge

Noto anche come Handle/Body pattern, viene usato per "disaccoppiare un'astrazione dalla sua implementazione, in modo tale che esse possano variare indipendentemente". Permette dunque di decidere, anche dinamicamente, quale implementazione dell'interfaccia voglio usare, ed è utile per sviluppare interfacce verso componenti che sono incomplete, non ancora note o non disponibili durante il testing.

Ad esempio, supponiamo di avere un sistema "VIP" che deve usare l'interfaccia di "Seat". Quello che farò sarà implementare l'interfaccia di Seat attraverso una gerarchia di classi, "SeatImplementation" che fornisce diverse implementazioni per Seat. SeatImplementation è una interfaccia, e può decidere anche dinamicamente quale implementazione usare tra Stub Code, AIMSeat e SARTSeat. Ad esempio Stub Code può essere utilizzata nel caso Seat non fosse ancora implementata, per il testing.

SeatImplementation fornisce le stesse operazioni di Seat, e io evito al client di sapere qual è l'implementazione che sto utilizzando (Stub Code ecc. implementeranno SeatImplementation). Avremo dunque sempre almeno 2 classi: l'interfaccia (in questo caso Seat) che viene fornita all'esterno e l'implementazione, che consiste in una gerarchia di classi (con un'interfaccia e diverse implementazioni).

Seat avrà una variabile d'istanza imp di tipo SeatImplementation, su cui vengono implementati i metodi dal client. Chiaramente prima di potervi invocare i metodi dovrò istanziare SeatImplementation, e potrò farlo o con un oggetto StubCode, o con un oggetto AIMSeat o con un oggetto SARTSeat. Insomma, con un qualunque oggetto il cui tipo è una delle sottoclassi di SeatImplementation. Quindi ho più implementazioni per una stessa astrazione, il tutto in maniera trasparente al client (VIP). Il trucco è quello di non avere direttamente nel client la variabile di tipo SeatImplementation, perché altrimenti è lui a decidere quale implementazione utilizzare. Per questo usiamo un oggetto intermedio (in questo caso Seat) che fornisce l'interfaccia a VIP, ma non l'implementazione, nascosta all'interno di Seat. Ovviamente questo è meno efficiente perché devo fare due chiamate, ma abbiamo una maggiore trasparenza.

4.2.4 Proxy

La creazione e/o inizializzazione di un oggetto può talvolta essere costosa. Prendiamo per esempio un browser web e una connessione piuttosto lenta. Quando carichiamo una pagina gli elementi più costosi da scaricare sono quelli multimediali, come le immagini.

Ebbene, un oggetto **Proxy** non è altro che un fantoccio, un oggetto molto poco costoso da istanziare che verrà caricato al posto di quello costoso ma meno necessario (in questo caso l'immagine). Quando l'istanziazione dell'oggetto reale sarà necessaria, sarà il proxy stesso a eseguire tale compito.

4.2.5 Command

Un'azione complessa da implementare è il comando **undo**. Un modo semplice ed efficace per fare ciò è con il design pattern **command**. Una generica azione non viene eseguita da un **Action Performer** ma bensì viene incapsulata in un oggetto **command**. Questo oggetto command prima di eseguire l'azione memorizza lo stato. Così facendo quando vogliamo annullare un comando basta chiamare il metodo **undo()** dell'oggetto command.

4.2.6 Observer

Anche chiamato **"Publish and Subscribe"**, stabilisce una dipendenza uno a molti tra oggetti in maniera che se un oggetto cambia stato, i suoi dipendenti verranno notificati e aggiornati automaticamente.

Viene utilizzato per mantenere la consistenza fra stati ridondanti e ottimizzare cambiamenti in batch per mantenere la consistenza.

4.2.7 Strategy

Supponiamo che esistano molti algoritmi diversi fra loro per la risoluzione di uno stesso task (come per esempio ordinamento di un array), e magari alcuni sono più appropriati di altri in momenti diversi dell'esecuzione o dello sviluppo. Bene, noi vogliamo poter decidere dinamicamente quale algoritmo utilizzare, oltre che aggiungere algoritmi a questa lista di potenziali algoritmi, e il tutto in maniera completamente trasparente.

Risulta in un certo senso simile al previamente visto **Bridge Pattern**, siccome ha diverse implementazioni dello stesso task. La differenza è che non stiamo parlando di oggetti che possono avere la stessa interfaccia bensì di algoritmi, di **operazioni** che possono avere differente implementazione.

4.2.8 Abstract Factory

Prendiamo qualcosa come un'interfaccia utente, che ha diversi **Looks and Feels**, cosa che vediamo abbastanza spesso in quasi tutti i software dotati di interfaccia utente. Ciò che voglio fare è indicare quale Look preferisco e istanziare automaticamente gli oggetti giusti (icone, scrollbar, sfondi, palette di colori), anziché lasciare all'utente il compito di dover selezionare ogni singolo dettaglio. La stessa cosa vale per esempio per l'installazione di un software su di un sistema operativo, voglio installare i driver corretti avendo solo l'informazione relativa al sistema operativo.

Il client, per ovviare a questo problema, ha al suo interno una variabile d'istanza di tipo `AbstractFactory`, la quale ha diverse implementazioni che chiameremo per esempio `ConcreteFactory`. Il client deve sapere quali sono queste `ConcreteFactory`, ovviamente deve conoscere l'interfaccia di `AbstractFactory` e deve anche conoscere l'interfaccia dei vari prodotti (`AbstractProduct1, AbstractProduct2`...), senza tuttavia sapere come

sono implementati i prodotti, in quanto è proprio questo il processo che vogliamo delegare alla **Factory** in modo trasparente.

4.2.9 Builder Pattern

Questo pattern è in un certo senso complementare all'**Abstract Factory**. Serve a rendere indipendente e trasparente il processo di inizializzazione rispetto all'utilizzatore.

Per esempio, per installare un sistema che abbiamo creato dovremo andare a inizializzare e installare tutta una serie di componenti, e non vogliamo far decidere all'utente quali è assolutamente necessario installare o in che ordine, quindi andiamo a creare un'operazione che andrà a specificare proprio queste cose. Tuttavia questa operazione sarà più semplice ed al alto livello, e quindi necessiterà di una singola invocazione da parte dell'utente.

4.3 Specifica delle Interfacce dei Sottosistemi

Nella fase di **System Design** abbiamo definito i Servizi dei moduli, poi abbiamo dettagliato la specifica sia da un punto di vista sintattico (tipi, parametri), sia semantico (pre e post condizioni).

Abbiamo diversi tipi di utenti che possono utilizzare una classe:

- L'**implementatore** realizza la classe, ne progetta le strutture dati e definisce le interfacce delle operazioni pubbliche, oltre che ovviamente implementarne il codice.

- L'**utente** invoca le operazioni della classe utili alla realizzazione di un'altra classe detta classe client. L'interfaccia della classe utilizzata definisce il modo in cui può essere usata, mostra il boundary della classe in termini di servizi offerti.

- L'**extender** sviluppa specializzazioni della classe. La specifica delle interfacce specifica il comportamento corrente della classe. L'extender può porre dei vincoli sulla classe specializzata rispetto a quella originale, andando così a modificarne l'interfaccia.

Dunque, durante l'**Analisi dei Requisiti** sono stati identificati gli attributi delle operazioni senza specificare il loro tipo e ponendo minima attenzione sui parametri. Abbiamo poi proseguito, con il **System Design** a rendere le operazioni parte di servizi e quindi dei vari moduli che compongono il nostro sistema. Durante l'**Object Design** ci occupiamo di aggiungere **informazioni di visibilità**, **informazioni sulla signature** e sui **tipi**, e aggiungere i **contratti** (la specifica semantica delle operazioni).

4.3.1 Informazioni sulla visibilità

UML definisce 3 livelli di visiblità:

- **Private** (mirato all'utilizzo da parte del class implementator), si indica ponendo il simbolo "-" prima del membro della classe, ed è utilizzato per attributi e operazioni che trovano utilità all'interno della classe in cui sono inseriti.

- **Protected** (mirato all'utilizzo da parte del class extender), si indica inserendo il simbolo "#" prima del membro della classe, ed è utilizzabile all'interno della classe ma anche dai suoi discendenti (relazione extends).

- **Public** (mirato all'utilizzo da parte del class user), si indica con il simbolo "+" prima del membro della classe, ed è utilizzato per attributi o operazioni accessibili da qualunque altra classe.

L'utilizzo di terminologie così vicine alla programmazione object oriented rende l'operazione di mappatura da UML a Java abbastanza automatica.

Euristiche per l'information hiding

Occorre definire attentamente le interfacce pubbliche delle classi così come per i sottosistemi, il che è un modo per costruire un firewall intorno alle classi.

Applicare sempre il principio **"need to know"**: solo se qualcuno ha bisogno di conoscere informazioni esse si rendono pubbliche, ma dovrebbero comunque essere fornite attraverso dei canali (per esempio con operazioni `get` e `set` anziché fornire accesso diretto agli attributi). L'idea è che meno si conosce dei funzionamenti interni dell'operazione e minori saranno le probabilità di essere influenzati da un cambiamento, e dunque sarà più facile modificare una classe quando necessario. Per esempio alcune informazioni potrebbero essere rappresentate in modo diverso all'interno di come sono presentate esternamente. Un esempio è memorizzare l'orario come un numero intero rappresentante i secondi ma poi elaborarlo per mostrarlo come ore, minuti e secondi all'esterno.

Il trade-off in questo caso è information hiding vs efficienza, visto che per accedere a un attributo privato può essere necessario più tempo.

Principi di Design dell'information hiding

Solo le operazioni di una classe possono modificare i suoi attributi (quindi si accede agli attributi solo tramite le operazioni).

Nascondiamo il contenuto degli oggetti tramite il suo confine, il suo boundary (per esempio definendo classi astratte che fanno da mediatore fra il sistema e il mondo esterno e i vari sottosistemi).

Non applicare un'operazione al risultato di un'altra operazione. È meglio scrivere un'altra operazione di più alto livello che le combina, così che l'utente non debba preoccuparsi di combinarle.

4.3.2 Informazioni di tipo e sulla signature

Gli attributi hanno il loro tipo, così come le operazioni hanno il tipo di ritorno e la signature completa con attributi e tipi. Gli attributi e le operazioni senza informazioni sul tipo possono essere accettabili in fase di analisi.

4.3.3 Aggiungere contratti (ossia la specifica semantica delle operazioni e delle classi)

I contratti su una classe consentono sia al chiamante che al chiamato (ossia la classe stessa) di condividere le stesse assunzioni sulla classe. Insomma, la specifica di una classe serve a far sì che sia chi deve implementare sia chi deve usare la classe abbiano **la stessa idea** di ciò che la classe fa: Chi la implementa fa in modo di fornire alle classi chiamanti certe funzionalità definite dalla specifica; chi la usa sa invece qual è il comportamento della classe specificato dai contratti.

I contratti includono tre tipi di vincoli:

- Un'**invariante**, che è un predicato che è sempre vero per tutte le istanze della classe e si applica allo stato di un oggetto. Le invarianti sono vincoli associati a classi e interfacce e sono usati per specificare vincoli di consistenza tra attributi di una classe (tutte le istanze di quella classe devono rispettare questi vincoli; se un oggetto cambia stato deve comunque farlo restando nel dominio definito dall'invariante, rispettandone i vincoli).

- Alle **operazioni** applichiamo due tipi di vincoli, il primo dei quali è la **precondizione**, che è un predicato che deve essere vero prima che l'operazione sia invocata, e quindi specifica vincoli che il chiamante deve soddisfare prima di chiamare l'operazione.

- Il secondo tipo di vincolo applicato alle **operazioni** è la **postcondizione**, la quale è un predicato che deve essere vero dopo che l'operazione è stata invocata, e specifica vincoli che l'oggetto deve rispettare dopo la chiamata dell'operazione.

4.3.4 OCL: Object Constraint Language

Per esprimere vincoli in UML si usa il linguaggo OCL, un linguaggio logico e formale. OCL permette di specificare formalmente vincoli su un singolo elemento o su gruppi di elementi di un modello, e un vincolo è espresso come un'espressione OCL che restituisce il valore **vero** o **falso**. OCL è un linguaggio dichiarativo, non procedurale (non possiamo vincolare il control flow).

Esempio , di cui poi andremo a spiegare ed analizzare il significato delle varie parti: Operazione `put` per una `HashTable`.

Invariante: `context HashTable inv: numElements >= 0`

Precondizione: `context HashTable::put(key, entry) pre:!containsKey(key)`
Postcondizione: `context HashTable::put(key, entry) post:containsKey(key) and get(key) = entry`

Context mi dice la classe a cui si fa riferimento, mente `::` è l'**operatore di scope** (utilizzato anche in C++, indica un'operazione di una classe, in questo caso mi va a dire che `put()` è un'operazione di `HashTable`; il punto, come siamo abituati in Java, va a indicare invece l'operazione per quella specifica istanza) e poi si indica l'operazione su cui stiamo andando a stabilire i vincoli.

4.3.5 Ereditarietà dei contratti

Quando si estende una classe, cosa succede ai contratti? Un class user si aspetta che i contratti che valgono per una superclasse valgano anche per la sottoclasse. Cosa succede dunque quando faccio **overriding**?

- **Precodnizione**: Un metodo di una sottoclasse può **indebolire** la precondizione del metodo che sovrascrive (il metodo della sottoclasse può avere a che fare con più casi). Per esempio prendiamo in considerazione un metodo che ha a che fare con numeri positivi, quindi la cui precondizione prevede che `n > 0`. Bene, se sovrascrivo questo metodo posso pensare di gestire anche il numero 0, quindi avrò per la sottoclasse una precondizione del tipo `n >= 0`.

  Perché possiamo indebolire la precondizione ma non restringerla? Perché altrimenti la sottoclasse non potrebbe gestire input altresì possibili per la superclasse, e questo sarebbe un comportamento anomalo in quanto la sottoclasse è anche un oggetto del tipo della superclasse e deve rispettarne i vincoli. Può indebolirla perché quando sono nel contesto della sottoclasse so come gestire i valori di input nuovi, mentre nel contesto della superclasse avrò solo input appartenenti al range di valori ristretti, che saprò ugualmente come gestire perché sono previsti dalla specifica della superclasse.

- **Postcondizione**: I metodi della sottoclasse devono assicurare le **stesse postcondizioni** della superclasse. Questo per lo stesso discorso del punto precedente: Se l'oggetto della sottoclasse non rispetta la postcondizione avrà un comportamento anomalo, dato che il programma si aspetta un comportamento la cui specifica è definita dalla superclasse, non dalla sottoclasse.

- **Invarianti**: Una sottoclasse deve rispettare tutte le invarianti della superclasse. Tuttavia, una sottoclasse può **restringere** le invarianti ereditate (quindi i possibili stati in cui può trovarsi un oggetto della sottoclasse sono un sottoinsieme degli stati in cui può trovarsi un oggetto della superclasse).

  Perché? Consideriamo il caso in cui il tipo della variabile è della superclasse ma andiamo a istanziare un oggetto della sottocalsse: stiamo rispettando comunque

l'invariante della superclasse in quanto ci troveremo in un sottoinsieme degli stati ammessi dalla sua invariante. Il contrario non sarebbe tuttavia contrario, con invarianti più permissive potremmo ritrovarci con stati anomali e ingestibili.

4.4 Caratteristiche dell'ODD

Il documento tramite il quale andiamo a documentare l'Object Design è detto **Object Design Document (ODD)**. Esso è molto simile al RAD, ma vengono aggiunti oggetti, modelli dinamici e funzionali dal dominio delle soluzioni, navigational map per il modello a oggetti e documentazione Javadoc per tutte le classi (dove specifico l'interfaccia delle classi, precondizioni, postcondizioni e invarianti). Nonostante le somiglianze con il RAD esso è un documento separato, e non una rifinitua di quest ultimo.

4.4.1 Convenzioni per l'ODD

Ogni sottosistema in un sistema offre un servizio (descrive l'insieme di operazioni offerte dal sistema); l'operazione di un servizio si specifica con:

- **Signature**: Nome dell'operazione, lista di parametri con relativi tipi e tipo di ritorno
- **Abstract**: Descrizione dell'operazione
- **Pre**: Precondizione per chiamare l'operazione
- **Post**: Postcondizione per descrivere lo stato dopo l'esecuzione dell'operazione

Per la specifica delle operazioni del servizio si usa Javadoc.

4.4.2 Packages

Andiamo a mettere tutto insieme costruendo i **packages**, dove vado a mettere le classi. Idealmente occorre utilizzare un package per ciascun sottosistema.

Come regola generale non dovremmo avere troppe classi in un solo package, con lo scopo di minimizzare l'accoppiamento massimizzando la coesione. Le classi che sono legate fra loro vanno nello stesso package, altrimenti in packages diversi. Anche i metodi non dovrebbero avere un numero eccessivo di parametri, sempre allo scopo di minimizzare l'accoppiamento.

Euristiche per il packaging . Ogni servizio del sottosistema è reso disponibile da uno o più oggetti interfaccia all'interno del package. Si inizia con un oggetto interfaccia per ciascun servizio del sottosistema, cercando di limitare il numero di operazioni dell'interfaccia (range indicativo 5-10). Se il servizio del sottosistema ha troppe operazioni, riconsideriamo il numero di sottosistemi in quanto questo potrebbe essere un buon indicatore del fatto che abbiamo accorpato due sottosistemi che avrebbero invece bisogno di essere separati.

4.5 Mapping models to code

Vedremo ora in maniera sistematica come i costrutti UML vengono mappati su codice, avendo come rifetimento il linguaggio Java. Quindi, abbiamo visto il riuso nella scelta dei componenti (off-the-shelf, oggetti di soluzione addizionali e design patterns) e la specifica dei servizi dei moduli tramite la descrizione dettagliata di ogni class interface. Adesso ci concentriamo sul Mapping dei modelli sul codice, che trasforma il modello di object design in codice, tenendo conto di criteri di perfomance e cercando di migliorare la comprensibilità, e vedremo come mapparlo in codice sorgente e schemi di memorizzazione.

Possiamo dividere le **attività di trasformazione e mapping** in **ottimizzazione** (teniamo in considerazione i requisiti di performance del sistema nell'implementazione del modello); **realizzazione delle associazioni** (mappandole a costrutti del codice sorgente, come riferimenti e collezioni di riferimenti); **mappare i contratti sulle eccezioni** (descriviamo il comportamento delle operazioni quando i contratti sono violati), anche se non è l'unico modo di mappare le violazioni delle precondizioni; **mappare i modelli delle classi su uno schema di memorizzazione** (in particolare su un DB relazionale).

Applicare le trasformazioni a un modello vuol dire cercare di modificare un modello per crearne un altro che vada incontro ad altri obiettivi (ad esempio semplificare, ottimizzare o rendere il modello più vicino alle specifiche). Ad esempio convertire un attributo in una classe, o aggiungere/togliere/rinominare classi, operazioni associazioni o attributi.

Si ristruttura anche la gerarchia delle classi per migliorare l'ereditarietà. **Aumentare l'ereditarietà** significa riarrangiare e aggiustare classi e operazioni per prepararsi meglio all'ereditarietà; astrarre comportamenti comuni delle specializzazioni per portarli nella generalizzazione (quindi parto con l'individuazione di una superclasse, costruisco le relazioni e l'ereditarietà - le sottoclassi saranno specializzazioni di questa superclasse - e porto in alto gli attributi e le operazioni comuni alla sottoclasse). Questa operazione può essere fatta all'interno di un sottosistema e viene chiamata anche **Exrtact Superclass Refactoring** (se effettuata nell'ambito del codice). Innanzitutto ci si prepara all'inheritance: non solo creo la superclasse, ma mi assicuro che tutte le operazioni sono praticamente le stesse (potrei avere due classi diverse e chiamare due operazioni che fanno praticamente la stessa cosa ma in maniera diversa). Magari in certi casi alcune operazioni hanno meno argomenti, quindi va usato l'overloading. Devo poi rinominare gli attributi che hanno lo stesso significato ma hanno nomi diversi, o ancora posso usare funzioni virtuali e overriding per definire operazioni definite in una classe ma non in un'altra. Poi posso astrarre gli attributi e il comportamento comune (insieme di operazioni con la stessa signature) e creare una superclasse a partire da essi. L'ereditarietà comunque si applica per migliorare modularità, estensibilità e riusabilità.

Ci sono diversi tipi di trasformazione che possono essere applicati nell'OD:

- Il primo tipo è la **trasformazione di un modello**, per esempio per migliorare l'ereditarietà applichiamo una trasformazione sul modello a oggetti. Ciò ci porta da un modello

nello spazio dei modelli a un altro modello sempre nello spazio dei modelli. Ma nell'OD abbiamo anche lo spazio del codice sorgente, dove iniziamo a implmentare e raffinare i nostri modelli.

- La tecnica che si applica nel codice sorgente per modificarne un elemento restando tuttavia in quello stesso spazio viene chiamata **Refactoring**. Il codice sorgente e il modello dovrebbero comunque essere sincronizzati.

- L'operazione che consiste nello scrivere codice basandosi sul modello viene chiamata **Forward Engineering**.

- Viceversa, quando andiamo ad astrarre un modello partendo dal codice parliamo di **Reverse Engineering**.

4.5.1 Ottimizzazione

L'attività di ottimizzazione è importante perché spesso il documento di analisi dei requisiti è semnaticamente corretto ma inefficiente se implementato direttamente. Le attività di ottimizzazione durante l'OD si possono dividere in:

Ottimizzazione dei cammini d'accesso

- Si aggiungono associazioni ridondanti per minimizzare il costo d'accesso di un attraversamento ripetuto di associazioni (si va a vedere quali sono le operazioni più frequenti, quanto spesso l'operazione è chiamata e si analizzano i sequence diagrams per vedere se un'operazione richiede molti attraversamenti).

- Si riduce la ricerca riducendo associazioni "molti a molti" o "uno a molti" in associazioni "uno a molti" o "uno a uno" quando possibile usando associazioni qualificate (con qualifiers).

- Si riarrangiano gli ordini di esecuzione, eliminando i dead path o invertendo l'ordine di alcuni loop che possono essere meno costosi.

- Spostare gli attributi coinvolti solo in operazioni di get e set in classi che chiamano questi attributi.

Collassare classi in attributi

Questo porta alla trasformazione di oggetti in attributi. La scelta è fra implementare un'entità come attributo embedded all'interno di un'altra entità oppure come classe separata con associazioni ad altre classi (le associazioni sono più flessibili e rendono meglio i concetti del dominio applicativo, ma spesso introducono delle indirezioni che non sono necessarie).

Una buona indicazione di una classe che potrebbe essere collassata in un attributo è quando le uniche operazioni definite sugli attributi sono `get` e `set`. È una cosa che non va fatta in fase di analisi perché bisogna avere chiaro quali sono le responsabilità di ogni classe, in genere questa decisione va fatta quando l'implementazione è già cominciata e si vuole mantenere la consistenza tra modello e codice.

Caching dei risultati di computazioni costose

Si memorizza in attributi derivati, definendo classe o attributi che mantengono informazioni localmente. Il problema sorge quando gli attributi derivati (i cui valori dipendono da valori di base) devono essere aggiornati e sincronizzati. Ci sono 3 modi:

- **Codice Esplicito (push)**: Quando cambia l'attributo indipendente viene settato un flag update ad OFF (ci indica che il valore derivato non è aggiornato), quindi ogni volta che voglio usare questo valore controllo questo flag per vedere se è aggiornato e nel caso non lo sia lo aggiorno. Il valore derivato viene dunque aggiornato solo quando necessario.

- **Active Value (notification, data trigger)**: Ogni volta che vengono modificate le variabili indipendenti, quella derivata viene aggiornata.

- **Computazione Periodica (pull)**: L'attributo derivato viene calcolato occasionalmente.

Quale dei tre è meglio? Dipende dalla **frequenza** con cui vengono usati gli attributi indipendenti e quello derivato. In fenerale questi approcci non sono mutualmente esclusivi ma possono essere combinati, per non appesantire il sistema e mantenere coerenza fra i valori.

Ritardare calcoli costosi

Si usa un Proxy Design Patter... Bada.

4.5.2 Come realizzare le Associazioni

Devo cercare di essere più **uniforme** possibile e fare in modo che le decisioni vengano prese su **una associazione alla volta**. Come esempi di implementazione uniforme: Se ho un'associazione 1-a-1, utilizzo i nomi dei ruoli nell'associazione come attributi nelle classi, quindi li trasformo in references; Se ho un'associazione 1-a-molti la trasformo in un set; Se ho un'associazione qualificata la trasformo in una HashMap, perché l'attributo dell'associazione qualificata lo uso come chiave per accedere agli elementi dell'altra classe.

4.5.3 Come trasformare i contratti su eccezioni

Alcuni linguaggi OO includono anche i vincoli, in modo da controllare i contratti e sollevare le eccezioni nel momento in cui il contratto è violato, ma Java non è uno di quelli, rendendo l'introduzione a questa sezione funzionalmente inutile, thanks Obama. In Java dobbiamo definire le classi eccezione, controllare il contratto usando i costrutti del linguaggio di programmazione e lanciare l'eccezione con la keyword `throw` seguito da un oggetto eccezione. Quindi per implementare una precondizione, il metodo che ha quella precondizione solleverà un'eccezione nel momento in cui essa viene violata (throw). Le classi client che usano quel metodo sanno che essa può essere sollevata ed eventualmente la gestirando con un blocco trt-catch.

Per implementare un contratto, idealmente per ogni operazione di un contratto devo: controllare la precondizione prima di iniziare un metodo con un test che potrebbe sollevare un'eccezione se non è soddisfatta; controllare la postcondizione alla fine dei metodi e lanciare un'eccezione se il contratto è violato; controllare le invarianti quando si controlla la postcondizione; avere a che fare con l'ereditarietà incapsulando il codice di controllo per precondizioni e postcondizioni in metodi separati che possono essere chiamati dalle sottoclassi.

Usare sistematicamente questo approccio assicura la verifica di ogni precondizione, postcondizione e invariante di ogni metodo, portando a un sistema robusto, ma questo approccio non è realistico per vari motivi: Sforzo di codifica eccessivo, introdurremmo difetti (chi ci dice che questi controlli non siano difettosi?), andremmo a offuscare il codice e avremmo problemi di performance. Quindi quello che non si fa è controllare il codice per postcondizioni e invarianti (di solito questa porzione di codice è ridondante rispetto a quello che implementa le funzioni della classe -in effetti un codice che controlla le postcondizioni è quasi identico a quello che implementa le funzionalità e non è di molto aiuto per scovare bug, esegue un compito che può essere meglio svolto dal **testing**-); in ogni caso è utile testare le **precondizioni** a livello di implementazione (testare e sollevare eccezioni, in modo che possano essere catturate e gestite), per postcondizioni e invarianti ci affidiamo invece ai casi di test. Se più metodi hanno le stesse precondizioni, il codice che le controlla può essere incapsulato in metodi separati. In più è opportuno commentare il codice che effettua i controlli.

4.5.4 Come mappare il modello delle classi su uno schema di memorizzazione

Volendo potrei usare un database object-oriented, che supporta tutti i concetti della programmazione object-oriented, tuttavia non sono molto usati, quindi devo mappare il modello di cui dispongo su uno schema relazionale.

Un databse relazionale è basato sull'algebra relazionale, i dati sono organizzati in tabelle, che hanno un certo numero di colonne e hanno tante righe quanti sono i record memorizzati all'interno di quella tabella. Usiamo il concetto di primary key (per identificare univoca-

mente una voce in una tabella) e foreign key (attributo in una tabella in riferimento alla chiave primaria di un'altra tabella). Per manipolare le tabelle si usa SQL. Sono supportati dei vincoli, come l'integrità referenziale, cioè che certi riferimenti a un'altra tabella devono esistere.

Come si mappa un modello a oggetti in un database relazionale? In maniera molto simile a quanto visto per i diagrammi ER: una classe è mappata su una tabella, gli attributi di una classe sono mappati su una colonna nella tabella, una associazione uno-a-molti è implementata con una chiave esterna, una molti-a-molti con una tabella separata, che ha associazioni 1-a-molti con le classi originali. I metodi non sono mappati. Se abbiamo inheritance ci sono due modi di mapparla: **Vertical mapping** (aggiungo una nuova tabella con il "ruolo" e l'id che si collega alle tabelle che ereditano), la cui ragione è una manutenzione più facile, siccome rispecchia meglio le classi all'interno del sistema, e **horizontal mapping** (si creano due tabelle indipendenti duplicando gli attributi della classe ereditata in tutte e due), la cui ragione è una performance migliore.

4.5.5 In conclusione

È necessario documentare le trasformazioni per mantenere la consistenza tra modelli a oggetti e il source code e per mantenere aggiornati tutti i documenti impattati. Quindi ci sono diverse responsabilità:

Il **core architect** prende le decisioni di design e sceglie le trasformazioni da applicare; L'**architecture liaison** si occupa di documentare i contratti associati alle interfacce dei sottosistemi e notificare i cambiamenti ai class users; lo **sviluppatore** si occupa di seguire le decisioni del core architect, applicando le trasformazioni, mappando gli object models al source code e mantenendo aggiornati i commenti al codice sorgente.

Insomma chi ha fatto System Design si occupa della parte di Object Design legata alle interfacce tra i moduli, e lo sviluppatore implementa i moduli, prendendo decisioni su strutture dati e algoritmi da utilizzare, e effettuando forward engineering delle trasformazioni del modello a oggetti sul codice, e viceversa per il reverse engineering, andando a informare il liaison a tal proposito. Chiaramente le trasformazioni le posso fare sul sottosistema che sto realizzando, non posso andare a impattare su sottosistemi esterni, perché andrebbe a impattare sulle interfacce dei moduli.

Chapter 5

Testing

5.1 Introduzione al Testing

Il **testing** consiste nel trovare le differenze tra il comportamento atteso, specificato tramite il modello del sistema, e il comportamento osservato nel sistema implementato. Dal punto di vista della modellazione, con il testing si cerca di mostrare che l'implementazione del sistema è inconsistente con il modello del sistema. Quindi, un po' controintuitivamente, lo scopo ultimo del testing è quello di **trovare** quanti più problemi possibili, che saranno poi corretti dagli sviluppatori.

Questa attività dovrebbe idealmente essere svolta da sviluppatori non coinvolti nella realizzazione del sistema, in quanto, in contrasto con le attività precedenti, è distruttiva nei confronti del sistema, anziché costruttiva.

I termini maggiormente utilizzati in questo contesto sono:

- **Affidabilità**: La misura di successo con cui il comportamento osservato nel sistema è conforme ad alcune specifiche del suo comportamento.
- **Failure**: Qualsiasi derivazione del comportamento osservato rispetto al comportamento atteso.
- **Fault (bug)**: La causa meccanica o algoritmica di uno stato di errore.

Ci sono molti tipi differenti di errori e molti modi di far fronte a questi. Chiaramente, prima di dire che un comportamento "strano" sia un fault, un errore o un difetto, bisogna prima specificare il comportamento desiderato.

Per far fronte a errori vari ed eventuali si possono adottare molte techinche, e noi adotteremo il testing, il quale non è mai buono abbastanza!

In generale, per trattare gli errori si utilizzano i seguenti approcci:

- **Error Prevention**: Avviene prima del rilascio del sistema. Consiste nell'applicare buone tecniche di programmazione, usare versioni di controllo e applicare verifiche per prevenire bug.

- **Error Detection**: Avviene mentre il sistema è in esecuzione: mediante Testing (creare failure in maniera pianificata), Debugging, Monitoring.
- **Error Recovery**: Recuperare da un failure una volta che il sistema è stato rilasciato.

Diamo ora delle definizioni di base per il Testing:

- **Errori**: Sono commessi da persone.
- **Fault**: Un fault è il risultato di un errore nella documentazione software, nel codice, etc.
- **Failure**: Un failure avviene quando si esegue un fault.
- **Incident (episodio)**: Conseguenza di failure (l'occorrenza di una failure può o meno essere visibile all'utente).
- **Testing**: Usare il software con casi di test allo scopo di trovare fault o di acquisire fiducia nel sistema.
- **Caso di test (Test Case)**: Insieme di risultati attesi (oracoli) e di input che vanno a usare un componente con lo scopo di causare failures.
- **Test Suite**: Insieme di casi di test.

5.1.1 Concetti di Testing

Una **componente** è una parte del sistema che può essere isolata per essere testata: può essere un gruppo di oggetti, un sottosistema o anche più sottosistemi. Eseguire test case su una componente o su una combinazione di componenti richiede che essi siano isolati dal resto del sistema. Quando questo non è possibile a causa per esempio di parti mancanti del sistema, per eseguire i test andiamo a usare i *Test Driver* e i *Test Stub*.

Un **Test Stub** è un'implementazione parziale di una componente che dipende dalla componente che andiamo a testare. In pratica simula le componenti chiamate dalla componente testata.

Un **Test Driver** invece è un'implementazione parziale di una componente che dipende dalla componente testata. In pratica va a simulare il chiamante della componente testata.

Questi due elementi ci permettono di eseguire test su un sistema ancora da implementare del tutto, accelerando i tempi di testing. Ovviamente Test Stub e Test Driver ben progettati richiedono un maggiore sforzo.

Una volta che i test sono stati eseguiti e le failure rilevate, gli sviluppatore cambiano la componente per eliminare il fallimento individuato, tramite una **correzione**. Chiaramente, una correzione potrebbe introdurre nuovi fault, ed è per questo che utilizziamo tecniche come il **Testing di Regressione**: esso è la riesecuzione di tutti i test effettuati precedentemente successiva a un cambiamento.

Alcune osservazioni È impossibile testare in maniera esaustiva ogni modulo di un sistema non banale, a causa di limitazioni teoriche (problemi di arresto per esempio) o pratiche (tempi/costi proibitivi). Secondo Dijkstra, un test può solo verificare la presenza di bug, non la loro assenza, quindi un test ha successo se riesce a causare una failure.

Tecniche di Gestione dei Fault

Il test delle componenti può essere effettuato mediante i seguenti:

- **Unit Testing (test di unità)**: Eseguito dagli sviluppatori su sottosistemi individuali, con l'obiettivo di confermare che il sottosistema in questione è codificato correttamente ed esegue le funzionalità intese.
- **Integration Testing**: Eseguito da sviluppatori su gruppi di sottosistemi (collezioni di classi) ed eventualmente sull'intero sistema, con l'obiettivo di testare le interfacce fra i sottosistemi.

Il test del sistema, invece, viene effettuato tramite i seguenti:

- **System Testing**: Eseguito dagli sviluppatori sull'intero sistema, per verificare se esso rispecchia i requisiti (funzionali e globali).
- **Acceptance Testing**: Valuta il sistema fornito dagli sviluppatori, viene eseguito infatti dai clienti. Ha l'obiettivo di verificare che il sistema rispecchia i requisiti del cliente e che è pronto all'uso.

5.1.2 Documentare il Testing

Per minimizzare le risorse necessarie, il Testing deve essere gestito e pianificato. Per fare ciò, vengono usati i seguenti documenti:

- **Test Plan**: Si focalizza sugli aspetti manageriali del testing. In particolare, documenta gli scopi, gli approcci, le risorse e lo scheduling delle attività di testing. In questo documento sono identificati i requisiti e le componenti che devono essere testati.
- **Test Case Specification**: Documenta ogni test. In particolare, contiene gli input, i driver, gli stub e gli output attesi dai test, così come i task che devono essere eseguiti.
- **Test Incident Report**: Documenta ogni esecuzione, e in particolare i risultati reali dei test e le differenze con quelli attesi.
- **Test Summary Report**: Elenca tutte le failure rilevate durante i test, che devono essere investigate. Da questo documento, gli sviluppatori analizzano e assegnano priorità a ogni failure e pianificano i cambiamenti da apportare al sistema e ai modelli. Questi cambiamenti possono introdurre nuovi test case e nuove esecuzioni di test.

5.2 Tecniche di Testing di Unità

Lo **Unit Testing** può essere eseguito in maniera informale, mediante la codifica incrementale, oppure mediante analisi statiche o dinamiche. L'*analisi statica* avviene mediante: esecuzione manuale, leggendo il codice sorgente; walkthrough (presentazione informale ad altri); ispezione del codice (presentazione formale ad altri); tool automatici per controllare errori sintattici e semantici, e divergenze dagli standard di codifica.

L'*analisi dinamica* invece avviene mediante: **black-box testing** (per testare i comportamenti di input/output), **white-box testing** (per testare la logica interna del sottosistema o dell'oggetto), **testing basato sulle strutture dati** (i tipi di dati determinano i casi di test).

In generale, il Testing di Unità nasce da tre motivazioni: Si riduce la complessità concentrandosi su una sola unit del sistema per volta, è più facile correggere i bug perché poche componenti sono coinvolte, ed è possibile testare più unità in parallelo. Le unità candidate per il test sono prese dal modello a oggetti e dalla decomposizione in sottosistemi (i sottosistemi devono essere testati dopo che ogni classe e oggetto al loro rispettivo interno è stato testato individualmente).

5.2.1 Black-Box Testing

Si focalizza sul comportamento di I/O, senza preoccuparsi della struttura interna della componente. Se per ogni dato input siamo in grado di prevedere correttamente l'output, allora il modulo supera il test. Tuttavia, è quasi sempre impossibile generare tutti i possibili input, cioè tutti i test case. L'obiettivo diventa, quindi, quello di ridurre il numero di casi di test effettuando un partizionamento: si dividono le condizioni di input in **classi di equivalenza** e si scelgono i test case per ogni classe di equivalenza.

Per selezionare le classi di equivalenza non ci sono regole, ma solo due linee guida:

- Se l'input è valido per un range di valori, si scelgono i test case da 3 classi di equivalenza: sotto il range, nel range e sopra il range.
- Se l'input è valido solo se appartiene a un insieme discreto, si scelgono i test case da 2 classi di equivalenza: valore discreto valido, valore discreto non valido.

Un'altra soluzione per scegliere solo un numero limitato di test case è giungere a conoscenza dei comportamenti interni delle unità da testare. (White-Box Testing).

5.2.2 Equivalence Class Testing

Nasce dall'esigenza di avere un testing completo ma che eviti ridondanza. Le classi di equivalenza sono partizioni dell'input set. Tutto l'inpuit set viene coperto (ottenendo la *completezza*) e le classi sono disgiunte (evitando la *ridondanza*). I test case sono elementi di ogni classe di equivalenza. La difficoltà sta nello scegliere saggiamente le classi di equivalenza, individuando i più probabili comportamenti sottostanti.

Parliamo ora di **weak** e **strong** equivalence class testing. Se prendiamo in considerazione una moltitudine di input che partecipano alla stessa operazione, come è realistico che sia, ci rendiamo conto che potrebbe non essere sufficiente andare a testare un valore per ogni input ma potremmo aver bisogno di prendere in considerazione le interazioni fra di essi.

Ebbene è proprio questa la differenza fra questi due tipi di testing. Il **Weak Equivalence Class Testing** sceglie un valore da ogni classe di equivalenza, mentre lo **Strong Equivalence Class Testing** si basa sul prodotto cartesiano degli insiemi di partizione e testa tutte le possibili interazioni fra le classi.

In generale, il Weak Equivalence Class Testing è appropriato quando i dati in input sono definiti in termini di range e insiemi di valori discreti.

5.2.3 Boundary Value Testing

Abbiamo partizionato i domini di input in classi di equivalenza partendo dall'assunzione che all'interno della stessa classe ci sia la stessa probabilità di generare errori. Tuttavia, alcuni errori tipici di programmazione avvengono ai confini fra classi differenti. Su questo concetto focalizza l'attenzione questo tipo di testing, più semplice ma complementare alle precedenti tecniche di testing viste.

se prendiamo per esempio in considerazione due variabili di input x_1 e x_2, sappiamo che esse sono comprese in degli intervalli. Ebbene, i valori che possono assumere rispettivamente sono il minimo, un po' più del minimo, un valore nominale, un po' meno del massimo e il massimo. Per convenzione questi valori si indicano con min, min+, nom, max- e max.

Il Test Set si ottiene fissando una variabile al suo valore nominale e facendo variare le altre. Quindi una finzione di n variabili richiede $4n + 1$ test cases. Questa tecnica è abbastanza rudimentale, non prende in considerazione la nautra delle variabili e inoltre è sensibile al robustness testing: infatti non andiamo a testare valori non validi.

5.2.4 Worst Case Testing

I valori limite (il testing precedente) partono dalla comune assunzione che le failure, la maggior parte delle volte, siano causate da un fault. Ma cosa succede se più di una variabile assume un valore estremo? Prendiamo in considerazione il prodotto cartesiano di min, min+, nom, max-, max. È chiaramente una tecnica più completa della precedente, ma molto più costosa: 5^n casi di test.

È una buona strategia se le variabili fisiche hanno numerose interazioni, e se le failure sono molto costose. Il **Robust** Worst Case Testing è ancora più efficace, in quanto prende in considerazione anche valori oltre il massimo e sotto il minimo, ovviamente a un costo ancora maggiore.

5.2.5 White-Box Testing

Questo tipo di testing si focalizza sulla completezza (copertura). Ogni statement nella componente è eseguito almeno una volta. Indipendentemente dall'input, ogni stato nel modello dinamico dell'oggetto e ogni interazione tra gli oggetti viene testata.

Esistono quattro tipi di white box testing:

- **Statement Testing (test algebrico)**: Si testano i singoli statement.
- **Loop Testing**: Provoca l'esecuzione del loop che deve essere saltato completamente. I loop possono essere eseguiti esattamente una volta o più di una volta.
- **Path Testing**: Assicura che i path del programma siano eseguiti.
- Branch Testing (testing condizionale): Assicura che ogni possibile uscita da una condizione sia testata almeno una volta.

5.2.6 Confronto fra testing White-Box e testing Black-Box

Il numero di cammini da testare è potenzialmente infinito o comunque non realistico da esplorare completamente. Il White-Box Testing va spesso a testare ciò che è stato fatto piuttosto che ciò che dovrebbe essere fatto. Non individua quindi i casi d'uso mancanti. Il Black-Box Testing invece, ha la peculiarità di essere una potenziale esplosione combinatoria di casi di test (dati validi e non), e siccome va a testare le specifiche delle componenti è più adatto a rivelare comportamenti mancanti.

È bene tenere a mente che comunque i due tipi di testing non sono mutualmente esclusivi e anzi, sono da considerarsi complementari.

5.3 Testing di Integrazione

Quando i bug in ogni componente sono stati rilevati e fixati, le componenti sono pronte per essere integrate in sottosistemi più grandi. Il **Test di Integrazione** rileva bug che non sono stati rilevati durante il test di unità, focalizzando l'attenzione su un insieme di componenti che vengono integrate. Due o più componenti vengono integrate e analizzate,e quando dei bug sono rilevati, possono essere aggiunte nuove componenti per correggerli.

Siamo quindi nella situazione in cui l'intero sottosistema è visto come una collezione di sottosistemi (insiemi di classi) determinati durante il system e l'object design. L'ordine in cui i sottosistemi vengono selezionati per il testing e per l'integrazione determina le strategie di testing che andremo ora a vedere.

5.3.1 Big Bang Integration

Le componenti vengono testate individualmente e poi insieme come un unico sistema. Sebbene sia molto semplice, è costoso: se un test scopre una failure, è impossibile stabilire

se è nell'interfaccia o all'interno di qualche componente.

5.3.2 Bottom Up Integration

I sottisistemi al livello più basso della gerarchia sono testati individualmente. I successivi sottosistemi ad essere testati sono quelli che chiamano i sottosistemi testati in precedenza. Si ripete ques'ultimo passo finché tutti i sottosistemi non sono stati testati. Per il testing si utilizzano dei Test Drivers, che simulano le componenti dei livelli più alti che non sono ancora state integrate.

Questo approccio non è buono per sistemi decomposti funzionalmente, poiché testa i sottosistemi più importanti solo alla fine, ma è utile per integrare sistemi object-oriented, real-time e con rigide richieste di performance.

5.3.3 Top Down Integration

Testa prima i livelli alti della gerarchia o i sottosistemi di controllo e, successivamente, combina tutti i sottosistemi che sono chiamati da quelli già testati e testa la collezione risultate di sottosistemi. Ripete il tutto fino a quando tutti i sottosistemi non sono stati incorporati nel test. Per il testing vengono utilizzati Test Stub, che simulano le componenti dei livelli più bassi che non sono ancora state integrate.

Il vantaggio di questo approccio è che i test case possono essere definiti in termini delle funzionalità del sistema. Tuttavia, scrivere gli stub può risultare difficile: devono consentire tutte le possibili condizioni da testare. Inoltre, è possibile che un grande numero di stub sia richiesto, soprattutto se il livello più in basso nel sistema contiene molti metodi.

5.3.4 Sandwich Testing

Combina l'uso di strategie top-down e bottom-up. Il sistema è visto come avente 3 strati: Un livello **target** nel mezzo, uno sopra il target e uno sotto.

Lo scopo del testing è quello di convergere al target. Se il nostro sistema ha più di 3 layer, potrebbe diventare non banale andare a identificare il target e i due livelli rispettivamente superiore e inferiore. Un modo potrebbe essere per esempio cercare di minimizzare il numero di stub e driver.

Il vantaggio principale nell'utilizzare questo approccio è che i test dei livelli in alto e in basso possono essere eseguiti in parallelo. Il problema è che non vengono testati sottosistemi individuali del livello target in modo completo, prima dell'integrazione. La soluzione può essere: *rullo di tamburi*

Strategia di Sandwich Testing Modificata Testa, in parallelo, il livello al top con stub per il target, il livello nel mezzo con driver e stub per i livelli rispettivamente superiore e inferiore, e il livello in basso con driver per il target. Inoltre, testa in parallelo il livello in alto che accede a quello nel mezzo e il livello in basso acceduto da quello nel mezzo.

5.4 System Testing

Unit Testing e Integration Testing focalizzano l'attenzione sulla ricerca di bug nelle componenti individuali e nelle interfacce tra le componenti. Il **System Testing** invece, assicura che il sistema completo sia conforme ai requisiti funzionali e non. Le attività per questo testing sono: Structure Testing, Functional Testing, Pilot Testing, Performance Testing, Acceptance Testing, Installation Testing.

L'impatto dei requisiti sul testing di sistema è pesante: più espliciti sono i requisiti, più facili sono da testare. La qualità degli use case determina la facilità del functional testing, mentre la qualità dei requisiti funzionali e non funzionali e dei vincoli determina la facilità del performance testing.

5.4.1 Structure Testing

Essenzialmente è la stessa cosa di un testing White-Box. L'obiettivo è quello di coprire tutti i cammini nel system design: usa tutti i parametri di input e output per ogni componente, usa tutte le componenti e tutti i chiamanti (ogni componente è chiamato almeno una volta e tutte le componenti sono chiamate da tutti i possibili chiamanti), usa testing delle condizioni e delle iterazioni come il testing di unità.

5.4.2 Funcional Testing

Essenzialmente lo stesso di un testing Black-Box. L'obiettivo è di testare le funzionalità del sistema, che viene trattato come una Black Box. I test case sono progettati a partire dal RAD e incentrati attorno ai requisiti e alle funzioni chiave (casi d'uso). Andiamo a istanziare casi d'uso per derivare casi di test (quindi praticamente creare scenari).

5.4.3 Performance Testing

Si vuole spingere il sistema già integrato ai suoi limiti, con l'obiettivo di metterlo sotto stress e cercare di "romperlo". Bisogna testare come si comporta il sistema quando va in sovraccarico: possono essere identificati eventuali colli di bottiglia? Si porvano flussi di istruzioni non usuali, si testano grandi moli di dati.

5.4.4 Acceptance Test

L'obiettivo è provare che il sistema è pronto per l'uso operativo. A tale scopo la scelta dei test è fatta dagli stakeholder, molti test possono essere presi da ltesting di integrazione e il test in sé è eseguito dal cliente e non dallo sviluppatore.

La maggioranza dei bug nel software è individuata dal cliente dopo che il sistema entra in uso, perciò vengono introdotti due test addizionali:

- **Alpha Test**: Gli stakeholder usano il software negli ambienti degli sviluppatori. Il software è usato con un settaggio controllato, con gli sviluppatori sempre pronti a correggere seduta stante i bug.

- **Beta Test**: Condotto negli ambienti degli sponsor, ma senza la presenza degli sviluppatori. Il software inizia a lavorare realisticamente nell'ambiente per il quale è stato progettato. Clienti potenziali potrebbero essere scoraggiati nel caso di scarse performance.

C'è da notare che questi due ultimi test negli ultimi anni hanno cambiato sensibilmente il loro significato in alcuni mercati, per esempio non è raro vedere un Alpha Test aperto al pubblico nell'ambiente videoludico.