

Elementi di Ingegneria del Software

Davide Quartucci

25 maggio 2023

Indice

1	Introduzione	2
2	Software Processes	6
3	Agile Principles	12
4	Agile development techniques - XP	14
5	AgileProjectManagement	17
6	Requirements Engineering	21
7	Software Architecture	23
8	UML	26
9	System Modeling	29
10	GRASP	32
11	GoF Patterns	35
12	Design to Code	39
13	Implementation issues	44
14	Software Testing	46
15	Software Evolution	49
16	Domande frequenti	50

Capitolo 1

Introduzione

1.1 Key Points

- Perchè Ingegneria del Software? Cosa si intende per Ingegneria del Software?
- L'etica dietro l'Ingegneria del Software
- Casi studio

1.2 Visione d'insieme

”L’ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione di software. Gli attributi essenziali dei prodotti software sono la **manutenibilità, l'affidabilità e la sicurezza, l'efficienza e l'accettabilità**. Le attività di alto livello di specifica, sviluppo, validazione ed evoluzione fanno parte di tutti i processi software. Esistono molti tipi diversi di sistemi e ognuno di essi richiede strumenti e tecniche di ingegneria del software adeguati per il loro sviluppo. Le idee fondamentali dell’ingegneria del software sono applicabili a tutti i tipi di sistemi software. Gli ingegneri del software hanno delle responsabilità nei confronti della professione ingegneristica e della società. Non devono limitarsi a occuparsi di questioni tecniche.”

1.3 Perchè Ingegneria del Software?

1. Le economie di TUTTI i Paesi sviluppati dipendono da software.
2. Sempre più sistemi sono controllati dal software
3. La spesa per il software rappresenta una frazione significativa del PNL in tutti i Paesi sviluppati.
4. I sistemi stanno diventando estremamente complessi
5. C’è richiesta di consegne rapide

La produzione di software non è un’attività artigianale: deve essere un processo industriale.

1.3.1 Tipi di software

- **Prodotti generici.**

1. Sistemi autonomi che vengono commercializzati e venduti a qualsiasi cliente che desideri acquistarli.
2. Esempi: software per PC, come programmi di grafica, strumenti di gestione dei progetti; software CAD; software per mercati specifici, come i sistemi di appuntamenti per i dentisti.
3. Le specifiche di ciò che il software deve fare sono di proprietà dello sviluppatore del software e le decisioni sulla modifica del software sono prese dallo stesso sviluppatore.

- **Prodotti personalizzati o su misura.**

1. Software commissionato da un cliente specifico per soddisfare le proprie esigenze.
2. Esempi: sistemi di controllo incorporati, software di controllo del traffico aereo, sistemi di monitoraggio del traffico. monitoraggio del traffico.
3. Il cliente possiede le specifiche del software e decide le modifiche da apportare al software.

1.3.2 Tipi di applicazione

- *Applicazioni stand-alone* - sono programmi software che possono essere eseguiti su un singolo computer locale senza la necessità di una connessione di rete. Questo tipo di applicazioni solitamente includono tutte le funzionalità necessarie e non richiedono componenti esterni per funzionare correttamente.
- *Applicazioni interattive basate su transazioni* - sono programmi che vengono eseguiti su un server remoto e ai quali gli utenti accedono da dispositivi remoti, come un PC o un terminale. Questo tipo di applicazioni spesso viene utilizzato per servizi online, come l'e-commerce, dove gli utenti possono eseguire transazioni attraverso un sito web.
- *Sistemi di controllo integrati* - sono software che gestiscono dispositivi hardware, come sensori, attuatori e sistemi di controllo. Questi sistemi sono spesso utilizzati in applicazioni industriali e automotive, dove sono necessari controlli precisi e sicuri. I sistemi embedded sono un tipo di sistema di controllo integrato che è programmato per eseguire funzioni specifiche su dispositivi hardware dedicati.

1.3.3 Attributi di un buon software

Lista:

- Attributi funzionali (prestazioni; cosa fa il sistema).
- Attributi non funzionali (qualità; come il sistema lo fa).

Product Characteristic	Description
Maintainability	Evolution qualities such as Testability, extensibility.
Dependability	Reliability, security, safety.
Efficiency	Response time, processing time, memory utilization.
Usability	Easy to <u>learn</u> how to use the system by target users. <u>Efficient to use</u> the system by users to accomplish a task. <u>Satisfying</u> to use by intended users.

1.3.4 Differenze nei nomi

Computer science: focus sulla teoria e i suoi fondamenti

Software engineering: si occupa della progettazione, dello sviluppo e della fornitura di software. Fa parte dell'ingegneria dei sistemi.

System engineering: tutti gli aspetti dello sviluppo di sistemi basati su computer: HW + SW + Processo

1.3.5 Processo produttivo di un software

- **Richieste per il software**, in cui clienti e ingegneri definiscono il software che deve essere prodotto e i vincoli sul suo funzionamento.
- **Progettazione e implementazione del software** (talvolta definita "sviluppo"), in cui il software viene progettato e programmato.
- **Convalida del software**, in cui il software viene controllato per garantire che sia quello richiesto dal cliente.
- **Evoluzione del software**, in cui il software viene modificato per riflettere l'evoluzione dei requisiti del cliente e del mercato.

1.3.6 I principi dell'ingegneria del software

I principi fondamentali dello sviluppo software che si applicano a tutti i tipi di sistemi software sono: l'utilizzo di un processo di sviluppo gestito, l'affidabilità e le prestazioni, la comprensione e la gestione dei requisiti del software, e il riutilizzo del software esistente. Essi sono importanti per garantire l'efficienza, l'affidabilità e il successo del processo di sviluppo software. Alcuni principi fondamentali si applicano a tutti i tipi di sistemi software, indipendentemente dalle tecniche di sviluppo utilizzate:

- I sistemi devono essere sviluppati utilizzando un processo di sviluppo gestito e compreso. Naturalmente, per i diversi tipi di software si utilizzano processi diversi.
- L'affidabilità e le prestazioni sono importanti per tutti i tipi di sistema.
- Comprendere e gestire le specifiche e i requisiti del software richiesti (ciò che il software deve fare).
- Se opportuno, si dovrebbe riutilizzare il software già sviluppato piuttosto che scriverne di nuovo.

1.3.7 Ingegneria del software per il web

il Web è la piattaforma sempre più utilizzata per lo sviluppo di sistemi software, grazie ai suoi servizi che ci consentono l'accesso alle funzionalità delle applicazioni tramite Internet. Il cloud computing è un approccio di fornitura di servizi informatici in cui le applicazioni vengono eseguite in remoto sul cloud. I principi fondamentali dell'ingegneria del software, discussi in precedenza, sono applicabili ai sistemi "Web-based", nonostante questi sistemi sono complessi.

1.3.8 I costi nell'ingegneria del software

Dipendono da:

- Il processo produttivo utilizzato
- Il tipo di software sviluppato.

Ogni approccio generico ha un profilo diverso di distribuzione dei costi.

- Circa il 60% dei costi è rappresentato dai costi di sviluppo, il 40% dai costi di test.
- Per custom software, i costi di evoluzione spesso superano quelli di sviluppo.

1.3.9 Riassumendo: Ingegneria del software

L'ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione del software, compreso il processo tecnico di sviluppo, la gestione del progetto e lo sviluppo di strumenti e metodi per supportare la produzione del software. Utilizza teorie e metodi appropriati per risolvere i problemi, tenendo conto dei vincoli organizzativi e finanziari.

1.4 Etica dell'ingegneria del software

L'ingegneria del software implica responsabilità più ampie della semplice applicazione di competenze tecniche. Gli ingegneri del software devono comportarsi in modo onesto ed eticamente responsabile per essere rispettati come professionisti. Il comportamento etico implica il rispetto di una serie di principi moralmente corretti che vanno oltre il semplice rispetto della legge.

1.4.1 Problemi comuni

- **Riservatezza**

- Gli ingegneri devono normalmente rispettare la riservatezza dei loro datori di lavoro o dei loro clienti, indipendentemente dal fatto che sia stato firmato o meno un accordo formale di riservatezza.

- **Competenza**

- Gli ingegneri non devono dichiarare erroneamente il proprio livello di competenza. Non devono non devono accettare consapevolmente lavori che non rientrano nelle loro competenze.

- **Diritti di proprietà intellettuale**

- Gli ingegneri devono essere a conoscenza delle leggi locali che regolano l'uso della proprietà intellettuale, come brevetti, copyright, ecc. Devono fare attenzione a garantire la protezione della proprietà intellettuale dei datori di lavoro e dei clienti.

- **Uso improprio del computer**

- Gli ingegneri del software non devono utilizzare le loro competenze tecniche per abusare dei computer altrui. L'uso improprio del computer va da quello relativamente banale (ad esempio, giocare sul computer di un datore di lavoro) a quello estremamente grave (diffusione di virus).

1.4.2 ACM/IEEE

Le società professionali degli Stati Uniti hanno elaborato un codice etico che i membri di queste organizzazioni sottoscrivono al momento dell'adesione. Il Codice contiene otto principi relativi al comportamento e alle decisioni degli ingegneri del software, compresi professionisti, educatori, manager, supervisori, responsabili delle politiche, tirocinanti e studenti della professione.

1. PUBBLICO - Gli ingegneri del software devono agire in modo coerente con l'interesse pubblico.
2. CLIENTE E DATORE DI LAVORO - Gli ingegneri del software devono agire nel migliore interesse del loro cliente e del loro datore di lavoro.
3. PRODOTTO - Gli ingegneri del software devono garantire che i loro prodotti e le relative modifiche soddisfino i più alti standard professionali possibili.
4. GIUDIZIO - Gli ingegneri del software devono mantenere l'integrità e l'indipendenza del loro giudizio professionale.
5. GESTIONE - I dirigenti e i responsabili dell'ingegneria del software devono sottoscrivere e promuovere un approccio etico alla gestione dello sviluppo e della manutenzione del software.
6. PROFESSIONE - Gli ingegneri del software devono promuovere l'integrità e la reputazione della professione.
7. COLLEGHI - Gli ingegneri del software devono essere corretti e solidali con i loro colleghi.
8. AUTOSUFFICIENZA - Gli ingegneri del software devono partecipare all'apprendimento continuo per quanto riguarda l'esercizio della loro professione e devono promuovere un approccio etico all'esercizio della professione.

1.5 Casi studio: lettura sulle slide

- Un microinfusore di insulina personale: un sistema integrato in una pompa di insulina utilizzata dai diabetici per mantenere il controllo della glicemia.
- Un sistema di gestione dei casi di salute mentale (Mentcare): un sistema utilizzato per mantenere i registri delle persone che ricevono assistenza per problemi di salute mentale.
- Una stazione meteorologica: un sistema di raccolta dati che raccoglie dati sulle condizioni meteorologiche in aree remote.
- SPID: un sistema di identità digitale - Un sistema per autenticare gli utenti su diversi servizi.
- iLearn: un ambiente di apprendimento digitale - Un sistema per supportare l'apprendimento nelle scuole.

Capitolo 2

Software Processes

2.1 Key Points

- Modelli per la progettazione software
- Attività della progettazione
- Affrontare il cambiamento
- Miglioramento del software

2.2 Visione d'insieme

La progettazione software è l'attività coinvolta nella produzione di un sistema software, mentre i modelli della progettazione software sono rappresentazioni astratte di questi processi. I modelli generali includono 2 tipologie di modello: "a cascata" e a "sviluppo incrementale".

	Plan-driven	Agile
linear	waterfall	—
iterative	UP, MSF, Sync&Stub,...	XP, SCRUM, ...

Figura 2.1: Modelli di progettazione

L'ingegneria dei "requisiti" è il processo fondamentale per lo sviluppo di un software specifico, in quanto consente di definire in modo preciso e completo cosa il software dovrà fare e come dovrà farlo. Successivamente, la progettazione e l'implementazione trasformano queste specifiche in un sistema eseguibile, mentre la convalida verifica che il sistema soddisfi le aspettative degli utenti e le specifiche. L'evoluzione del software è un processo continuo, necessario per mantenere il software utile nel tempo e per far fronte ai cambiamenti delle esigenze degli utenti. I prototipi e la consegna progressiva sono attività utilizzate per affrontare i cambiamenti in modo efficiente. Gli approcci al miglioramento dei software includono gli approcci "agile" e quelli "maturity-based"; il SEI definisce una scala di valutazione dei processi software basata sull'uso di buone pratiche di ingegneria del software.

2.3 Modelli per la progettazione software

La progettazione software è **un insieme di attività strutturate necessarie per sviluppare un sistema software**. Ci sono 4 attività principali coinvolte in ogni processo software: specifiche, progettazione, convalida ed evoluzione. La descrizione della progettazione software comprende solitamente le attività che devono essere svolte e il loro ordine, ma può anche includere prodotti, ruoli e pre/postcondizioni.

2.3.1 Processi "agile" e pianificati

I processi plan-driven sono quelli in cui tutte le attività sono pianificate in anticipo e i progressi sono misurati rispetto a questo piano, mentre nei processi "agile" la pianificazione è progressiva e il processo è più facilmente modificabile per rispondere alle mutevoli esigenze dei clienti. La maggior parte dei processi pratici combina elementi di entrambi gli approcci. Inoltre, non esiste un processo da seguire per il software giusto o sbagliato, ma deve essere scelto in base al contesto.

2.3.2 Modelli di progettazione software

- Il modello a cascata
 - Modello plan-driven. Fasi separate e distinte di specifica e sviluppo
- Sviluppo incrementale
 - Specifica, sviluppo e convalida si alternano. Può essere di tipo plan-driven o agile.
- Altri...

Entrambi i modelli possono includere il riutilizzo e l'assemblaggio di componenti esistenti. In pratica, la maggior parte dei sistemi di grandi dimensioni viene sviluppata utilizzando un processo che incorpora elementi di tutti questi modelli.

2.3.3 Le fasi del modello a cascata

Il modello a cascata prevede fasi distinte:

1. Analisi e definizione dei requisiti
2. Progettazione del sistema e del software
3. Implementazione e test delle unità
4. Integrazione e test di sistema
5. Funzionamento e manutenzione

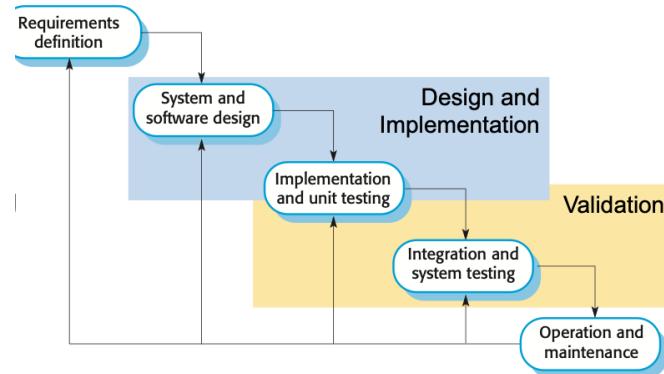


Figura 2.2: WaterfallModel

2.3.4 i problemi del modello a cascata

Lo svantaggio principale del modello a cascata è la difficoltà di accogliere i cambiamenti dopo che il processo è stato avviato. In linea di principio, una fase deve essere completata prima di passare alla fase successiva. La suddivisione inflessibile del progetto in fasi distinte rende difficile rispondere alle mutevoli esigenze dei clienti. Pertanto, questo modello è appropriato solo quando i requisiti sono ben compresi e le modifiche saranno piuttosto limitate durante il processo di progettazione. Pochi sistemi aziendali hanno requisiti stabili. Il modello a cascata è utilizzato soprattutto per progetti di ingegneria dei sistemi di grandi dimensioni, in cui un sistema viene sviluppato in diversi siti. In queste circostanze, la natura pianificata del modello a cascata aiuta a coordinare il lavoro.

2.3.5 Sviluppo incrementale

- **Outline:** una versione minimale dei requisiti e del design
- **Quick cycles** ognuno dei quali prevede
 - Specificazione di un delta
 - Sviluppo del delta
 - Convalida del delta

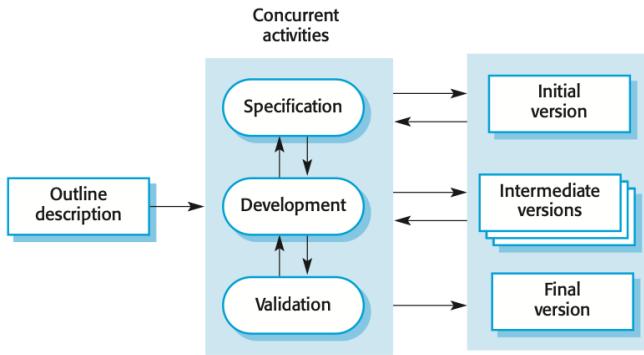


Figura 2.3: Sviluppo-Incrementale

2.3.6 Vantaggi dello sviluppo incrementale

- Si riducono i costi di adattamento ai cambiamenti dei requisiti dei clienti.
 - La quantità di analisi e documentazione da rifare è molto inferiore a quella richiesta dal modello a cascata.
- È più facile ottenere il feedback dei clienti sul lavoro di sviluppo svolto.
 - I clienti possono commentare le dimostrazioni del software e vedere quanto è stato implementato.
- È possibile consegnare e distribuire più rapidamente il software utile al cliente.
 - I clienti possono utilizzare e trarre valore dal software prima di quanto sia possibile con un processo a cascata.
 - MVP - Prodotto Minimo Vitale

2.3.7 I problemi dello sviluppo incrementale

- Il processo non è determinato in anticipo.
 - I manager devono pianificare tempi e costi all'inizio e misurare i progressi. misurare i progressi.
- La struttura del sistema tende a degradarsi con l'aggiunta di nuovi incrementi.
 - A meno che non si dedichino tempo e denaro al refactoring per migliorare il software, i cambiamenti regolari tendono a corrompere la sua struttura. Incorporare ulteriori modifiche al software diventa sempre più difficile e costoso.

2.3.8 Tipi di SW riutilizzabili

Il riutilizzo del software, specialmente attraverso l'utilizzo di componenti o sistemi applicativi esistenti, è diventato sempre più comune nel processo di sviluppo del software. Questo approccio può migliorare l'efficienza, ridurre i costi e accelerare la distribuzione del software. Inoltre, molte aziende utilizzano anche software open-source come componenti riutilizzabili nei loro sistemi. Ecco alcune tipologie:

- Sistemi applicativi stand-alone (talvolta chiamati COTS) configurati per l'uso in un particolare ambiente.
- Collezioni di oggetti sviluppati come pacchetti da integrare con un framework di componenti come .NET o J2EE.
- Servizi Web sviluppati in base a standard di servizio e disponibili per l'invocazione remota.
 - Questo è il modello di riutilizzo prevalente oggi, il software viene rilasciato in docker

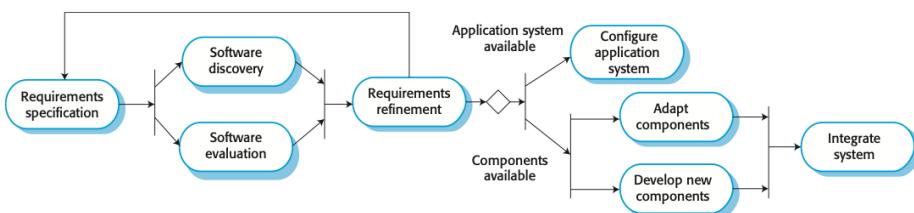


Figura 2.4: Reuse in SW engineering

2.3.9 Vantaggi e svantaggi dei SW riutilizzabili

- **Riduzione dei costi e dei rischi**, in quanto viene sviluppato meno software da zero
- Consegnata e distribuzione **più rapida** del sistema
- Ma i compromessi sui requisiti sono inevitabili, quindi il sistema **potrebbe non soddisfare le reali esigenze degli utenti**
- **Perdita di controllo** sull'evoluzione degli elementi del sistema riutilizzati

2.4 Attività della progettazione

Le attività della progettazione software sono sequenze interconnesse di attività tecniche, collaborative e manageriali con l’obiettivo generale di specificare, progettare, implementare e testare un sistema software. Le quattro attività di base della progettazione sono organizzate in modo diverso nei vari processi di sviluppo.

1. Processo di specifica

- Il processo di definizione dei servizi necessari e dei vincoli per il funzionamento e lo sviluppo del sistema.
- Requisiti:
 - Elicitazione e analisi dei requisiti: Che cosa richiedono o si aspettano gli stakeholder del sistema?
 - Specifica dei requisiti: Definizione dettagliata dei requisiti
 - Convalida dei requisiti: Verifica della validità dei requisiti

2. Sviluppo

- Il processo di conversione delle specifiche del sistema in un sistema eseguibile.
- Progettazione del software
 - Progettare una struttura software che realizzi la specifica;
 - La DOCUMENTAZIONE fa parte del software.
- Implementazione
 - Tradurre questa struttura in un programma eseguibile;
- Le attività di progettazione e implementazione sono strettamente correlate e possono essere interconnesse.

Le attività di **design** nel processo di sviluppo software includono l’identificazione della struttura complessiva del sistema, la progettazione delle strutture dei dati del sistema e della loro rappresentazione in un database, la definizione delle interfacce tra i componenti del sistema e la ricerca di componenti riutilizzabili e la progettazione del loro funzionamento se non sono disponibili.

Nell’attività di **implementazione**, il software viene sviluppato tramite la scrittura di uno o più programmi o la configurazione di moduli esistenti. Design e implementazione sono attività interconnesse. La programmazione è un’attività individuale e non ha un processo standard. Il debug è l’attività di individuazione e correzione dei difetti del programma.

3. Validazione

La validazione nel processo di sviluppo software ha lo scopo di dimostrare che un sistema è conforme alle sue specifiche e soddisfa i requisiti del cliente. Ciò include processi di verifica e revisione, nonché il collaudo del sistema. Il testing del sistema prevede l’esecuzione del sistema con casi di test derivati dalle specifiche dei dati reali che il sistema deve elaborare. L’attività di collaudo è la più comunemente utilizzata, ma ci sono anche altre attività come l’ispezione del codice e il profiling del codice. Ecco le fasi di testing:

- (a) Test dei componenti: I singoli componenti vengono testati in modo indipendente; I componenti possono essere funzioni o oggetti o raggruppamenti coerenti di queste entità.
- (b) Test del sistema nel suo complesso: Il test delle proprietà emergenti è particolarmente importante.
- (c) Test del cliente: Test con i dati dei clienti per verificare che il sistema soddisfi le loro esigenze.

4. Evoluzione

Il software è flessibile e può cambiare per adattarsi a requisiti in continua evoluzione. Questi cambiamenti possono essere causati da mutazioni delle circostanze aziendali o da altri fattori. La demarcazione tra lo sviluppo del software e la sua evoluzione o manutenzione sta diventando sempre più irrilevante poiché sempre meno sistemi sono completamente nuovi e richiedono un intervento di sviluppo completo.

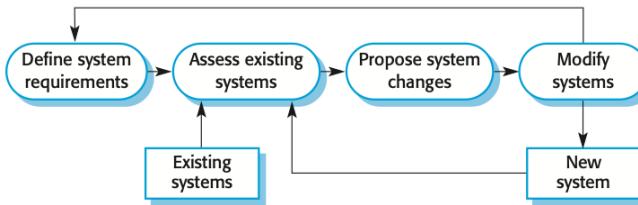


Figura 2.5: Software-Evolution

2.5 Affrontare il cambiamento

Ci sono 3 concetti fondamentali per saper affrontare al meglio il cambiamento:

1. Il Cambiamento in sé:

Il cambiamento è inevitabile nei grandi progetti software, portando alla rielaborazione e ai costi di implementazione di nuove funzionalità. L'anticipazione del cambiamento può essere fatta attraverso attività di sviluppo di prototipi del sistema per mostrare ai clienti alcune caratteristiche chiave del sistema. La tolleranza al cambiamento può essere raggiunta attraverso uno sviluppo incrementale del sistema, in cui le modifiche proposte possono essere implementate in incrementi che non sono ancora stati sviluppati, o solo su una parte del sistema.

2. Il Prototipo del progetto:

Un prototipo è una versione iniziale di un sistema utilizzata per dimostrare i concetti e provare le opzioni di progettazione. Può essere utilizzato per contribuire all'elicitazione e alla convalida dei requisiti nei processi di ingegneria, per esplorare le opzioni e sviluppare il design dell'interfaccia utente nei processi di progettazione e per eseguire test back-to-back nel processo di collaudo. I prototipi possono essere basati su linguaggi rapidi o strumenti e possono comportare l'eliminazione di funzionalità per concentrarsi su aree del prodotto che non sono ben comprese. Tuttavia, i prototipi sono normalmente non documentati e devono essere scartati dopo lo sviluppo perché non sono una buona base per un sistema di produzione. Inoltre, la struttura del prototipo è solitamente degradata a causa dei rapidi cambiamenti e il prototipo probabilmente non soddisfa i normali standard di qualità dell'organizzazione. Ecco i benefici nell'avere un prototipo:

- (a) Miglioramento nell'usabilità del sistema.
- (b) Maggiore rispondenza alle reali esigenze degli utenti.
- (c) Miglioramento della qualità del design.
- (d) Miglioramento della manutenibilità.
- (e) Riduzione dello sforzo di sviluppo.

3. La Consegna Graduale:

Il modello di sviluppo incrementale prevede che lo sviluppo e la consegna del sistema vengano suddivisi in progressi, con ogni incremento che fornisce una parte della funzionalità richiesta. I requisiti degli utenti vengono classificati in base alle priorità e i requisiti con più alta priorità vengono inclusi nei primi progressi. Una volta avviato lo sviluppo di un incremento, i requisiti vengono congelati, anche se i requisiti per i progressi successivi possono continuare a evolversi. Questo approccio viene spesso utilizzato nei metodi "agile". Ogni progresso viene valutato prima di procedere allo sviluppo dell'incremento successivo. I progressi vengono consegnati gradualmente per l'utilizzo da parte degli utenti finali, il che consente una valutazione più realistica nell'uso pratico del software. Tuttavia, questo modello può essere difficile da implementare per i sistemi sostitutivi, poiché i progressi hanno meno funzionalità del sistema da sostituire.

- Vantaggi di una consegna graduale:
 - Le richieste del cliente possono essere fornite con ogni incremento, in modo che le funzionalità desiderate siano disponibili prima.
 - I primi incrementi fungono da prototipo per aiutare a definire i requisiti per gli incrementi successivi.
 - Riduzione del rischio di fallimento complessivo del progetto.
 - I servizi di sistema a più alta priorità tendono a ricevere il maggior numero di test.
- Svantaggi di una consegna graduale:

- La maggior parte dei sistemi richiede un insieme di strutture di base che vengono utilizzate da diverse parti del sistema. Poiché i requisiti non vengono definiti in dettaglio fino a quando non viene implementato un incremento, può essere difficile identificare le strutture comuni necessarie a tutti gli incrementi.
- L'essenza dei processi iterativi è che la specifica viene sviluppata insieme al software. Tuttavia, questo è in conflitto con il modello di molte organizzazioni, in cui il sistema completo di specifiche è parte del contratto di sviluppo del sistema.

2.6 Miglioramento della progettazione

Molte aziende adottano l'approccio degli "update dei software" per migliorare la qualità del loro software, ridurre i costi o accelerare i processi di sviluppo. Ciò significa comprendere i processi esistenti e modificarli per aumentare la qualità del prodotto e/o ridurre i costi e i tempi di sviluppo. L'update si concentra sul miglioramento della gestione della progettazione e sull'introduzione di buone pratiche di ingegneria del software. L'approccio "agile" si concentra sullo sviluppo iterativo e sulla riduzione delle spese generali. Le caratteristiche principali dei metodi agili sono la rapidità di consegna delle funzionalità e la reattività alle mutevoli esigenze dei clienti.

Attività da svolgere per migliorare il processo:

1. Misurazione: Si misurano uno o più attributi del processo o del prodotto software. Queste misurazioni formano una linea di base che aiuta a decidere se i miglioramenti del processo sono stati efficaci.
2. Analisi: Si valuta il processo attuale e si identificano i punti deboli e i colli di bottiglia del processo. Si possono sviluppare modelli di processo (talvolta chiamati mappe di processo) che descrivono il processo.
3. Modifica: Vengono proposte modifiche al processo per risolvere alcuni dei punti deboli identificati. Queste vengono introdotte e il ciclo riprende per raccogliere dati sull'efficacia delle modifiche.

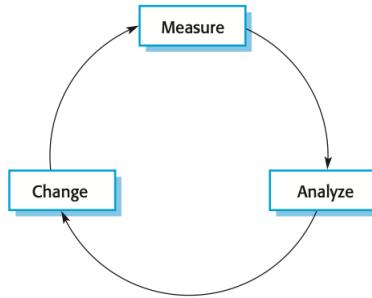


Figura 2.6: ImprovementCycle

Capitolo 3

Agile Principles

3.1 Key Points

- Principi "agile" e applicazioni
- Metodologie "agile"

3.2 Visione d'insieme

L'approccio Agile si basa su una serie di principi, tra cui la scomposizione del prodotto in una serie di blocchi gestibili e comprensibili (I requisiti instabili non ne impediscono il progresso). Consideriamo le tecniche di sviluppo Agile (XP) e la gestione Agile del progetto (SCRUM)

3.3 Principi Agile e applicazioni

La rapidità di sviluppo e consegna è oggi spesso il requisito più importante per i sistemi software. Le aziende operano in un contesto in rapida evoluzione ed è praticamente impossibile produrre una serie di requisiti software stabili. Il software deve evolversi rapidamente per riflettere le mutevoli esigenze aziendali. Il processo di sviluppo pianificato è essenziale per alcuni tipi di sistemi, ma non soddisfa queste esigenze aziendali. I metodi Agile sono emersi per ridurre radicalmente i tempi di consegna dei sistemi software funzionanti.

3.3.1 Obiettivo dell'Agile

Negli anni '80 e '90 l'elevato costo dei metodi di progettazione del software ha portato alla creazione dei metodi Agile, che si basano sull'approccio allo sviluppo del software, con l'obiettivo di fornire rapidamente un software funzionante e di evolverlo rapidamente per soddisfare i requisiti in evoluzione. I metodi Agile riducono le spese generali nel processo del software e permettono di rispondere rapidamente ai cambiamenti dei requisiti senza eccessive rielaborazioni. Il sistema viene sviluppato come una serie di versioni o incrementi, con le parti interessate coinvolte nella specifica e nella valutazione delle versioni, con attenzione al codice funzionante e con team indipendenti e coordinamento leggero. L'approccio Agile viene utilizzato per lo sviluppo di prodotti software di piccole o medie dimensioni destinati alla vendita, nonché per lo sviluppo di sistemi personalizzati all'interno di un'organizzazione, dove c'è un chiaro impegno da parte del cliente a partecipare al processo di sviluppo e dove ci sono poche regole e normative esterne che influenzano il software.

3.3.2 Agile manifesto

1. La nostra massima priorità è soddisfare il cliente attraverso la consegna tempestiva e continua di software di valore.
2. Accogliere i requisiti che cambiano, anche in ritardo nello sviluppo, i processi Agile sfruttano il cambiamento a vantaggio del cliente.
3. Le aziende e gli sviluppatori devono collaborare quotidianamente per tutta la durata del progetto.
4. Consegnare software funzionante frequentemente, da un paio di settimane a un paio di mesi, con una preferenza per i tempi più brevi.

5. Costruire progetti attorno a persone motivate.
6. Date loro l'ambiente e il supporto di cui hanno bisogno, e fidatevi di loro per portare a termine il lavoro.
7. Il metodo più efficiente ed efficace per trasmettere informazioni a un team di sviluppo è la conversazione faccia a faccia.
8. Il software di lavoro è la principale misura del progresso.
9. I processi agili promuovono lo sviluppo sostenibile. Lo sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere un ritmo costante a tempo indeterminato.
10. L'attenzione continua all'eccellenza tecnica e al buon design aumenta l'agilità.
11. La semplicità, ovvero l'arte di massimizzare la quantità di lavoro non svolto, è essenziale.
12. Le migliori architetture, i requisiti e la progettazione emergono da team auto-organizzati.
13. A intervalli regolari, il team riflette su come diventare più efficace, quindi mette a punto e regola il proprio comportamento di conseguenza.

3.4 Metodologie Agile

Esistono diverse metodologie, le più famose sono:

- Extreme Programming - XP
- Dynamic System Development Method - DSDM
- SCRUM
- Feature Driven Development - FDD
- Crystal
- Agile modeling
- Lean Software Development
- ...

3.4.1 XP

Extreme Programming (XP) è un metodo Agile sviluppato alla fine degli anni '90 che adotta un approccio estremo allo sviluppo iterativo. Tra le sue tecniche di sviluppo ci sono la realizzazione di nuove versioni del software più volte al giorno e l'esecuzione di tutti i test per ogni build, accettando la build solo se i test sono stati eseguiti correttamente. Sebbene lo sviluppo Agile utilizzi le pratiche di XP, il metodo originariamente definito non è molto utilizzato.

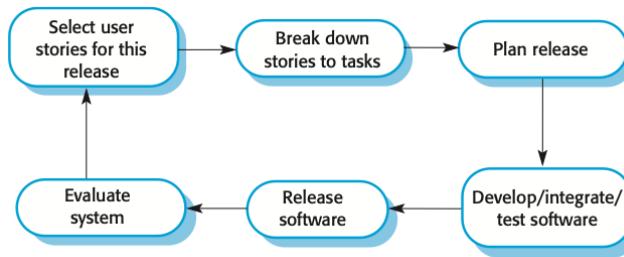


Figura 3.1: XP

Il metodo di sviluppo Agile Extreme Programming (XP) prevede una pianificazione incrementale dei requisiti, con suddivisione in storie e compiti, e una produzione di rilasci frequenti e incrementalni. La progettazione è semplice e si concentra sui requisiti attuali, mentre lo sviluppo segue il test-first, il refactoring e la programmazione a coppie. Gli sviluppatori lavorano su tutte le aree del sistema in modo da evitare isole di competenza e si adotta l'integrazione continua. Il ritmo sostenibile è importante, così come la presenza a tempo pieno di un rappresentante del cliente nel team di sviluppo.

Capitolo 4

Agile development techniques - XP

4.1 Key Points

- User Stories
- Refactoring
- Test-first
- Programmazione a coppie

4.2 Visione d'insieme

Tra le pratiche di tipo Agile abbiamo: Decisioni lato utente, rifattorizzazione, sviluppo basato su test e programmazione a coppie.

4.3 Users Stories

In XP, un cliente o un utente fa parte del team XP ed è responsabile delle decisioni sui requisiti di progettazione.

- I requisiti dell'utente sono espressi come storie o scenari.
- Questi vengono scritti su schede e il team di sviluppo li suddivide in attività di implementazione. Il cliente sceglie le storie da includere nella release successiva in base alle sue priorità e alle stime dei tempi.(Non devono essere troppo lunghe)

Una storia utente è finita quando è soddisfacente. La struttura è: **As a, I want to, so that**

Ecco alcuni esempi:

testUS1_01(): input(name), selectfromDD(name), output(record), check(name, ...)

User stories: 'As a physician, i want to access a patient record so that i can manage prescription' 'As a physician i want to access an existing prescription so that i can validate it' 'As a physician i want to update a patient record so that i can insert a new prescription' 'As a physician i want to access a formulary so that i can select a prescription for a patient' 'As a physician i want the system to validate my prescription so that it is safe'.

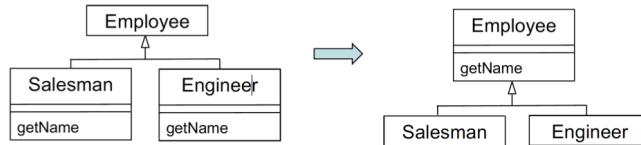
Acceptance criteria: Given (that i have access the system) When (I select a patient) Then (i see patient a record)

4.4 Refactoring

Convenzionalmente nell'ingegneria del software l'obiettivo è quello di prevedere il cambiamento. Vale la pena spendere tempo e sforzi per anticipare i cambiamenti, in quanto ciò riduce i costi nelle fasi successive del ciclo di vita del SW. Il team di programmazione cerca i possibili miglioramenti del software e li apporta anche quando non sono necessari nell'immediato. Questo migliora la comprensibilità del software e riduce la necessità di documentazione, inoltre le modifiche sono più facili da realizzare perché il codice è ben strutturato e chiaro(anche per uno sviluppatore esterno); Tuttavia, alcune modifiche richiedono il refactoring dell'architettura e questo è molto più costoso e complesso rispetto ad un classico refactoring.

4.4.1 Esempio di refactoring

L'obiettivo è riorganizzare la gerarchia delle classi per eliminare il codice duplicato, quindi inizialmente parto riordinando e rinominando attributi e metodi per renderli più comprensibili e successivamente sostituisco il codice inline con chiamate a metodi inclusi in una libreria.



4.5 Test-first

XP adotta un approccio "test-first" per lo sviluppo del software, in cui i test sono scritti prima del codice e vengono eseguiti automaticamente ad ogni modifica apportata (questo garantisce testing graduale e completo). Questo approccio coinvolge l'utente nella convalida dei test e assicura che i requisiti siano ben definiti. I test sono scritti come programmi e vengono eseguiti automaticamente utilizzando un framework di test come JUnit. La scrittura di test automatizzati aiuta a individuare e correggere errori in modo tempestivo e garantisce che il software sia funzionante e affidabile. ATTENZIONE! i test e gli acceptance criteria sono due cose diverse, infatti i test sono scriptati come programmi eseguibili, devono coprire situazioni con diversi input e diverse condizioni, coprono anche casi negativi, cercano specificamente gli errori.

4.5.1 Il ruolo del cliente

Il ruolo del cliente nel processo di test è quello di contribuire allo sviluppo di test di accettazione per le storie che devono essere implementate nella prossima versione del sistema. Il cliente, che fa parte del team, scrive i test man mano che lo sviluppo procede. Tutto il nuovo codice viene quindi convalidato per garantire che sia quello di cui il cliente ha bisogno. Tuttavia, le persone che adottano il ruolo di cliente hanno un tempo limitato a disposizione e quindi non possono lavorare a tempo pieno con il team di sviluppo. Potrebbero ritenere che fornire i requisiti sia stato un contributo sufficiente e quindi potrebbero essere riluttanti a farsi coinvolgere nel processo di test.

4.5.2 Automatizzazione dei test

L'automazione dei test è fondamentale in XP perché consente di eseguire rapidamente e in modo affidabile un insieme completo di test ogni volta che viene apportata una modifica al sistema. Ciò riduce i rischi associati all'introduzione di errori e consente di individuare tempestivamente eventuali problemi. Inoltre, l'automazione dei test consente di scrivere test più accurati e completi rispetto ai test manuali, riducendo il rischio di errori umani.

4.5.3 Problemi con i test

È importante che i programmatori comprendano l'importanza dei test e si impegnino a scriverli correttamente. Se i test non sono completi o non coprono tutti i possibili scenari, il sistema potrebbe non funzionare correttamente. È anche importante riconoscere che alcuni test possono essere difficili da scrivere, ma questo non significa che debbano essere ignorati. In questi casi, è necessario lavorare sodo per trovare un modo per testare la funzionalità in modo adeguato. Inoltre, è importante adottare un approccio a lungo termine e continuare a rivedere e aggiornare l'insieme di test per garantire che rimanga completo e aggiornato.

4.6 Programmazione a coppie

La programmazione a coppie è una pratica Agile che consiste nel lavorare in coppia per sviluppare il codice. Ci sono diversi vantaggi nell'utilizzare questa metodologia:

- Diffusione delle conoscenze:** lavorando insieme, i membri del team condividono le loro conoscenze e competenze, riducendo così il rischio di dipendenza da singoli individui e migliorando la proprietà comune del codice.
- Revisione del codice:** ogni riga di codice viene esaminata da almeno due persone, aumentando la qualità complessiva del codice e riducendo il numero di bug e errori.

3. **Refactoring:** lavorando insieme, i membri del team sono in grado di identificare più facilmente le aree del codice che richiedono miglioramento e di apportare modifiche in modo più efficiente.
4. **Maggiore efficienza:** nonostante la collaborazione richieda più tempo rispetto al lavoro individuale, ci sono prove che suggeriscono che la programmazione a coppie può essere più efficiente di due programmati che lavorano separatamente.

La differenza principale sta nel fatto che nel caso di task "nuovi" la coppia performa meglio del singolo; invece nel caso di task ripetitivi il singolo riesce a performare in modo più efficiente soprattutto con l'avanzare del tempo. Infine la programmazione a coppie può anche essere utilizzata come strumento di formazione per i membri del team meno esperti, poiché possono lavorare a stretto contatto con membri più esperti per apprendere nuove tecniche e metodi di sviluppo del software.

Capitolo 5

Agile Project Management

5.1 Key Points

- Gestione del progetto Agile
- Metodo scalare Agile
- Problemi Agile

5.2 Visione d'insieme

I clienti devono avere le consegne progressive e in tempo, così ottengono un feedback sul funzionamento del prodotto. Le spese generali del progetto sono ridotte e la documentazione è al minimo. SCRUM è incentrato su una serie di sprint, che sono fixati periodicamente quando viene sviluppato un update. Molti metodi di sviluppo pratici sono un mix di sviluppo agile e plan-based; È difficile scalare i metodi agile per sistemi di grandi dimensioni perché richiedono una progettazione a monte e documentazione, che possono essere in conflitto con l'approccio informale dei metodi agile.

5.3 Gestione del progetto Agile

La responsabilità principale dei Software-project-manager è quella di gestire il progetto in modo che il software venga consegnato in tempo e nel rispetto del budget previsto. Tuttavia, l'approccio standard plan-driven può essere integrato con l'approccio agile per ottenere migliori risultati. Nel metodo agile, i project manager adottano uno stile di gestione più flessibile e adattabile che si concentra sullo sviluppo incrementale del software e sull'utilizzo di pratiche agile. Ciò significa che invece di sviluppare l'intero software in una sola volta, il progetto viene suddiviso in fasi più piccole e il team lavora su ciascuna di queste fasi in modo progressivo. In questo modo, il team può adattarsi rapidamente ai cambiamenti del progetto, alle esigenze del cliente e rispondere con maggiore flessibilità alle richieste del mercato. La gestione agile dei progetti richiede quindi un'attitudine diversa da quella tradizionale, che si concentra sulla collaborazione, l'interazione costante con il cliente e la flessibilità.

SCRUM:



5.3.1 Ruoli

Il **Product Owner** è il responsabile dell'identificazione delle caratteristiche e dei requisiti del prodotto, stabilendo le priorità per lo sviluppo e rivedendo continuamente il backlog del prodotto; inoltre può essere un cliente, ma anche un product manager di un'azienda o un altro rappresentante delle parti interessate (E' lui che decide se accettare o no i risultati). Il **team di sviluppo** dovrebbe essere composto da circa 7 persone auto-organizzate, responsabili

dello sviluppo del software e di altri documenti essenziali del progetto.(più persone ci sono più è complessa la struttura organizzativa). Lo **Scrum Master** è un facilitatore che organizza le riunioni quotidiane, tiene traccia dell'arretrato di lavoro e comunica con il team e con gli stakeholder esterni. Il ruolo dello Scrum Master è simile a quello di un Project Manager, ma è incentrato sul supporto al team nella sua auto-organizzazione e sull'eliminazione degli ostacoli che impediscono il suo successo.

5.3.2 SCRUM events

Mon	Tue	Wen	Tue	Fri
	9:00 Daily 15'	9:00 Daily 15'	9:00 Daily 15'	9:00 Daily 15'
9:00 Sprint Planning 2h				
		Backlog grooming		
				Sprint review 1h
				Sprint retrospective 30-45'

La settimana inizia con lo **Sprint-planning**: della durata tra le 1-4 settimane, durante la quale avviene la progettazione degli incrementi del prodotto. Il tutto deve essere costruito, testato, documentato e confezionato. Durante lo Sprint non ci sono modifiche a ciò che è stato già pianificato e la durata deve essere tale da tenere le modifiche al di fuori dello Sprint; inoltre viene creato un breve obiettivo (1-16 ore) su cui il team si concentrerà durante questo periodo, soprattutto sugli elementi del backlog da completare, il tutto senza l'intervento dello Scrum Master. A questo punto abbiamo giornalmente il **Daily scrum** che dura in media 15 minuti, si svolgono in piedi, e non servono per risolvere problemi (sono aperte a chiunque voglia partecipare). Solo i membri del team, Scrum Master e il Product Owner possono parlare durante la riunione. Le riunioni aiutano ad evitare altri tipi di riunioni non necessarie e l'intero team partecipa per condividere informazioni sui progressi compiuti nel giorno precedente, i problemi riscontrati e ciò che è previsto per il giorno successivo. In questo modo, tutti i membri del team sono aggiornati sullo stato del progetto e possono ripianificare il lavoro a breve termine se necessario. Successivamente c'è la **Sprint review** durante la quale il team e gli stakeholder sono invitati a ispezionare gli update del prodotto realizzato durante lo sprint. La revisione coinvolge tutto il team e può essere aperta a chiunque voglia partecipare. Durante la review, il team presenta ciò che ha realizzato durante lo sprint sotto forma di demo. La revisione è informale, senza slide e richiede circa due ore di preparazione. Tutto il team partecipa per condividere il proprio lavoro con gli stakeholder e ricevere feedback. Infine abbiamo la **Sprint retrospective** che avviene alla fine di ogni sprint e coinvolge solo i membri del team. Durante questo momento, il team si ferma per 15-30 minuti per valutare ciò che ha funzionato e ciò che non ha funzionato durante lo sprint. L'obiettivo è migliorare continuamente il processo di sviluppo del prodotto e identificare le aree di miglioramento per il prossimo sprint.

5.3.3 Articoli

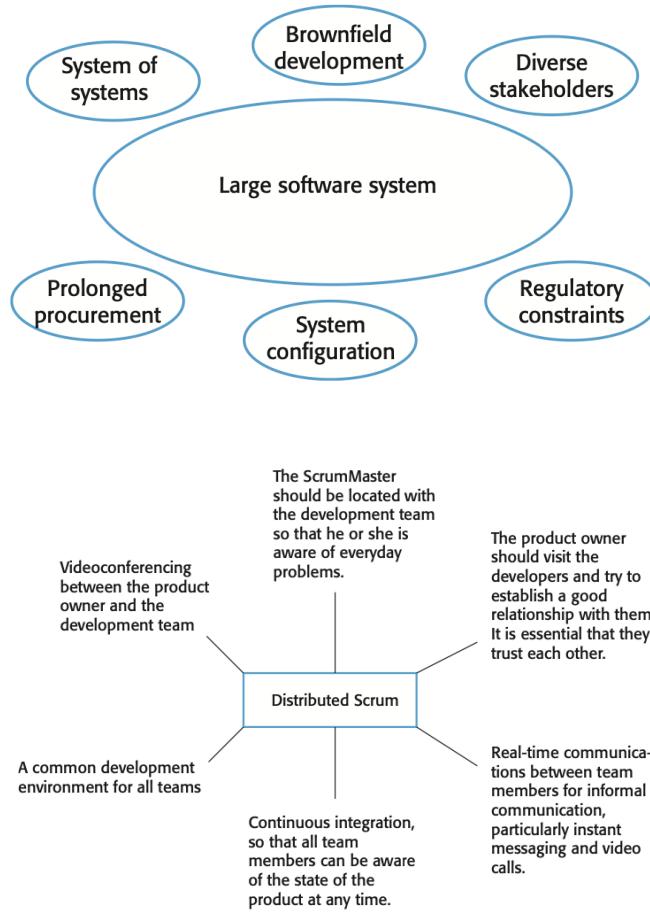
- **Product Backlog:** è un elenco priorizzato di elementi da costruire che il team Scrum deve affrontare. Questi elementi possono riguardare il software, i requisiti del software, le stories e altre descrizioni dei compiti supplementari necessari, come la definizione dell'architettura o la documentazione per l'utente.
- **Sprint Backlog:** è l'insieme di tutti i Product Backlog Items (PBI) selezionati per lo Sprint, che può essere suddiviso in task e stimato dai membri del team di sviluppo. Questo rende visibile il lavoro necessario per raggiungere l'obiettivo dello Sprint. Il lavoro non viene mai assegnato e la stima del lavoro rimanente viene aggiornata quotidianamente. Ogni membro del team può aggiungere, cancellare o modificare il backlog dello sprint e, se il lavoro non è chiaro, può essere definito un elemento del backlog con una quantità di tempo maggiore e ri-suddiviso in seguito. Il lavoro rimanente viene aggiornato costantemente per riflettere ciò che si è appreso.
- **Product increment:** è il risultato finale dello sprint, ovvero l'insieme di tutti i PBI che sono stati completati e quindi pronti per la produzione. Deve essere ispezionabile e potenzialmente consegnabile, il che significa che è finito e non ha bisogno di ulteriori lavori per essere incluso nel prodotto finale. Solo il Product Owner può decidere se rilasciarlo o meno.

5.3.4 Workflow

Scrum è un metodo Agile che si concentra sulla gestione dello sviluppo iterativo, prevedendo tre fasi: una fase iniziale di pianificazione in cui si stabiliscono gli obiettivi del progetto e si progetta l'architettura del software, seguita da una serie di cicli di sprint in cui si sviluppa un incremento del sistema. Infine, la fase di chiusura: in cui il progetto deve essere completo di documentazione necessaria.

5.4 Metodo scalare Agile

I metodi Agile sono efficaci per progetti di piccole e medie dimensioni con un piccolo team co-locato, ma per scalare questi metodi per progetti più grandi e lunghi con più team di sviluppo, sono necessarie modifiche. I team di grandi dimensioni sono di solito collezioni di sistemi separati che interagiscono e comunicano spesso in sedi diverse. I sistemi di grandi dimensioni sono anche vincolati da norme e regolamenti esterni, hanno tempi di sviluppo e approvvigionamento lunghi, e hanno un insieme eterogeneo di stakeholder. Lo "scaling up" implica l'utilizzo di metodi Agile per lo sviluppo di sistemi software di grandi dimensioni, mantenendo i fondamenti dell'Agile come pianificazione flessibile, rilasci frequenti, integrazione continua, sviluppo guidato dai test e buone comunicazioni tra i team. Tuttavia, l'approccio completamente incrementale all'ingegneria del software è impossibile e non può esistere un unico product owner o rappresentante del cliente. Inoltre, per lo sviluppo di grandi sistemi, non è possibile concentrarsi solo sul codice del sistema e sono necessari meccanismi di comunicazione intersettoriale. Anche se l'integrazione continua è praticamente impossibile, è essenziale mantenere frequenti build di sistema e rilasci regolari del sistema. Le date dei rilasci sono allineate e ci sono riunioni giornaliere dello Scrum of Scrums per

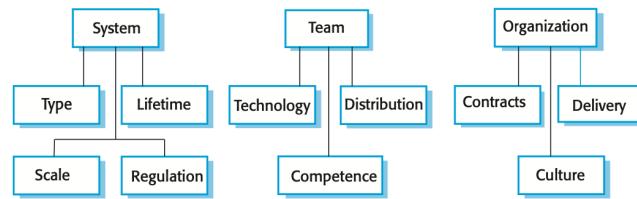


discutere i progressi. La "scalabilità" di uno Scrum dipende da fattori come il tipo di applicazione, la dimensione e la dispersione del team, la durata del progetto. Scrum è stato utilizzato con successo in progetti con oltre 500 persone coinvolte.

5.5 Problemi Agile

I project manager che non hanno esperienza di metodi Agile potrebbero essere riluttanti ad accettare un nuovo approccio. Le grandi organizzazioni con procedure burocratiche potrebbero essere incompatibili con i metodi Agile;

inoltre i metodi agile possono essere meno adatti per la manutenzione dei sistemi software esistenti, che costituisce la maggior parte dei costi del software nelle grandi aziende. I metodi agile devono supportare la manutenzione e lo sviluppo, e sorgono problemi se il team di sviluppo originale non può essere mantenuto. La mancanza inoltre di documentazione, coinvolgimento dei clienti e continuità del team di sviluppo sono problemi principali che potrebbero sorgere.



In breve, il tipo di sistema da sviluppare e la scala del progetto determinano se sia meglio utilizzare un approccio agile o plan-driven. Se il sistema è sufficientemente piccolo e il team è co-locato, allora l'approccio agile potrebbe funzionare bene. Al contrario, se il sistema richiede un team distribuito più grande, l'approccio potrebbe essere una combinazione di agile e plan-driven. Inoltre, la durata del sistema e le normative esterne possono influenzare la necessità di una documentazione dettagliata. Altri fattori da considerare includono le competenze del team di sviluppo, l'organizzazione del team, le tecnologie di supporto e la disponibilità dei rappresentanti dei clienti per fornire feedback. Infine, è importante valutare se l'approccio agile si inserisce nella cultura organizzativa della documentazione dettagliata.

Capitolo 6

Requirements Engineering

6.1 Key Points

- Requisiti funzionali e non funzionali
- Processi dell'ingegneria dei requisiti
- Esempio

6.2 Visione d'insieme

I requisiti di un sistema software definiscono le funzionalità e i vincoli del sistema. I requisiti funzionali specificano i servizi che il sistema deve fornire o le operazioni che deve eseguire, mentre i requisiti non funzionali vincolano il processo di sviluppo e spesso si riferiscono alle proprietà emergenti del sistema. L'ingegneria dei requisiti è un processo iterativo che comprende l'elicitazione delle informazioni dal cliente, la specificazione formale dei requisiti e la verifica della loro validità, coerenza, completezza, realismo e verificabilità. La gestione dei requisiti implica il controllo delle modifiche durante il processo di sviluppo. Il processo di definizione dei servizi e dei vincoli del sistema che un cliente richiede è chiamato ingegneria dei requisiti, e i requisiti di sistema sono le descrizioni dei servizi e dei vincoli generati durante questo processo. I requisiti possono essere usati come base per un'offerta di contratto o come base del contratto stesso. I requisiti dell'utente sono scritti in linguaggio naturale e diagrammi e sono rivolti ai clienti, mentre i requisiti di sistema sono descrizioni dettagliate delle funzioni, dei servizi e dei vincoli del sistema e possono far parte di un contratto tra cliente e appaltatore. I tipi di stakeholder includono utenti finali, gestori del sistema, proprietari del sistema e parti interessate esterne come pazienti, medici, infermieri, addetti all'accoglienza, personale informatico e responsabile dell'etica medica. I metodi agili spesso non utilizzano requisiti di sistema dettagliati, ma possono utilizzare un'ingegneria dei requisiti incrementale e le "storie dell'utente", il che può essere pratico per i sistemi aziendali, ma problematico per i sistemi critici o sviluppati da diversi team.

6.3 Requisiti funzionali e non funzionali

I **requisiti funzionali** sono la descrizione sulle funzionalità o servizi che il sistema deve fornire, mentre i **requisiti non funzionali** sono i vincoli come quelli sul tempo, sviluppo e costi. I requisiti funzionali possono essere di 2 tipi: user requirements, system requirements. I requisiti non funzionali invece possono riguardare l'architettura complessiva di un sistema piuttosto che i singoli componenti(nb: i requisiti n.funzionali posso generare requisiti f. lavorando sull'architettura).

6.4 Processi dell'ingegneria dei requisiti

Il processo di RE (Requirement Engineering) varia a seconda del dominio applicativo, delle persone e dell'organizzazione coinvolta, ma ci sono attività comuni come l'elicitazione, l'analisi, la convalida e la gestione dei requisiti. In ambiente agile, la RE è un'attività iterativa in cui queste attività si intrecciano per generare system requirements/user requirements.

6.4.1 Requirements elicitation

In sintesi, la gestione dei requisiti può essere complessa poiché gli stakeholder potrebbero non sapere con certezza cosa vogliono, potrebbero avere requisiti contrastanti e i requisiti potrebbero cambiare nel tempo a causa di nuove parti interessate o di cambiamenti nel contesto aziendale. Inoltre, la comunicazione tra gli specialisti delle applicazioni e l'ingegnere dei requisiti potrebbe essere difficile a causa di un linguaggio tecnico complicato/discordante; il tutto viene discusso attraverso brainstorming/interviews. Ecco alcuni esempi di requisiti:

- Una biblioteca gestisce prestiti di 100.000 volumi a 5.000 iscritti. La biblioteca dotata di un sistema di catalogazione dei libri. I volumi sono catalogati con i metadati bibliografici usuali (autore, titolo, editore, anno, ecc.) e identificati mediante il proprio ISBN ed un contatore di copia.
- Ci sono due tipi d'utente: il bibliotecario e l'iscritto; il primo può aggiornare la base di dati, mentre il secondo può solo consultare i dati dei libri. A tutti gli utenti sarà fornita un'interfaccia web. Un iscritto chiede alla biblioteca il prestito di uno o più volumi alla; la biblioteca invia al cliente la lista dei volumi disponibili.
- I libri sono prestati agli iscritti della biblioteca e gli iscritti sono identificati sia da un codice numerico, che dal cognome, nome e data di nascita. Il bibliotecario accede mediante password alle operazioni d'aggiornamento, mentre l'iscritto accede liberamente alle operazioni di consultazione e con SPID alla richiesta prestito.

6.4.2 Requirements specification

Il processo di scrittura dei requisiti comprende la stesura di requisiti utente e system in un documento di requisiti ben strutturato. I requisiti utente devono essere comprensibili per gli utenti finali, mentre quelli di sistema sono più dettagliati e tecnici. I requisiti possono far parte di un contratto e devono essere completi, precisi e non confusi. Il documento dei requisiti software è la dichiarazione ufficiale di ciò che è richiesto agli sviluppatori del sistema e dovrebbe definire COSA il sistema deve fare piuttosto che COME lo deve fare. Ci sono standard specifici per la documentazione dei requisiti come: IEEE Std 830-1998, VOLERE e Enterprise-specific standards.

6.4.3 Requirements validation

Il processo di validazione dei requisiti deve assicurare che il sistema fornisca le funzioni richieste dal cliente in modo coerente, completo, realistico e verificabile. La revisione dei requisiti può avvenire tramite analisi manuale e sviluppo di test per verificare la funzionalità. I requisiti devono essere verificabili, comprensibili, tracciabili e adattabili. Ciò significa che devono essere realisticamente testabili e compresi correttamente.

6.4.4 Requirements change

Il processo di gestione dei requisiti è fondamentale per gestire le modifiche del sistema nel tempo, dovute ai cambiamenti ambientali o ai nuovi requisiti degli utenti. Ci possono essere nuovi requisiti emergenti durante lo sviluppo del sistema o dopo il suo utilizzo, ed è quindi necessario tenere traccia dei singoli requisiti e mantenere i collegamenti tra quelli dipendenti. Per decidere se una modifica dei requisiti deve essere accettata, è necessario effettuare un'analisi del problema e specificare il cambiamento. Successivamente, si valuta l'impatto della modifica sui requisiti del sistema e si procede alla sua implementazione, modificando il documento dei requisiti e, se necessario, il progetto e l'implementazione del sistema.

6.5 Esempio

Esempio di user-stories per un'app di Cripto manager:

- As a user, I want to create an account, so that I can login
- As a user, I want to have a tutorial, so that I can understand
- As a user, I want to interact with a consultant
- As an external cryptoAsset manager, I want to propose my coin
- As a user, I want to automatically manage

Capitolo 7

Software Architecture

7.1 Key Points

- Cosa si intende per architettura software?
- Architectural views
- Architectural patterns

7.2 Visione d'insieme

L'architettura del software è la descrizione di come un sistema software è organizzato. Le decisioni di progettazione includono decisioni sul tipo di applicazione, sulla distribuzione del sistema e sugli stili architettonici da utilizzare. Le architetture possono essere documentate da diverse prospettive o punti di vista, logical view, process view e development view. Architectural patterns sono un mezzo per riutilizzare le conoscenze sulle architetture generiche dei sistemi. Descrivono l'architettura, spiegano quando può essere utilizzata e ne descrivono i vantaggi e gli svantaggi.

7.3 Cosa si intende per architettura software?

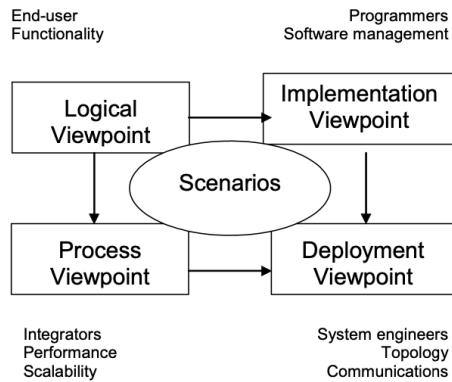
Architectural design si occupa di comprendere come dovrebbe essere organizzato un sistema software e di progettare la struttura complessiva di quel sistema, inoltre rappresenta il collegamento fondamentale tra "l'ingegneria dei requisiti" e la "detailed design", in quanto identifica i principali componenti strutturali di un sistema e le relazioni tra di essi. Il risultato del processo di progettazione architettonica è un modello architettonico spesso rappresentata attraverso una struttura a blocchi (es: UML). L'architettura del software può essere divisa in due categorie: l'architettura in scala ridotta che si concentra sulla struttura dei singoli programmi, e l'architettura in scala maggiore, che si occupa dei sistemi aziendali complessi distribuiti su diversi computer. La progettazione architettonica è importante in quanto consente di identificare i componenti strutturali principali di un sistema e le relazioni tra di essi. La ristrutturazione dell'architettura di un sistema è costosa, ma l'architettura può essere utilizzata come strumento di discussione e di analisi del sistema, di riutilizzo su larga scala e di documentazione. I diagrammi a blocchi sono il metodo più comune per documentare le architetture del software, e una visione ad alto livello del sistema è utile per la comunicazione con gli stakeholder e la pianificazione del progetto. L'obiettivo finale della progettazione architettonica è di produrre un modello completo del sistema che mostri i suoi componenti e le loro connessioni.

7.4 Architectural views

È necessario presentare diversi punti di vista della stessa architettura:

1. **logical viewpoint** = supporta i requisiti funzionali ovvero i servizi che il sistema dovrebbe fornire ai suoi utenti finali.
2. **process viewpoint** = risolve aspetti della fase di esecuzione (attività, thread e processi). Tiene conto di alcuni REQUISITI NON FUNZIONALI (come le performance)
3. **implementation viewpoint** = definisce l'ORGANIZZAZIONE dei moduli implementativi. Il software è suddiviso in piccoli pacchetti: librerie di programmi o sottosistemi che possono essere sviluppati da uno o più sviluppatori.

4. **deployment viewpoint** = definisce come gli altri punti di vista devono essere integrati e considera i requisiti non funzionali del sistema come la disponibilità del sistema, l'affidabilità (tolleranza ai guasti), le prestazioni (throughput) e la scalabilità.
5. **scenario viewpoint** = piccolo sottoinsieme di scenari importanti che fa vedere che gli altri 4 punti di vista lavorano insieme



Inoltre tenere conto della differenza tra: Architettura altoLivello (linguaggio informale) e architettura di bassoLivello (linguaggio formale).

7.5 Architectural patterns

Sono dei modelli descrittivi di alcuni aspetti, esiste una libreria di patterns(modelli) disponibile, da cui scegliere quale usare:

- Layered-pattern: questo pattern organizza il software in una serie di strati o livelli, ognuno dei quali rappresenta un'astrazione di livello superiore rispetto a quello sottostante. In genere, i livelli inferiori gestiscono aspetti tecnici e di basso livello, mentre i livelli superiori forniscono funzionalità di alto livello e più orientate al business.
- Master-slave: questo pattern prevede una suddivisione di ruoli tra i componenti di un sistema distribuito. Un componente principale, il "master", coordina le attività degli altri componenti, i "slave", che eseguono compiti specifici. Il pattern è particolarmente utile per la gestione di sistemi complessi, in cui è necessario coordinare le attività di molte parti.
- Client-server: questo pattern prevede la suddivisione delle funzionalità di un sistema in due parti principali: il "client", che è responsabile dell'interazione con l'utente finale, e il "server", che fornisce le funzionalità di back-end. Il pattern è particolarmente utile per sistemi distribuiti in cui è necessario gestire grandi volumi di dati o fornire servizi a un gran numero di utenti.
- Broker pattern: questo pattern prevede l'uso di un intermediario che gestisce la comunicazione tra i vari componenti, permettendo loro di scambiarsi messaggi in modo efficiente e trasparente.
- Peer-to-peer pattern: questo pattern prevede una rete di nodi o "peer" che collaborano tra di loro per raggiungere un obiettivo comune, senza la presenza di un server centrale. Il pattern è particolarmente utile per sistemi distribuiti in cui è necessario ridurre la dipendenza da un singolo punto di fallimento.
- Event-bus pattern: questo pattern prevede l'uso di un bus che funziona come un intermediario tra i vari componenti, inviando messaggi a tutti i componenti interessati quando si verifica un evento specifico. Il pattern è particolarmente utile per gestire situazioni in cui molti componenti devono essere notificati di un evento, senza che ci sia bisogno di stabilire una connessione diretta tra ciascun componente.
- Model-View-Controller (MVC) pattern: questo pattern prevede una suddivisione dei compiti in un sistema software in tre parti principali: il modello, la vista e il controller. Il modello rappresenta i dati e le logiche di business del sistema, la vista gestisce l'interfaccia utente, mentre il controller coordina le interazioni tra il modello e la vista. Il pattern è particolarmente utile per separare le responsabilità e migliorare la manutenibilità del codice.

- Blackboard pattern: questo pattern prevede l'uso di una lavagna condivisa come meccanismo di coordinamento tra i componenti di un sistema consentendo di coordinare le attività in modo collaborativo. Il pattern è particolarmente utile per sistemi in cui la soluzione a un problema richiede la collaborazione di molte parti, come ad esempio nel campo dell'intelligenza artificiale.

NB:

SOTTOSISTEMA = insieme di FUNZIONALITA'/RESPONSABILITA'

UML = rappresenta l'interfaccia di un progetto sw.

Capitolo 8

UML

8.1 Key Points

- Use case diagram
- Class model diagram
- Activity diagram
- Sequence diagram
- State diagram

8.2 Visione d'insieme

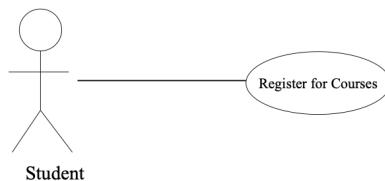
UML è un linguaggio di specifica standardizzato per la modellizzazione degli oggetti. Ci sono diversi diagrammi UML:

- Diagramma dei casi d'uso
- Diagramma delle classi
- Diagramma delle attività
- Diagramma di sequenza
- Diagramma degli stati

Ci sono diversi strumenti UML disponibili.

8.3 Use case diagram

Erano nati fuori dall'UML a supporto del requirements elicitation. Ci sono 2 elementi da tenere in considerazione: usecase e actor. Ogni use case è disegnato tramite un'ellisse e rappresenta una task effettuata da un attore(user o sistema), il quale è rappresentato da uno stickman. Un insieme di use case rappresenta il sistema. Esistono 4 tipi



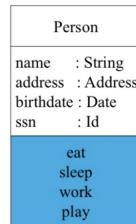
di collegamento:

1. associazione: comunicazione tra attore e use case [linea]
2. generalizzazione: relazione tra un use case generale ed uno speciale. [freccia che punta verso il padre]

3. inclusione: relazione di inclusione tra 2 use case quando hanno un comportamento simile. [Freccia tratteggiata -include- dallo use case più vago a quello più specifico(incluso)]
4. estensione: un use case aggiunge un comportamento al caso base. [Freccia tratteggiata -extend- verso il caso base]

8.4 Class model diagram

Le funzioni mappate nell'UML vengono raggruppate in delle classi. Una **classe** è **rappresentazione** di un **oggetto**. E' composta da un *nome* = tag, *attributi* = Proprietà che descrivono l'oggetto, *operazioni* = descrivono il comportamento della classe



8.4.1 UML relationships(tra classi)

In UML le connessioni tra gli oggetti(classi) sono modellate come relazioni e ce ne sono di 3 tipi: generalizzazioni, associazioni e dipendenze.

- La Generalizzazione = connessione tra sottoclasse e superclasse. Nel caso della programmazione ad oggetti la generalizzazione è implementata con il meccanismo dell'inheritance(ereditarietà)
- Associazione = 2 classi comunicano tra loro, inoltre quest'associazione può avere un nome oppure un comportamento di uno dei 2 oggetti, direzione ecc... Infine esistono 2 tipi di relazione di associazione:
 - Aggregazione = 1 oggetto che ha vita a sè stante è contenuto in un altro oggetto più grande
 - Composizione = 1 oggetto che ha vita solo dipendentemente da un altro oggetto più grande è contenuto in esso
- Dipendenza = relazione di tipo semantico che non implica associazione

8.4.2 Interfaccia

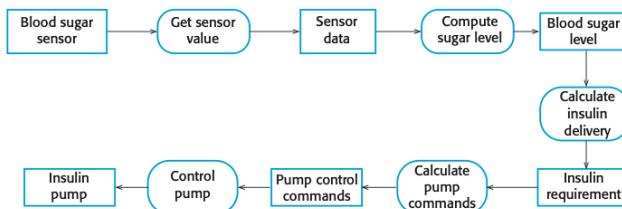
L'interfaccia è un insieme di operazioni con un nome, che specificano solo il comportamento dell'oggetto senza far vedere la struttura interna

8.4.3 Package

Permette di raggruppare più classi, può contenere al suo interno anche altri packages

8.5 Activity diagram

Il diagramma di attività è un flowchart che rappresenta il flusso di attività = processi e sottoprocessi (non codice) ed il loro ordine.



8.6 Sequence diagram

Sequence diagram è un diagramma che enfatizza l'ordine temporale di un oggetto. Viene rappresentato attraverso un rettangolo bianco = rappresenta il tempo di vita di un oggetto. Inoltre abbiamo i messaggi che sono rappresentati da frecce orizzontali e infine l'asterisco che rappresenta la ripetizione del messaggio.

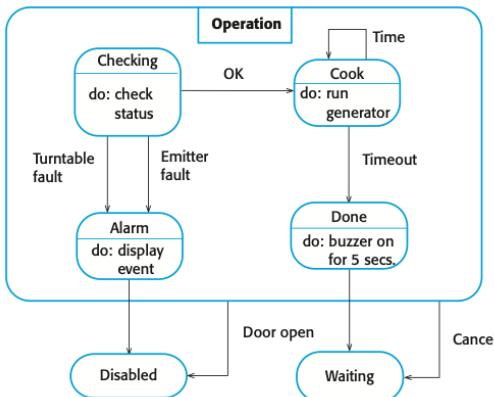
Costrutti addizionali opzionali:

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write <i>loop(n)</i> to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as <i>loop(i, 1, 10)</i>
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

8.7 State diagram

I modelli a macchina a stati mostrano gli stati del sistema come nodi ed gli eventi come archi tra questi nodi. Quando si verifica un evento, il sistema passa da uno stato ad un altro. Si focalizza sullo stato del sistema. Lo stato è la risposta del sistema ad uno stimolo, solitamente al diagramma è collegata una tabella degli stati ed una tabella degli stimoli.(con la descrizione di ogni stato e di ogni stimolo)

Esempio: microonde



Capitolo 9

System Modeling

9.1 Key Points

- Da Use Cases a Oggetto del dominio
- Da Use Cases al comportamento esterno
- Dal comportamento esterno al comportamento: cross-subsystem
- Dal comportamento esterno a quello interno
- Dal comportamento interno al class model

9.2 Visione d'insieme

La modellazione del sistema riguarda il passaggio dalle specifiche d'uso (storie degli utenti) al codice. Passaggi intermedi:

- Identificazione del modello di dominio
- Identificazione dei sottosistemi (architettura)
- Identificazione delle interazioni tra sottosistemi
- Costruzione della struttura interna dei sottosistemi
- Classi di confine
- Classi di entità
- Classi di controllo
- Ogni classe ha: Variabili, Metodi

Fondamentalmente dalle user stories che contengono specifiche si produce un 'domain model' = rappresentazione grafica del nostro modello; inoltre ogni user story rappresenta un external-behaviour. Unendo un external-behaviour con l'architettura del sistema si ottiene un sistema più complesso e completo, detto cross subsystem-behaviour. Successivamente voglio definire l'internal behaviour, per farlo integro il cross subsystem-behaviour con il domain model, poi si realizza il design class dyagram e solo infine si passa al codice(per individuare gli oggetti si utilizzano delle tecniche chiamate design patterns)

9.3 Da Use Cases a Oggetto del dominio

Il modello di dominio è una rappresentazione schematica del modello(Situazione reale, non software). Per prima cosa si identificano i noun phrases(sostantivi), poi si decide quali sono gli oggetti e attributi e infine si inseriscono i relativi attributi nel dominio degli oggetti, identificando anche la loro relazione.

Esempio(parole in corsivo = sostantivi):

- Step 1

1. Il *cliente* arriva alla cassa *POS* in un negozio con gli *articoli* da acquistare.
2. Il *cassiere* apre una nuova *vendita*.
3. Il *cassiere* inserisce l'*identificatore* dell'articolo.
4. Il sistema registra la *linea dell'ordine* del prodotto e presenta la *descrizione dell'articolo, il prezzo e il totale parziale*.
5. Il cliente effettua il *pagamento*.

- Step 2

```

@startuml
object sale {
    date
    total
}
object customer
object cashier
object item
object register
object item
object store
object payment
object productCatalog
object productDescription {
    price
    brand
    textDescription
}
sale *--> item
item *--> register
store *--> register : has
register *--> cashier
customer *--> sale
register *--> sale
sale *--> payment
item *--> productDescription
store *"--> "1..1 productCatalog: has
@enduml

```

L'importante è capire se un noun phrase è oggetto di dominio(complesso) oppure attributo(semplice).

ATTENZIONE: domain model è diverso dal data model, il primo include attori, dati transitori e non deve essere troppo dettagliato, invece il secondo mostra solo dati persistenti e esclude le classi senza attributi

9.4 Da Use Cases al comportamento esterno

Il comportamento esterno(behaviour model) è un modello dinamico di un sistema durante l'esecuzione. Mostra ciò che accade o ciò che dovrebbe accadere quando un sistema risponde a uno stimolo dall'ambiente circostante. Di stimoli ne esistono 2 tipi:

- DATA-DRIVEN MODELING: Si analizza la risposta del sistema se vengono inseriti dei dati in input
- EVENT-DRIVEN MODELING: Si analizza la risposta del sistema ad un evento

Una buona rappresentazione del behaviour model è attraverso i sequence diagrams. Un diagramma solitamente corrisponde ad un'use case

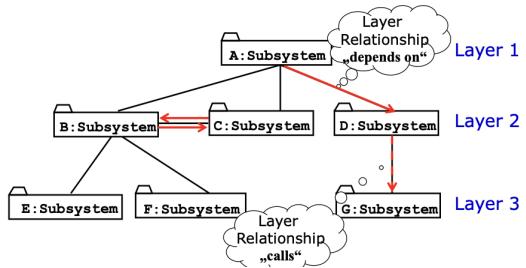
9.5 Dal comportamento esterno al comportamento: cross-subsystem

Ora vogliamo tradurre il nostro behaviour in cross-subsystem behavioiu model attraverso l'architettura. Il c-s behaviour model è caratterizzato da 2 elementi: Il sottosistema, cioè una collezione di classi, associazioni, eventi interconnessi tra loro rappresentati a livello grafico da un package e successivamente un servizio(service) cioè un insieme di operazioni con nome. Inoltre è importante distinguere tra: subsystem interface cioè insieme di operazioni in UML che specificano l'integrazione e il flusso di informazioni da e verso il sottosistema, e API con implementazione in un linguaggio specifico.

- I sottosistemi sono caratterizzati dal subsystem interface object, un oggetto di facciata che gestisce tutti i servizi(utile solamente dal punto di vista grafico). Esempio: "Advertisement" può essere s-i object di qualche sottosistema.

- Le relazioni tra sottosistemi possono essere di 3 tipi:

1. Layer A "depends on" layer B = è una dipendenza a compile time (linea solida).
2. layer A "calls" layer B = è una dipendenza a run time. Si rappresenta con una linea tratteggiata.
3. relazione di partizione: i sottosistemi hanno mutua conoscenza l'uno dell'altro.



9.6 Dal comportamento esterno a quello interno

Abbiamo individuato il dominio e i system-events a livello di sottosistemi, ora dobbiamo identificare gli oggetti nel modello, per questo useremo i design patterns.

I problemi più difficili nello sviluppo di sistemi orientati agli oggetti sono:

- Identificare gli oggetti
- Scomporre il sistema in oggetti

L'analisi dei requisiti si concentra sul dominio dell'applicazione a differenza dell'architettura che affronta sia il dominio dell'applicazione che quello di implementazione (Oggetti di soluzione addizionali)

9.6.1 Design Patterns

Un design pattern descrive un problema che si presenta nel nostro ambiente e successivamente descrive una soluzione a tale problema, in modo tale che possa essere utilizzata milioni di volte senza mai farla nel medesimo modo; ne esistono di 2 tipi: Grasp patterns e GoF patterns.

ATTENZIONE: Per design patterns non si intende:

- Strutture Dati che possono essere codificate in classi e riutilizzate per come sono(linked lists, hash tables...)
- Complex domain-specific designs(per un intera applicazione e subsystem)

9.6.2 Il processo

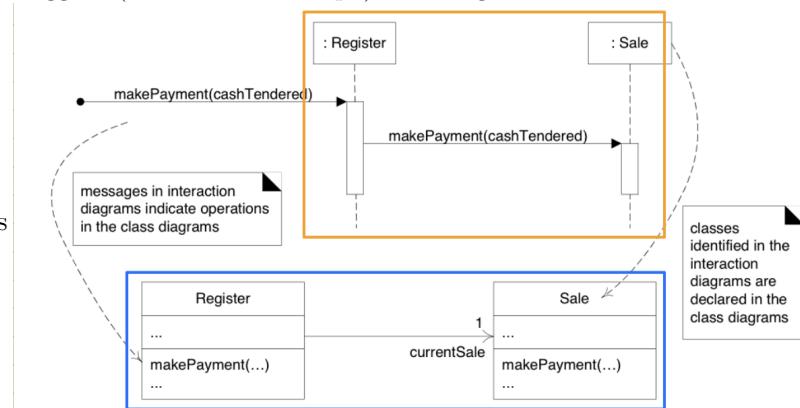
Ci sono tre tipi di classi:

- classi boundary = sono le astrazioni delle interfacce
- classi di controllo = sono gli oggetti software indipendenti dall'applicazione di dominio che sono di solito persistenti
- classi di entità = sono quelli che gestiscono i casi d'uso descritti nel pattern Controller

9.7 Dal comportamento interno al class model

Dopo aver individuato gli oggetti e le loro relazioni i sequence dyagram objects si mappano in classi di dominio(classi senza metodi e corpo) convertite poi in classi oggetto(con metodo e corpo) del design class model, cioè codice.

- In arancione: classi del Domain Model
- In blu: classi oggetto del Design Class Model



Capitolo 10

GRASP

10.1 Key Points

- Creator
- Information Expert
- Controller
- Low Coupling
- High Cohesion
- Pure Fabrication

10.2 Visione d'insieme

Ora analizzeremo un esempio di design pattern, iniziando con il Grasp Pattern(**G**eneral, **R**esponsability, **A**signment, **S**oftware, **P**atterns) e i suoi principi cardine che ci permetteranno di implementare il Responsability Driven Design (RDD), ovvero stabilire i criteri con cui si assegnano le responsabilità agli oggetti e come interagiscono tra loro.

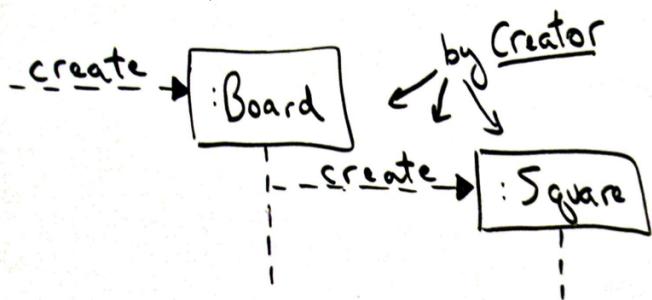
10.3 Creator

Problema: Chi crea A?

Soluzione: assegnare alla classe B la responsabilità di creare un'istanza di A se una o più sono vere:

1) B contiene o aggredisce A. 2) B registra A. 3) B utilizza da vicino A. 4) B ha i dati di inizializzazione per A

ATTENZIONE! L'associazione può essere molto complessa, in questi casi è necessario creare una nuova classe utilizzando un altro tipo di pattern: **Factory**



10.4 Information Expert

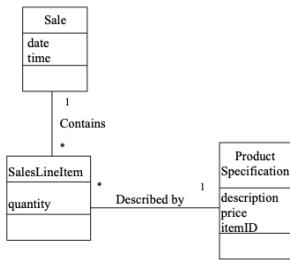
Problema: come assegnare responsabilità agli oggetti? Quale classe ne ha le responsabilità?

Soluzione: assegnare la responsabilità alla classe che ha le informazioni necessarie per soddisfarla

Esempio: sistema di vendita tramite un e-commerce

Who should be responsible for knowing the grand total of a sale?

- (We know we need a grand total from Use Cases / requirements and interaction diagrams for this scenario)



We don't have any design class yet, so **look into Domain Model**

(note: no responsibilities yet)

Non ha senso creare una classe a parte solo per un metodo, assegno tale responsabilità ad una classe che cerco nel Domain Model. Una volta trovata tale classe, si importa nel Design Class Model aggiungendole il metodo che assolve la responsabilità che ho appena valutato; talvolta è necessario dividere una responsabilità tra classi implementando metodi diversi per ognuna.

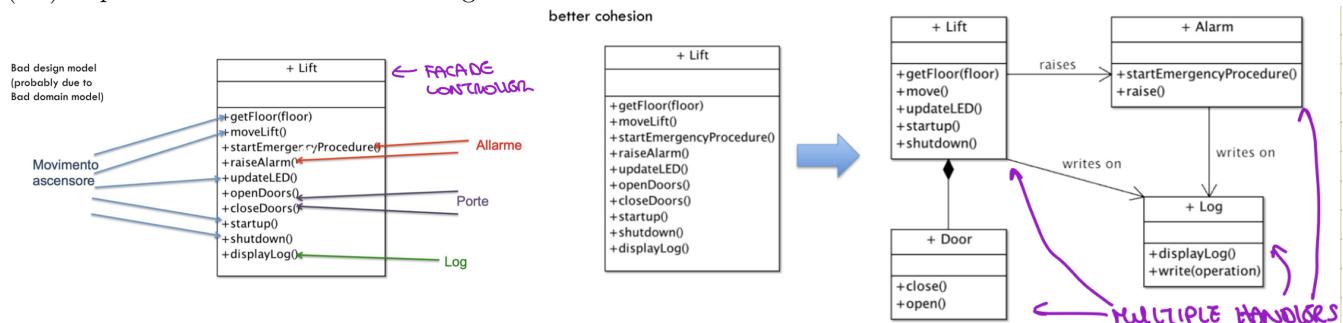
NB: il principio dell'information expert non va usato per la responsabilità di input/output: per questi 2 aspetti deve essere presente e si utilizza un Persistence Services Subsystem (come un database per l'output)

10.5 Controller

Problema: Qual'è l'oggetto nel dominio che è responsabile di ricevere le richieste al sistema?

Soluzione: Il controller è il primo oggetto che è responsabile di ricevere o gestire le richieste al sistema; inoltre ci sono 2 strade da poter percorrere: scegliere una classe oggetto che gestisce tutto il "facade controller", (per pochi eventi) oppure separare gli use cases gestiti dai diversi handler. (per tanti eventi)

Il controller conosce e mantiene anche lo stato del sistema, ad esempio: in un ascensore in cui un facade controller (lift) è spezzato in molti handlers = migliore coesione:

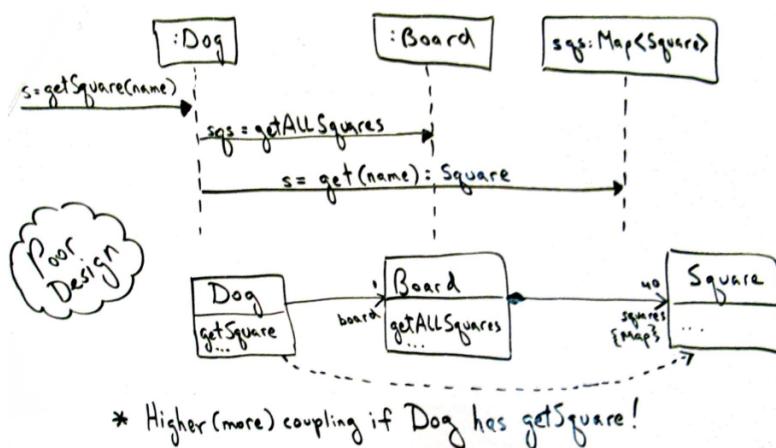


10.6 Low Coupling

Problema: Come ridurre l'impatto di un eventuale cambiamento del sistema?

Soluzione: Assegnare responsabilità... così si mantiene basso il livello di accoppiamento tra classi-oggetti. Infatti conviene inserire un metodo in una classe-oggetto piuttosto che aggiungere connessioni tra gli oggetti.

Esempio: monopoly

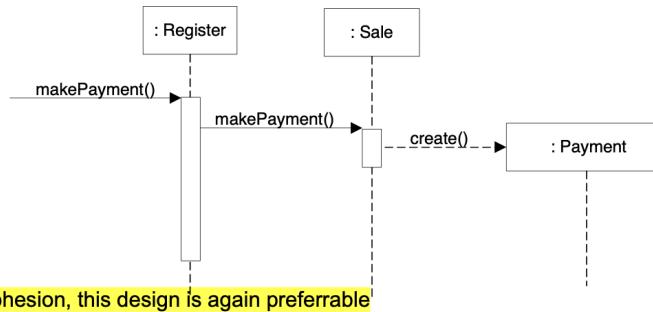


10.7 High Cohesion

Problema: Come gestire le responsabilità in modo che un oggetto non ne sia sovraccarico?

Soluzione: Assegnando le responsabilità in modo che ogni oggetto faccia solo operazioni specifiche, ed inerenti

È complementare al low coupling, ma attenzione! viene accettato solamente se favorisce performance migliori. Un esempio può essere il seguente:



10.8 Pure Fabrication

Problema: Non c'è nessun oggetto appropriato a cui attribuire una responsabilità?

Soluzione: Disegnare una classe-oggetto artificiale da 0. Spesso tali oggetti rappresentano un comportamento a cui attribuire la responsabilità. Lo rivedremo nei GoF-Patterns essendo tutti dei Pure Fabrication

Capitolo 11

GoF Patterns

11.1 Key Points

- Singleton, Factory
- Adapter, Composite, Facade
- Strategy
- Observer
- Command
- Delegation

11.2 Visione d'insieme

I GoF pattern forniscono modelli utili per risolvere situazioni ricorrenti, sono tutti dei "Pure Fabrication". Analizzeremo 3 tipologie di patterns: **creational** che si occupano dell'inizializzazione e della configurazione degli oggetti, **structural** per la composizione di classi o oggetti e per la separazione dell'interfaccia e dell'implementazione delle classi, i **behaviour** che si occupano delle interazioni dinamiche tra gli oggetti, e di come distribuiscono le responsabilità.

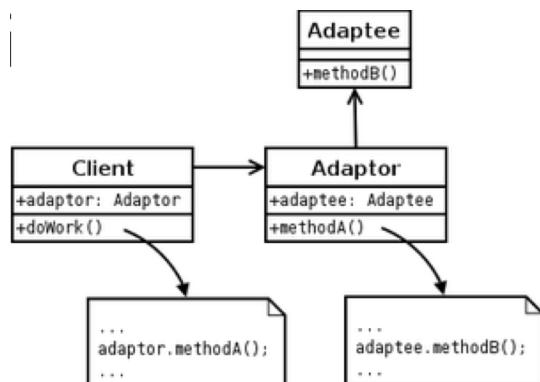
11.3 Adapter

Structural pattern che si occupa di convertire l'interfaccia di una classe in una richiesta dal cliente.

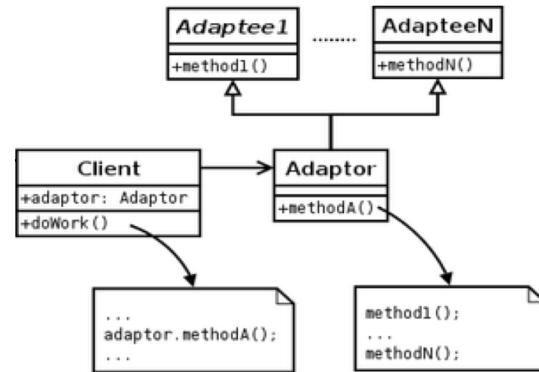
Problema: Come risolvere interfacce incompatibili o fornire un'interfaccia stabile a componenti simili con interfacce diverse?

Soluzione: Convertire il componente con l'interfaccia originale in un'altra interfaccia attraverso un adapter intermedio. L'adapter traduce le chiamate alla sua interfaccia in chiamate all'interfaccia originale e la quantità di codice necessario per farlo è tipicamente piccola. Ci sono due tipi di adapter:

- Adapter a Oggetti:** questo tipo contiene l'istanza della classe che incapsula(Adaptee) così può richiamarne il metodo(methodB()). In questa situazione l'adapter chiama 1. ma l'istanza attraverso l'adaptor methodA() che invoca a sua volta adaptee.method(B). Qui è mostrato l'adapter a Oggetti espresso in UML. L'adapter nasconde l'interfaccia adaptee dal cliente.



- Adapter a Classe:** questo tipo di adapter utilizza molteplici interfacce polimorfiche per raggiungere il suo obiettivo. Viene creato implementando o ereditando sia l'interfaccia attesa che l'interfaccia preesistente. È tipico che
2. l'interfaccia attesa sia creata come una classe di interfaccia pura, specialmente in linguaggi come Java che non supportano l'ereditarietà multipla (eredita l'interfaccia e la implementa, vedi method1()).



11.4 Singleton

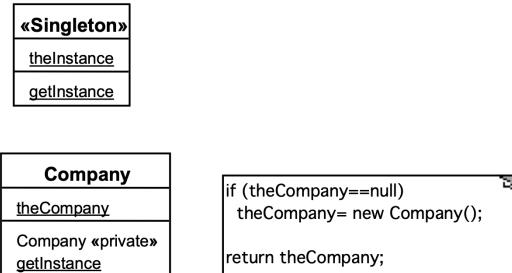
Creational pattern che garantisce accesso a una singola istanza inoltre si assicura che esista una singola istanza della classe in ogni dato momento.

- vi si accede tramite un metodo public static getInstance()
- equivalente all'inizializzazione 'static' di una classe, ma aggiunge encapsulamento, flessibilità e thread safety

Problema: Come si fa ad assicurare che non sia mai possibile creare più di un'istanza di una classe singleton?

Soluzione: Avere una classe privata chiamata "theInstance" che memorizza l'istanza. Poi creare un metodo di classe pubblico (metodo statico) chiamato, ad esempio, "getInstance". La prima volta che il metodo viene chiamato, crea una singola istanza e la memorizza in "theInstance". Le chiamate successive restituiscono semplicemente "theInstance". È necessario avere un costruttore privato che assicura che nessun'altra classe possa creare un'istanza della classe singleton.

Qui, la classe Company può rappresentare diverse importanti caratteristiche dell'azienda (operazioni e attributi). Il metodo public **getInstance()** rende questa istanza globalmente accessibile. Nota: in effetti, l'istanza Singleton è una variabile globale. Riduci al minimo l'utilizzo di queste.



Vi è una critica all'utilizzo del pattern singleton, in quanto alcuni lo considerano un anti-pattern, giudicando che sia troppo utilizzato, introduce restrizioni inutili in situazioni in cui non è effettivamente richiesta una sola istanza di una classe e introduce uno stato globale nell'applicazione. Inoltre il thread safety se non gestito, possibile caso in cui due thread creano l'istanza contemporaneamente. (vengono istanziate 2 o più banche invece che una). È molto complesso da produrre e può incorrere in problemi di scalabilità essendo l'istanza singleton unica, potrebbe rappresentare un collo di bottiglia nel caso più parti del sistema richiedano l'accesso alla risorsa contemporaneamente. (una banca gestisce tutte le transazioni)

esempio codice: <https://stem.elearning.unipd.it/mod/resource/view.php?id=320424>

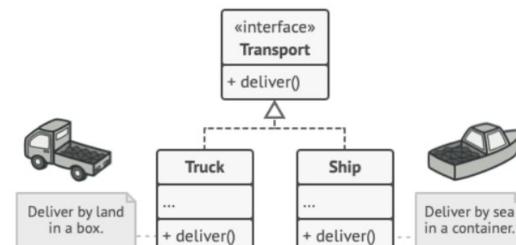
11.5 Factory

Definisce un'interfaccia per creare un oggetto, ma lascia alla sottoclassificazione di decidere quale classe va istanziata.

Problema: Chi dovrebbe essere responsabile della creazione di oggetti quando ci sono considerazioni speciali, come una logica complessa, il desiderio di separare le responsabilità di creazione per una maggiore coesione e così via?

Soluzione: Creare un Pure Fabrication per ampliare la gamma di oggetti che il programma può gestire, preservando le funzionalità esistenti.

Esempio: app di logistica, gestisce trasporti tramite furgone. Come aggiungere il trasporto via nave? modifco l'intero codice oppure creo un'interfaccia interfaccia comune (astratta) che dichiara un metodo (virtuale puro); successivamente creo sottoclassi che implementano il metodo in modi diversi. L'interfaccia "madre" dichiara il metodo **deliver()**, che viene implementato in modo diverso a seconda del tipo dell'oggetto che invoca il metodo.



Può risultare utile quando non si conoscono a priori i tipi esatti di oggetti che il codice dovrà gestire, permette di riciclare il codice con minime modifiche, disaccoppiare l'interfaccia madre dai prodotti concreti. Ma il codice può apparire più complesso: molte sottoclassi vanno implementate
esempio codice: <https://stem.elearning.unipd.it/mod/resource/view.php?id=320429>

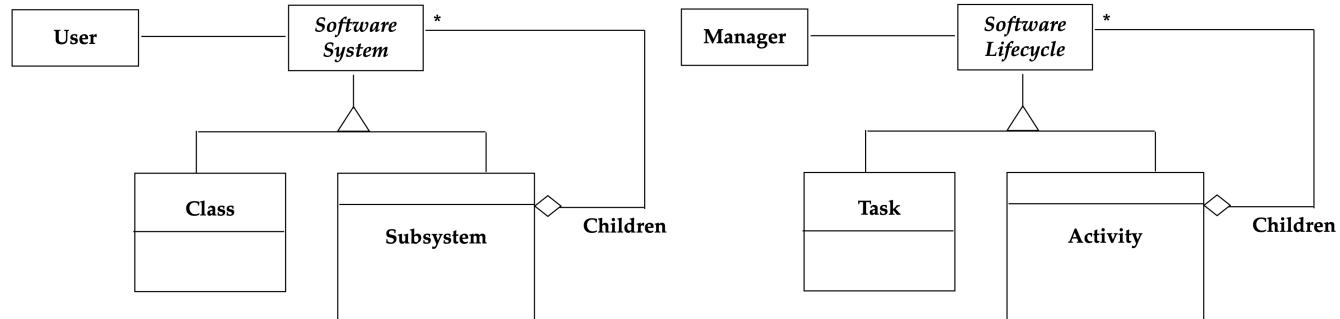
11.6 Strategy

Lo strategy pattern è un design pattern comportamentale(Behaviour) che fornisce un modo per selezionare dinamicamente un algoritmo da un gruppo di algoritmi durante il runtime. In altre parole, fornisce un'astrazione per selezionare uno degli algoritmi specifici basati su un determinato contesto. Ciò consente di modificare il comportamento dell'oggetto in modo dinamico senza dover modificare il codice sorgente. Questo pattern promuove il principio del "single responsibility" poiché separa la logica dell'algoritmo dal resto del codice e consente di cambiare facilmente l'algoritmo utilizzato senza dover modificare l'interfaccia pubblica del sistema.

11.7 Composite

Structural pattern che permette di rappresentare gerarchie di oggetti partendo dalla composizione di oggetti più semplici, che possono essere ulteriormente composti in oggetti più complessi. In pratica, questo pattern consente di trattare in modo uniforme oggetti singoli e oggetti composti, permettendo di lavorare con strutture ad albero di oggetti in modo semplice ed efficiente. Inoltre, il Composite Pattern può essere utilizzato per applicare operazioni ricorsive su tutti gli elementi di una gerarchia di oggetti senza conoscere in anticipo la loro struttura specifica.

Esempio: Cosa c'è in comune tra un sistema software e il suo ciclo di vita? Sia il sistema software che il ciclo di vita del software sono rappresentati come gerarchie di parti e interi che formano una struttura ad albero. Nel pattern Composite, una struttura ad albero viene utilizzata per rappresentare una gerarchia di oggetti, dove gli oggetti possono essere parti o interi. Nella definizione del sistema software, i sottosistemi rappresentano le parti e le collezioni di classi rappresentano gli interi. Nella definizione del ciclo di vita del software, le attività rappresentano le parti e le collezioni di compiti rappresentano gli interi. Le foglie dell'albero rappresentano classi nel sistema software e compiti nel ciclo di vita del software. Il pattern Composite fornisce un modo per trattare le parti e gli interi in modo uniforme e consente di creare strutture gerarchiche complesse.



11.8 Facade

Structural pattern che semplifica l'interfaccia per un subsystem. In altre parole, il pattern Facade fornisce un'interfaccia semplificata per un insieme di classi complesse o un sottosistema, nascondendo la complessità e le dettagliate implementazioni dietro questa interfaccia unificata. Il Facade è utile quando si desidera semplificare l'utilizzo di un sottosistema complesso, migliorare la leggibilità del codice e aumentare l'efficienza del sistema. Inoltre, questo pattern favorisce anche il principio di separazione dell'interfaccia dall'implementazione, in quanto l'interfaccia semplificata fornita dalla Facade è indipendente dall'implementazione delle classi nel sottosistema. Ad esempio, immaginiamo di avere un sottosistema di classi complesse utilizzate per la gestione dei dati. Questo sottosistema potrebbe includere classi per la lettura e la scrittura dei dati su diversi formati di file, come CSV, JSON, XML e così via. Invece di dover conoscere e utilizzare tutte queste classi complesse direttamente, si può creare una Facade che semplifichi l'utilizzo del sottosistema. La Facade potrebbe avere metodi pubblici che semplificano l'accesso ai diversi formati di file, nascondendo i dettagli di implementazione complessi delle classi del sottosistema. In questo modo, la Facade semplifica notevolmente l'utilizzo del sottosistema, eliminando la necessità di conoscere la complessità dell'implementazione delle classi del sottosistema.

11.9 Observer

Behaviour che stabilisce una dipendenza uno-a-molti tra gli oggetti, in modo che quando uno dei soggetti cambia il suo stato, tutti gli oggetti dipendenti da esso siano notificati e aggiornati automaticamente. In altre parole, un oggetto, chiamato "soggetto" o "observable", tiene traccia degli oggetti dipendenti da esso, chiamati "osservatori" o "observer", e li notifica automaticamente di qualsiasi cambiamento di stato (low coupling). L'Observer pattern è utile quando si deve garantire che gli oggetti in un sistema siano tenuti sincronizzati con il loro stato corrente, senza che ci sia bisogno di codice ripetitivo per ogni oggetto interessato. Ad esempio, un'applicazione di previsioni meteorologiche può utilizzare l'Observer pattern in modo che ogni volta che un sensore registra una nuova lettura, i display delle previsioni e le notifiche vengano automaticamente aggiornati. In questo modo, il sistema può essere esteso in modo efficiente per includere nuovi display o notifiche senza dover modificare il codice sorgente esistente. Inoltre può risultare utile per dei GUI widget che si aggiornano automaticamente ad ogni cambiamento di stato. In sintesi, l'Observer pattern permette una facile gestione delle dipendenze tra gli oggetti in un sistema, garantendo che gli oggetti osservatori siano sempre allineati con gli oggetti soggetto senza la necessità di codice ripetitivo. esempio codice: <https://stem.elearning.unipd.it/mod/resource/view.php?id=322170>

11.10 Command

Behaviour che permette di incapsulare una richiesta come un oggetto, permettendo di parametrizzare i client con diverse richieste, accodare o registrare richieste e implementare funzionalità di undo/redo. In pratica, il pattern prevede la definizione di una classe Command che rappresenta una certa azione da compiere, ad esempio una operazione di calcolo o una modifica ad uno stato interno di un oggetto. La classe Command espone un metodo 'execute()' che rappresenta l'esecuzione dell'azione. Un oggetto chiamato Invoker conosce la classe Command e l'utilizza per incapsulare le richieste di azione che riceve dai client. L'Invoker può anche tenere traccia delle richieste passate e permettere l'annullamento delle stesse attraverso una funzionalità undo/redo. Il pattern può essere usato per separare l'oggetto che esegue l'azione (il Receiver) dall'oggetto che richiede l'azione (il client), permettendo di variare facilmente l'azione eseguita dal receiver senza dover modificare il client. Inoltre, il pattern supporta l'inserimento di nuove funzionalità senza dover modificare il codice esistente.

11.11 Delegation

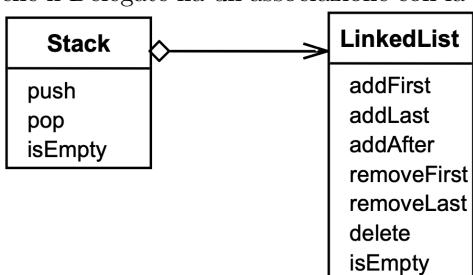
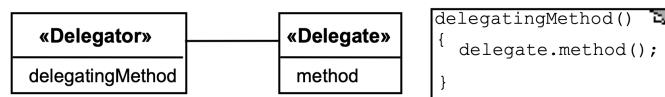
Contesto:

- Stai progettando un metodo in una classe
- Ti rendi conto che un'altra classe ha un metodo che fornisce il servizio richiesto
- L'ereditarietà non è appropriata

Problema: Come puoi utilizzare in modo più efficace un metodo già esistente nell'altra classe?

Vuoi minimizzare il costo di sviluppo riutilizzando i metodi; inoltre, ridurre l'accoppiamento tra le classi. Inoltre, i metodi dovrebbero essere vicini ai dati e la classe esistente con il metodo potrebbe essere il luogo più appropriato per il metodo esistente.

Soluzione: Creare un metodo in una classe Delegator che chiama solo un metodo in una classe Delegate vicina. In questo modo, stiamo riutilizzando il metodo di cui il Delegate ha la responsabilità. Con classe vicina, intendiamo che il Delegate ha un'associazione con la classe Delegator.



```
push()
{
    list.addFirst();
}
```

Qui possiamo vedere che le operazioni dello Stack 'push', 'pop' e 'isEmpty' possono facilmente utilizzare i metodi esistenti della classe Linked List - 'addFirst', 'addLast' e 'isEmpty'. Pertanto, abbiamo la relazione sopra indicata.

esempio codice: <https://stem.elearning.unipd.it/mod/resource/view.php?id=320429>

Capitolo 12

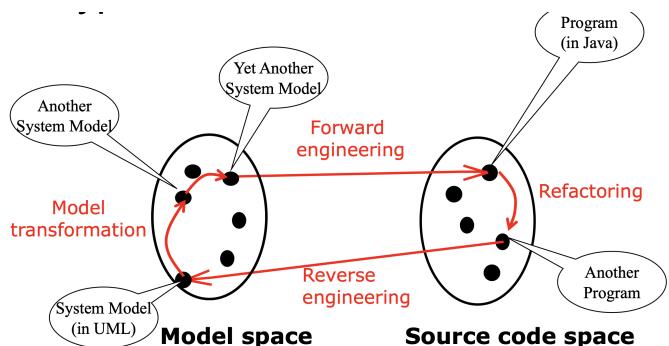
Design to Code

12.1 Key Points

- Model transformations
- Refactoring
- Forward engineering (dal modello alla classe)
- Model-driven engineering

12.2 Visione d'insieme

Descriveremo le 4 trasformazioni sul modello e codice. La trasformazione più rilevante è quella dal modello al codice, o forward engineering; Alcuni strumenti consentono una (parziale) automazione di questo processo di trasformazione.



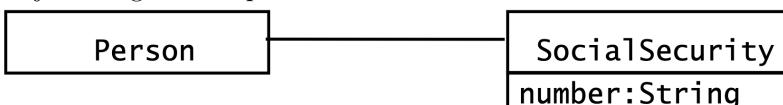
12.3 Model transformations : EREDITARIETÀ

La model transformation consiste nel trasformare il modello per:

- **migliorare l'ereditarietà** oppure introdurla dove possibile
- **generalizzare** e trovare una superclasse da utilizzare
- **collassare oggetti** riducendo le classi-oggetto che hanno poche funzionalità ad attributi di altre classi oggetto

In sintesi, per preparare le classi alla generalizzazione, è necessario assicurarsi che tutte le operazioni abbiano la stessa firma, ma spesso le firme non corrispondono. Le superclassi sono utili perché aumentano la modularità, l'estendibilità e la riusabilità, migliorano la gestione della configurazione e sono utilizzate da molti design pattern. È possibile modificare un modello esistente per consentire l'utilizzo di un design pattern.

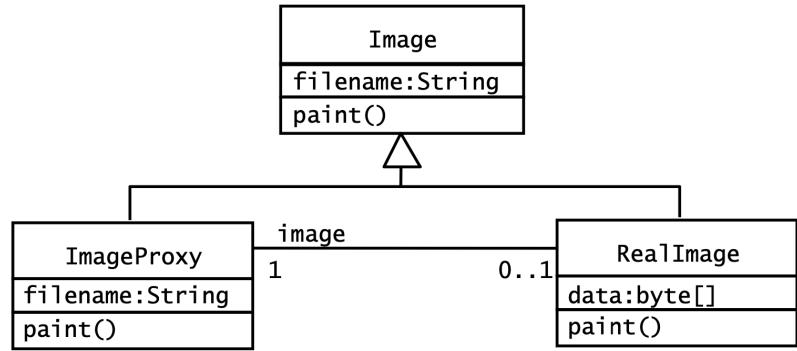
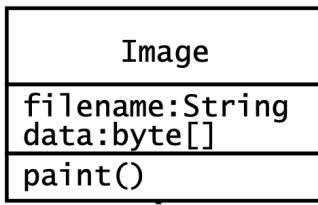
Object design model prima della trasformazione:



Object design model dopo della trasformazione:



La trasformazione di un oggetto in un attributo di un altro oggetto viene solitamente effettuata se l'oggetto non ha alcun comportamento dinamico interessante (solo operazioni get e set).



12.4 Refactoring

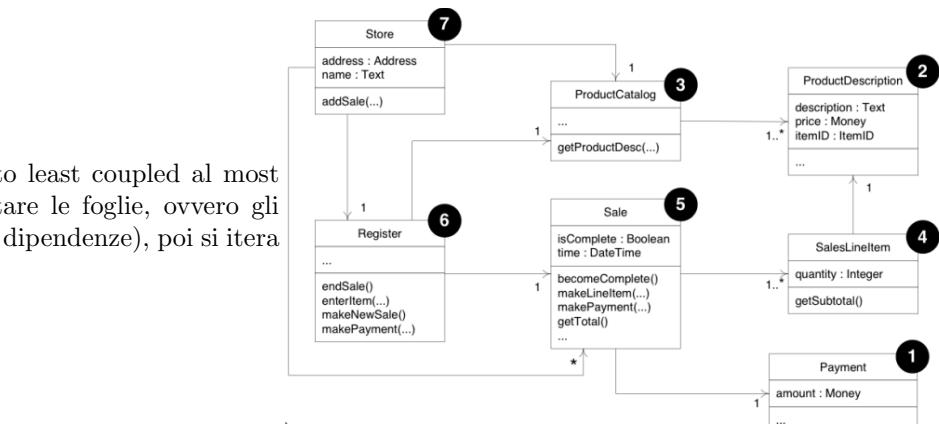
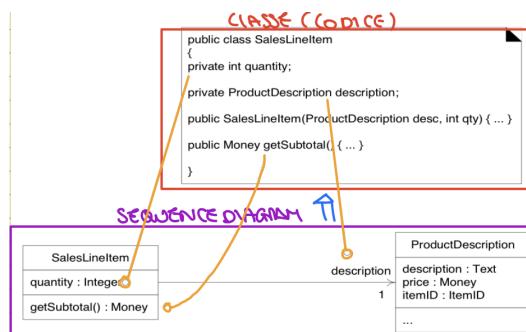
Il refactoring è il processo di ristrutturazione del codice sorgente al fine di migliorare la sua qualità, la leggibilità, la manutenibilità, l'efficienza o altre sue caratteristiche senza modificarne il comportamento esterno. Questo processo prevede l'applicazione di tecniche specifiche al fine di rendere il codice più chiaro, snello e organizzato, in modo che sia più facile da leggere e da mantenere nel tempo. Il refactoring può essere effettuato su qualsiasi tipo di codice, sia esso software, script o pagine web, e può essere eseguito manualmente o con l'ausilio di strumenti automatici di analisi e di refactoring del codice. (praticamente come la model transformation ma solo a livello di codice)

12.5 Forward engineering

Obiettivo: Implementare il modello di progettazione degli oggetti in un linguaggio di programmazione attraverso: definizioni di classi e interfacce, definizioni di metodi, lavorare da OOA/D artifacts, creare definizioni di classe da diagrammi di classe di progettazione (DCD), mappatura dell'ereditarietà, delle dipendenze e delle associazioni.

12.5.1 Create class definitions

Il sequence diagram mi fornisce lo scheletro delle implementazioni, le frecce invece sono i metodi.

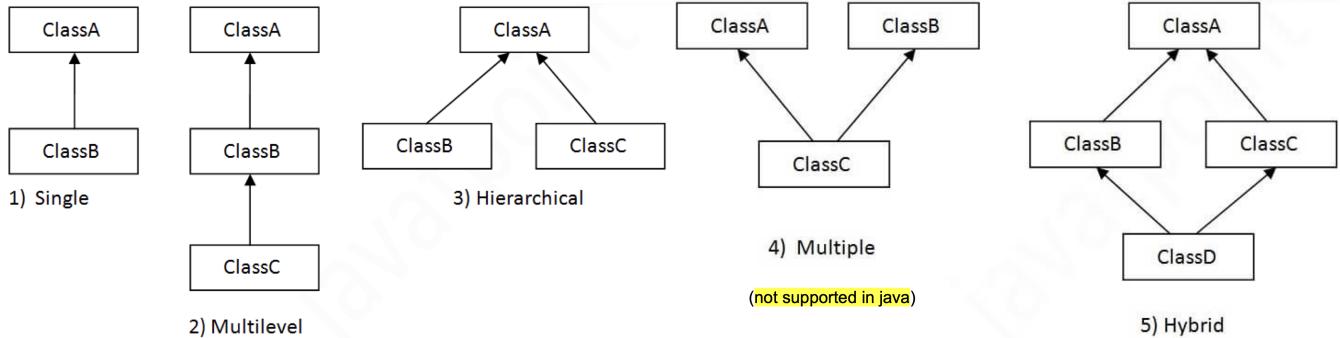


L'implementazione va dall'oggetto least coupled al most coupled. Si inizia ad implementare le foglie, ovvero gli oggetti con meno coupling (meno dipendenze), poi si itera tale procedimento.

12.5.2 Realize inheritance

Obiettivo: abbiamo un modello UML con ereditarietà e vogliamo tradurlo in codice sorgente. Domanda: quali meccanismi del linguaggio di programmazione possono essere utilizzati? Focalizziamoci su Java. Java fornisce i seguenti meccanismi: sovrascrittura dei metodi (predefinita in Java), classi finali, metodi finali, metodi astratti, classi astratte, interfacce.

- Specializzazione e generalizzazione: definizione di sottoclassi (Parola chiave di Java: extends)
- Realizzazione di ereditarietà semplice: la sovrascrittura dei metodi non è consentita (Parola chiave di Java: final)
- Implementazione ereditarietà: sovrascrittura dei metodi (Nessuna parola chiave necessaria: la sovrascrittura dei metodi è predefinita in Java)
- Specifica di ereditarietà: Specifica di un'interfaccia (Parole chiave di Java: abstract, interface)



12.5.3 Map dependencies

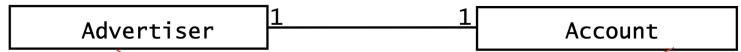
Una relazione di dipendenza denota una relazione semantica tra elementi del modello, dove una modifica nella classe fornitrice può causare una modifica nel cliente. La semantica si concentra sulla relazione tra i significati, come parole, frasi, segni, simboli, ecc. Cosa succederebbe se java.sql cambiasse, dove tutti i nostri meccanismi di persistenza usano (dipendono) dagli oggetti in questo pacchetto? È necessario determinare cosa causa la visibilità del fornitrice al cliente.

Ecco 4 tipi di "communications pathways" dal fornitrice al cliente:

- Riferimento a variabile locale = L'oggetto fornitrice è dichiarato localmente (creato temporaneamente durante l'esecuzione di un'operazione/metodo). L'oggetto cliente dichiara una variabile locale all'interno di un metodo specifico all'interno del cliente.
- Riferimento al parametro = L'oggetto fornitrice è un parametro o la classe di ritorno di un'operazione nell'oggetto cliente. Un metodo nel cliente ha un parametro formale di tipo fornitrice o il metodo restituisce un oggetto di tipo fornitrice (File Reader.... Questi restituiscono altri oggetti che gestiscono l'I/O...)
- Riferimento globale = L'oggetto fornitrice è globale e autoesplicativo
- Riferimento al campo = L'oggetto fornitrice è un membro dati nell'oggetto cliente. C'è una variabile di istanza all'interno dell'oggetto di tipo fornitrice.

12.5.4 Map associations

Object design model before transformation:



Source code after transformation:

```

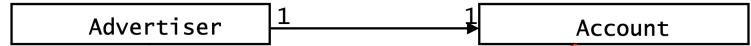
public class Advertiser {
    private Account account;
    public Advertiser() {
        account = new Account();
    }
    public Account getAccount() {
        return account;
    }
}

```

i0/04/23

14 - design to code

Object design model before transformation:



Source code after transformation:

```

public class Advertiser {
    /* account is initialized
     * in the constructor and never
     * modified. */
    private Account account;
    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}

public class Account {
    /* owner is initialized
     * in the constructor and
     * never modified. */
    private Advertiser owner;
    public Account(owner:Advertiser) {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}

```

Object design model before transformation:



Source code after transformation:

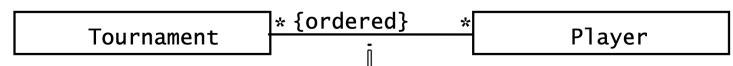
```

public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount(Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}

public class Account {
    private Advertiser owner;
    public void setOwner(Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
                newOwner.addAccount(this);
            if (oldOwner != null)
                old.removeAccount(this);
        }
    }
}

```

Object design model before transformation



Source code after transformation

```

public class Tournament {
    private List players;
    public Tournament() {
        players = new ArrayList();
    }
    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }
}

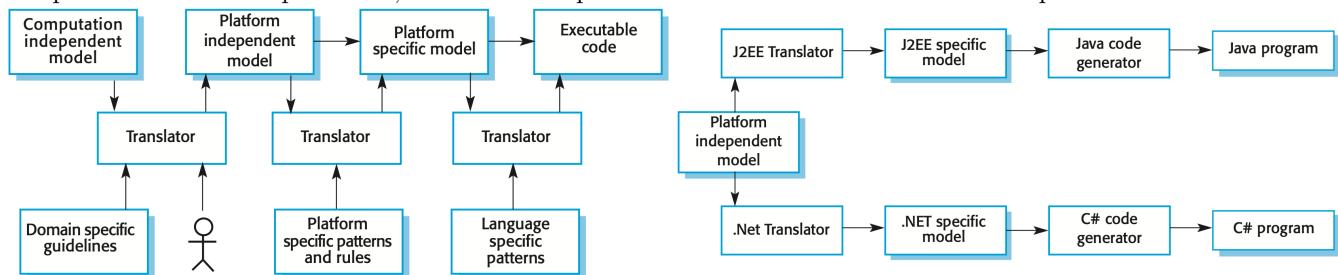
public class Player {
    private List tournaments;
    public Player() {
        tournaments = new ArrayList();
    }
    public void addTournament(Tournament t) {
        if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
        }
    }
}

```

4. many-to-many association (bidirezionale):

12.6 Model-driven engineering

Il Model-Based Software Engineering è un'approccio alla progettazione del software in cui i modelli sono gli output principali del processo di sviluppo e i programmi che vengono eseguiti sono generati automaticamente dai modelli. Questo solleva gli ingegneri dai dettagli del linguaggio di programmazione e della piattaforma di esecuzione. L'MBE è ancora in una fase precoce di sviluppo e non è chiaro se avrà un impatto significativo sulla pratica dell'ingegneria del software. Tuttavia, i vantaggi dell'approccio includono la possibilità di considerare i sistemi a livelli di astrazione più elevati e la possibilità di generare codice in modo automatico, il che rende più facile adattare i sistemi a nuove piattaforme. Gli svantaggi includono il fatto che i modelli per l'astrazione non sono necessariamente corretti per l'implementazione e che i risparmi derivanti dalla generazione di codice possono essere compensati dai costi di sviluppo dei traduttori per nuove piattaforme. Ci sono tre tipi di modelli: il Modello Indipendente dalla Computazione, il Modello Indipendente dalla Piattaforma e i Modelli Specifici della Piattaforma.



Capitolo 13

Implementation issues

13.1 Key Points

- Reuse
- Configuration management
- Host-target development
- Open source development

13.2 Visione d'insieme

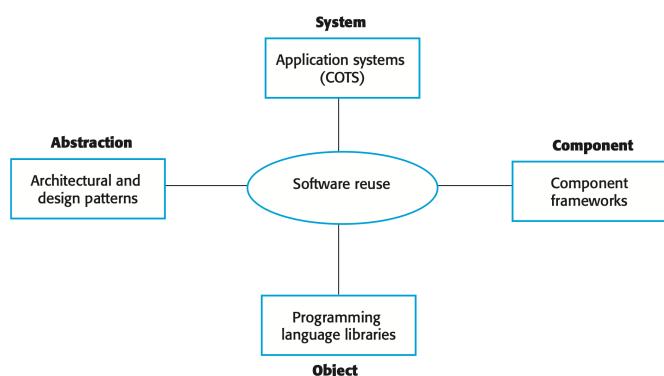
Durante lo sviluppo del software, è sempre opportuno considerare la possibilità di riutilizzare software esistente, sia come componenti, servizi o sistemi completi. Configuration management è il processo di gestione delle modifiche ad un sistema software in evoluzione. È essenziale quando un team di persone collabora allo sviluppo del software. La maggior parte dello sviluppo software è basato su una configurazione host-target. Si utilizza un IDE su una macchina host per sviluppare il software, che viene poi trasferito su una macchina target per l'esecuzione. Lo sviluppo open source implica la pubblicazione del codice sorgente di un sistema. Ciò significa che molte persone possono proporre modifiche e miglioramenti al software.

13.3 Reuse

Un approccio allo sviluppo basato sul riutilizzo del software esistente è fondamentale. Ecco i livelli per cui vale questo approccio:

- Livello di **astrazione**: non si riutilizza direttamente il software ma si utilizza la conoscenza dei design pattern di successo nella progettazione del software.
- Livello di **oggetti**: si riutilizzano direttamente gli oggetti di una libreria anziché scrivere il codice da soli.
- Livello di **componenti**: sono collezioni di oggetti e classi di oggetti che vengono riutilizzati in sistemi applicativi.
- Livello di **sistema**: si riutilizzano interi sistemi applicativi.

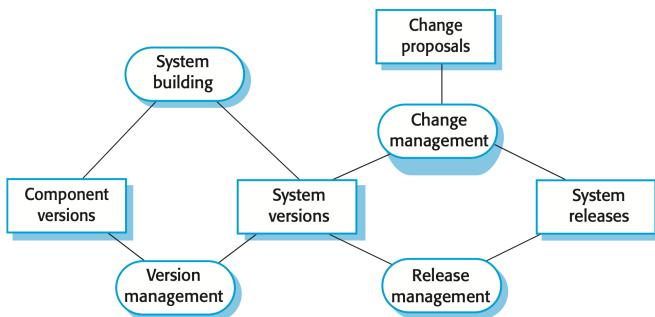
Attenzione ai **costi** che possono derivare dal tempo impiegato per cercare il software da riutilizzare, costi d'acquisto del software, costi d'adattamento e della configurazione dei componenti o dei sistemi software per riflettere i requisiti del sistema; oltre ai costi dell'integrazione degli elementi software (se stai utilizzando software proveniente da diverse fonti) con il nuovo codice che hai sviluppato.



13.4 Configuration management

Configuration management è il nome dato al processo generale di gestione di un sistema software in continua evoluzione. Lo scopo è di supportare il processo di integrazione in modo che tutti i programmatore possano accedere al codice del progetto e ai documenti in modo controllato, scoprire quali cambiamenti sono stati apportati, e compilare e collegare i componenti per creare un sistema.

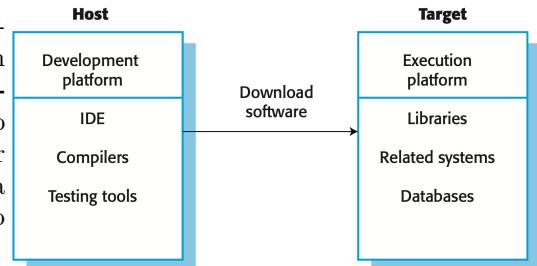
Le 3 fondamentali attività sono:



1. **Gestione delle versioni**, dove viene fornito supporto per tenere traccia delle diverse versioni dei componenti software
2. **L'integrazione di sistema**, dove viene fornito supporto per aiutare gli sviluppatori a definire automaticamente un sistema che integri nuove parti
3. **Il tracciamento dei problemi**, dove viene fornito supporto per consentire agli utenti di segnalare bug e altri problemi, e per consentire a tutti gli sviluppatori di vedere chi sta lavorando su questi problemi e quando vengono risolti

13.5 Host-target development

La maggior parte del software viene sviluppato su un computer (Host), ma viene eseguito su una macchina separata (Target). Più in generale, possiamo parlare di **piattaforma di sviluppo** e di **piattaforma di esecuzione** (include il sistema operativo installato e altro software di supporto come un sistema di gestione di database o, per le piattaforme di sviluppo, un ambiente di sviluppo interattivo). La piattaforma di sviluppo di solito ha un software installato diverso rispetto alla piattaforma di esecuzione oltre a diverse architetture.



ATTENZIONE: È importante che sia **garantita compatibilità tra i due sistemi**.

13.6 Open source development

Lo sviluppo open source è un approccio dove il codice sorgente di un software è pubblicato e i volontari sono invitati a partecipare al processo di sviluppo. Ci sono molti prodotti open source importanti, tra cui il sistema operativo Linux, Java, il server web Apache e il sistema di gestione del database MySQL. Molti produttori stanno utilizzando un approccio open source per lo sviluppo di software perché credono che coinvolgere la comunità open source permetterà di sviluppare il software più rapidamente. Anche se il codice sorgente è libero, ci possono essere restrizioni sul suo utilizzo.

- Vantaggi: libero e gratuito
- Svantaggi: ho poco controllo su quello che viene fatto

Capitolo 14

Software Testing

14.1 Key Points

- Development Testing
- Release Testing
- User Testing

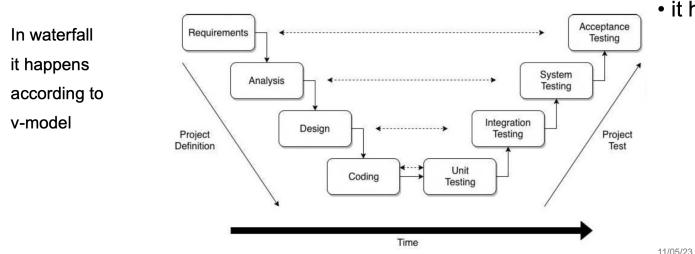
14.2 Visione d'insieme

Rappresenta una delle parti del Validation-Verification in cui si inseriscono dei dati e viene validata la risposta,(approccio BLACKBOX) inoltre vengono evidenziati gli errori, ma non tutti. Gli obiettivi sono:

- **validation testing:** dimostrare ai dev. e al consumatore che il sw funzioni
- **defect testing:** evidenziare i difetti nel sw

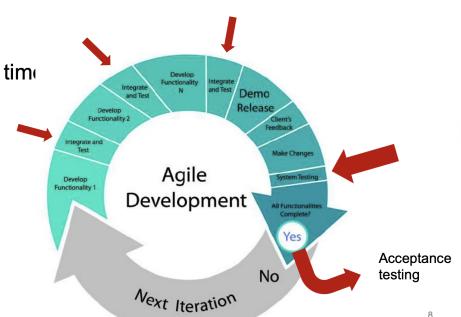
Il testing può avvenire con diverse fasi di sviluppo in base al tipo di testing:

V-model testing



Agile testing

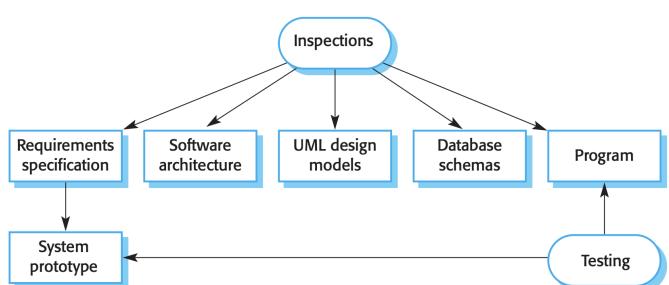
- In an agile model
- it happens multiple times



I criteri del V&V sono:

1. sw purpose
2. user expectation
3. marketing environment (importanza nel mettere il prodotto sul mercato rapidamente)

Inoltre esistono le **ispezioni** del software che si concentrano sull'analisi statica della rappresentazione del sistema per individuare problemi (verifica statica) e possono essere integrate dall'uso di strumenti per analizzare documenti e codice. A differenza del **sw testing** che si occupa di testare e osservare il comportamento del prodotto (verifica dinamica) mediante l'esecuzione del sistema con dati di prova e l'osservazione del suo comportamento operativo.



14.3 Development Testing

Viene fatto dal team di sviluppatori durante lo sviluppo del sw per scoprire bug e difetti(nell'agile anche 1 volta al giorno). Si suddivide a sua volta in:

14.3.1 Unit Testing

Validare funzionalità di oggetti e metodi (fatto con JUnit). È un **defect testing** delle classi con tutti i suoi metodi e in tutti i stati attraverso l'utilizzo di **TEST-CASE** cioè un insieme di test che testano metodi di una classe attraverso diversi input e condizioni(ogni metodo ha un suo test-case). Lo unit test deve essere automatico attraverso dei test-automation frameworks (come JUnit). Esistono 2 tipi di unit Test-case:

1. *Operazioni normali*: con input attesi
2. *Operazioni inattese*: con input non previsti

Il test si divide in 3 parti:

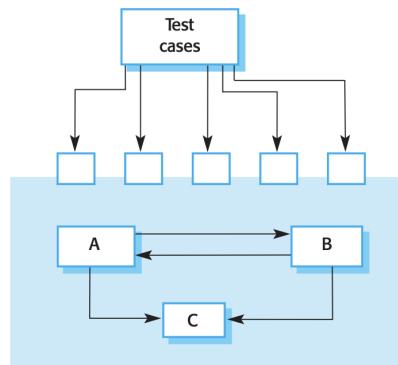
1. **Setup-part** = inizializzare il sistema con il test case
2. **Call-part** = chiamare oggetti e metodi da testare
3. **Assertion-part** = comparare i risultati della chiamata con quelli attesi

Infine ci sono diverse strategie di testing:

- **Partition testing**: identificare alcune partizioni di dominio(input/output) e testarle. Attenzione, conviene sempre testare gli estremi di dominio a parte dato che è più facile incorrere in errori
- **Testing guidelines**: sono alcune linee guida da seguire, tra cui testare Estremi di dominio, grandezze diverse, sequenze nulle, messaggi d'errore, forzare invalid output e risultati di grandezza variabile

14.3.2 Component Testing

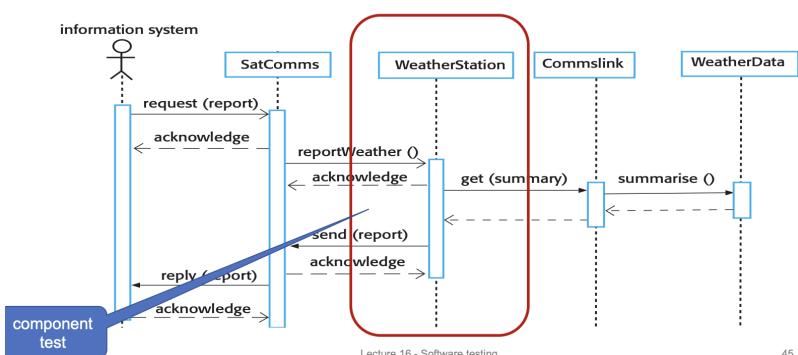
I componenti software sono spesso componenti composti che sono costituiti da diversi oggetti che interagiscono tra loro. Ad esempio, nel sistema di una stazione meteorologica, il componente di riconfigurazione include oggetti che gestiscono ciascun aspetto della riconfigurazione. L'accesso alla funzionalità di questi oggetti avviene attraverso l'**interfaccia del componente** definita. L'obiettivo è verificare che i componenti siano in grado di comunicare correttamente tra loro e di rispettare i requisiti definiti nell'interfaccia del componente. È importante sottolineare che durante la prova dei componenti si assume che le prove a livello di unità sugli oggetti interni al componente siano state eseguite con successo. In questo modo, la prova del componente si concentra sull'integrazione e sul comportamento dell'interfaccia del componente, piuttosto che sulle singole unità; è un passo cruciale nel processo di verifica e convalida, in quanto consente di identificare eventuali problemi o errori di **interface misunderstanding** che possono sorgere nell'interazione tra gli oggetti all'interno di un componente composito.



14.3.3 System Testing

Durante lo sviluppo coinvolge l'integrazione dei componenti per creare una versione del sistema e quindi testare il sistema integrato. L'obiettivo del system test è testare le interazioni tra i componenti, verifica che i componenti siano compatibili, interagiscano correttamente e trasferiscano i dati corretti al momento giusto attraverso le loro interfacce. Durante il test, componenti riutilizzabili sviluppati separatamente e sistemi preconfezionati possono essere integrati con componenti di nuova realizzazione. Il sistema completo viene quindi testato.

Collect weather data sequence chart



14.3.4 Testing policies (development testing)

Esistono policies per quanto riguarda il testing per cui tutte le funzioni accessibili attraverso interfaccia grafica e tutti gli user input devono essere testati

14.3.5 Test-driven development

Il Test-Driven Development (TDD) è un approccio allo sviluppo del software in cui si alternano i test e lo sviluppo del codice. I test vengono scritti prima del codice, superare i test è l'elemento fondamentale dello sviluppo. Il codice viene sviluppato incrementalmente insieme al proprio test. Non si passa all'incremento successivo fino a quando il codice sviluppato non supera i test. Il TDD è stato introdotto come parte dei metodi agili come l'Extreme Programming, ma può essere utilizzato anche nei processi di sviluppo pianificati. Inoltre include Regression testing: una suite di test che viene ri-eseguita ogni volta che viene apportata una modifica al programma. In un processo di testing manuale, il regression testing è costoso, ma con il testing automatizzato è semplice e diretto. I test devono essere eseguiti "con successo" prima che la modifica venga consolidata.

14.4 Release Testing

Il release testing è il processo di testing di una versione specifica di un sistema destinato all'uso al di fuori del team di sviluppo. L'obiettivo principale del processo di release testing è convincere il fornitore del sistema che sia abbastanza valido per l'uso; pertanto, deve dimostrare che il sistema offre la sua funzionalità, prestazioni e affidabilità specificate, e che non fallisce durante l'uso normale. Il release testing è di solito un processo di testing black-box in cui i test sono derivati solo dal sistema: è una forma di system testing. Un team separato che non è stato coinvolto nello sviluppo del sistema dovrebbe essere responsabile del release testing attraverso l'utilizzo di scenari.

14.5 User Testing

È una fase del processo di testing in cui gli utenti o i clienti forniscono input e consigli sul testing del sistema. Il testing utente è essenziale, anche quando sono stati effettuati test approfonditi di sistema e di rilascio. Esistono 3 tipi di user testing:

- **Alpha testing:** gruppo ristretto di utenti appartente al team di sviluppo
- **Beta testing:** una versione del software viene resa disponibile per più utenti per consentire loro di sperimentare e segnalare problemi che scoprono agli sviluppatori del sistema
- **Acceptance testing:** i clienti testano un sistema per decidere se è pronto per essere accettato dagli sviluppatori del sistema e implementato nell'ambiente del cliente, principalmente per sistemi personalizzati.

ATTENZIONE! NEL PROGETTO NON SARANNO RICHIESTE RELEASE TESTING E USER TESTING

Capitolo 15

Software Evolution

15.1 Key Points

- Evolution processes
- Legacy systems
- Software maintenance

15.2 Visione d'insieme

Capitolo 16

Domande frequenti

1. FACADE CONTROLLER VS USE CASE CONTROLLER
2. DUE ESEMPI DI ARCHITECTURAL PATTERN
3. SISTEMI A SCATOLA CHIUSA E SCATOLA APERTA NEL V&V
4. STATE DIAGRAM
5. ACTIVITY DIAGRAM
6. V MODEL (CHIESTA 4 VOLTE)
7. STRATEGY PATTERN
8. QUALI SONO I TIPI DI DIAGRAMMI(ACTIVITY,SEQUENCE...)
9. OBSERVER PATTERN (CHIESTA 2 VOLTE)
10. TRE RUOLI AGILE SCRUM
11. DIFFERENZA TRA SYSTEM E RELEASE TEST (CHIESTA 3 VOLTE)
12. QUALI SONO I MODELLI ARCHITETTURALI(PEER2PEER,CLIENT SERVER,...)
13. DIFFERENZA TRA ACTIVITY DIAGRAM E SEQUENCE
14. SCRUM (3 ruoli, 3 artifatti)
15. Tipo nel waterfall, che partì dal design/elicitation, arrivi all'implementazione e poi fai i test, partendo dagli unit test, poi system test eccetera