

Ingegneria del Software

Riassunto concetti chiave

Andrea Mansi - UNIUD 2019-2020

Bibliografia:

- libro "Ingegneria del software 10° ed." Ian Sommerville (ISBN-13: 978-8891902245)
- libro "Analisi e progettazione di sistemi software industriali vol. 1" prof. Andrea Baruzzo (ISBN-13: 978-1977702050)
- slides prof. C. Tasso e prof. A. Baruzzo (UniUD) anno accademico 19/20

(la numerazione dei capitoli combacia con quella del primo libro; i relativi riassunti di tali capitoli sono spesso integrati con materiale delle slide e del secondo libro)

Il PDF è un riassunto dei concetti chiave e delle notazioni UML utili per il ripasso del corso di Ingegneria del Software, è dunque consigliato l'acquisto e la lettura di entrambi i libri per una preparazione completa.

Indice

1. Introduzione	4
2. Processi software	5
Classificazione dei processi software (standard ISO/IEC 12207)	5
Modelli dei processi software	5
Modello a cascata (waterfall model)	6
Sviluppo incrementale.....	6
Integrazione e configurazione (modelli orientati al riuso)	7
Sviluppo evolutivo (prototyping).....	7
Modelli formali	7
Attività di processo.....	8
Specificazione del software.....	8
Sviluppo del software (progettazione e implementazione).....	8
Convalida del software	9
Evoluzione del software	9
Project Management	9
3. Sviluppo agile del software.	10
Metodi agili	10
Tecniche di sviluppo agile	11
Storie utente	11
Refactoring	12
Sviluppo con test iniziali	12
Programmazione a coppie	12
Gestione agile della progettazione	12
Scalabilità dei metodi agili.....	12
Metodi agili guidati da piani	13
4. Ingegneria dei requisiti	14
Requisiti funzionali e non funzionali	14
Requisiti funzionali	15
Requisiti non funzionali	15
Processi di ingegneria dei requisiti	16
Deduzione e analisi dei requisiti	16
Specificazione dei requisiti	17
Convalida dei requisiti	17

Modifica dei requisiti (gestione dei requisiti)	18
5. Modelli di sistema.....	19
Architettura guida da modelli.....	19
UML.....	20
Casi d'uso	20
Diagramma delle classi.....	23
Diagrammi di sequenza.....	27
Storie utente (user story mapping).....	34
DFD – Data Flow Diagram	35
Diagramma di contesto e DFD di livello 0.....	35
Linee guida	36
Schema HIPO: Hierachical Input Process Output	36
Structure chart (SC).....	36
Reti di Petri	37
es. Reti di Petri	39
Esercizio su UML e OOD (vecchio esame con svolgimento del prof. A. Baruzzo)	40
es. Casi d'uso	40
es. Diagramma delle classi	41
es. Diagramma di sequenza.....	41
es. Metriche di Lakos	42
6. Progettazione architettonale	43
Decisioni di progettazione architettonurale	43
Viste architettonurali	44
Schemi architettonurali	45
Architetture applicative.....	46
7. Progettazione e implementazione.....	47
Object-Oriented design.....	48
Coesione e accoppiamento	48
Principi SOLIDI del design OO	49
Progettazione per i test (design for testability - DFT).....	50
Stratificazione fisica e testing	51
Progettazione aciclica	51
Component dependency and normalized metrics (LAKOS)	52
Complessità ciclomatica CC.....	52
Debito tecnico.....	53
8. Test del software.....	54
Fasi dei test di sviluppo	56
Test delle unità	56
Test delle componenti	57
Test del sistema	57
Sviluppo guidato da test	58
9. Evoluzione del software.....	59
Processi evolutivi	59
Sistemi ereditati (legacy systems).....	60
Manutenzione del software	60

Previsione della manutenzione	61
Reingegnerizzazione del software	61
10. Sistemi fidati	62
Fidatezza in quanto proprietà	62
Sistemi sociotecnici	63
Ridondanza	64
Processi software fidati	64
Metodi formali	65
11. Ingegneria dell'affidabilità	66
Disponibilità e affidabilità	66
Requisiti	67
Metriche di affidabilità e disponibilità	67
Architetture fidate	68
Sistemi di protezione	68
Architetture auto-monitorate	68
Programmazione a N versioni	69
Diversità	69
Misura dell'affidabilità	69
12. Ingegneria della sicurezza	70
Requisiti	70
13. Ingegneria della protezione	71
16. Ingegneria del software basato sui componenti	72
Componenti	72
17. Ingegneria del software distribuito	73
18. Ingegneria del software orientato ai servizi	74
19. Gestione della progettazione	75
Gestione dei rischi	76
20. Pianificazione dei progetti	77
21. Gestione della qualità	78
Qualità del software	79
Standard	79
Revisioni e ispezioni	80
Gestione qualità e sviluppo agile	81
Misure software	81
Metriche prodotto	82
Esercizi di teoria	83

1. Introduzione

L'ingegneria del software ha lo **scopo di supportare lo sviluppo di software professionale**, anziché di software individuale. Include tecniche che supportano le specifiche, la progettazione e l'evoluzione dei programmi software. L'ingegneria del software fa parte dell'ingegneria dei sistemi, che riguarda tutti gli aspetti tecnico-informatici: ingegneria dell'hardware, software e dei processi.

L'ingegneria del software è una disciplina ingegneristica che riguarda tutti gli aspetti della produzione del software, dalle prime fasi della specifica fino alla manutenzione dopo la messa in produzione. Il software non identifica soltanto i programmi, ma anche tutta la documentazione associata, le librerie, i siti web di supporto e i dati di configurazione.

Esistono **due tipologie fondamentali di software**:

- **Prodotti generici**: sistemi autosufficienti prodotti da un'organizzazione che si occupa di sviluppo e disponibili sul mercato per i clienti.
- **Prodotti personalizzati** (o su richiesta/commissiona): sistemi commissionati a un fornitore di software da uno **specifico cliente**.

La differenza sostanziale è che la **software house** controlla le specifiche solo nei prodotti generici, per i prodotti personalizzati tipicamente sono controllate da chi commissiona il software.

L'approccio sistematico che viene utilizzato nell'ingegneria del software è detto **processo software**. Un **processo software** è un insieme di attività che porta alla creazione di un prodotto software. Ogni processo software ha **quattro attività fondamentali**:

1. **Specifiche del software**: clienti e ingegneri definiscono le funzionalità e i vincoli operativi del software da produrre;
2. **Sviluppo del software**: viene progettato e pianificato e implementato il software;
3. **Convalida del software**: viene convalidato il software per garantire le aspettative del cliente;
4. **Evoluzione del software**: il software viene modificato e mantenuto per continuare a soddisfare i requisiti del cliente sul mercato anche durante la fase di produzione.

Ciascun tipo di sistema software richiede tecniche di **ingegneria del software specializzate**, in quanto ciascun software ha caratteristiche diverse. Tuttavia, esistono dei **concetti fondamentali dell'ingegneria del software** che si applicano a tutti i sistemi software.

- I sistemi devono essere sviluppati utilizzando un **processo di sviluppo chiaro e accuratamente pianificato**.
- La **fidatezza e le prestazioni** sono importanti per ciascun tipo di sistema.
- È importante capire e gestire la specifica e i requisiti del software. **Occorre sapere cosa vogliono i differenti clienti e gestire le loro aspettative sul prodotto finale**.
- Occorre **utilizzare** con efficacia ed efficienza le **risorse esistenti** (tra cui il software esistente).

Seguono le **caratteristiche essenziali di un software di qualità**, indipendentemente dalla sua tipologia:

- **Accettabilità**: il software deve essere **accettabile per il tipo di utenti per il quale è stato progettato**, deve quindi essere comprensibile, usabile e compatibile con l'utente finale e le sue aspettative e necessità.
- **Fidatezza**: la fidatezza include varie caratteristiche tra cui **affidabilità, disponibilità, protezione e sicurezza**. Un software fidato non dovrebbe causare danni fisici o economici nel caso che si verifichi un malfunzionamento.
- **Efficienza**: il software non dovrebbe fare un uso dispendioso delle risorse del sistema, come la memoria o potenza di calcolo della CPU.
- **Manutenibilità**: il software dovrebbe essere scritto in modo da **potersi evolvere facilmente** alle nuove richieste dei clienti.

L'ingegneria dei SISTEMI comprende: HW, SW e PERSONE.

L'ingegneria del SW e' un suo sottoramo.

4

Un SISTEMA e' un collezione di componenti che lavorano insieme per un obiettivo comune.

Un SISTEMA LEGACY e' un sistema sociotecnico sviluppato in passato impiegando tecnologie che col tempo sono diventate vecchie. Modificare una parte di questi sistemi richiede la modifica di altri componenti

Proprieta' EMERGENTI: non vengono fuori guardando i singoli componenti del sistema, bensì l'insieme complessivo (es. il peso dell'intero sistema)
Possono essere di due tipi: - FUNZIONALI: relazione tra input-output, cosa fa il mio sistema (senza guardare cosa succede dentro)
- NON FUNZIONALI: le modalità in cui il sistema è stato costruito (cosa fa al suo interno)

2. Processi software

Un processo software è un insieme di attività che porta alla creazione di un prodotto software. Non esiste un processo software universalmente applicabile. Il processo software utilizzato dipende dall'azienda produttrice e dal tipo di software da sviluppare.

Tuttavia, tutti i processi software devono includere in qualche maniera le quattro attività fondamentali dell'ingegneria del software citate nel capitolo precedente:

1. Specifica del software
2. Sviluppo del software
3. Convalida del software
4. Evoluzione del software

Vedremo ora i processi guidati da piani e i processi agili, le due principali categorie di processi software:

- I processi guidati da piani (plan-driven software processes) sono processi dove tutte le attività sono pianificate in anticipo e il loro avanzamento è misurato rispetto a quanto previsto dal piano.
- I processi agili sono caratterizzati da una pianificazione incrementale (a fasi) che continua durante il corso di sviluppo del software. Sono processi di natura iterativa.

Classificazione dei processi software (standard ISO/IEC 12207)

I processi software possono essere classificati nelle seguenti tre categorie:

1. **Processi primari**: attività necessarie a specificare, progettare, produrre, mantenere ed estendere il prodotto software. (Analisi requisiti, specifiche, design, implementazione, Testing, mantenimento)
2. **Processi di supporto**: vengono eseguiti parallelamente ai processi primari al fine di supportarli e garantire la qualità ed il successo del progetto.
3. **Processi di gestione**: attività necessarie a gestire il progetto di sviluppo del prodotto software.

Modelli dei processi software

Un modello di processo software (anche detto SDLC – Software Development Life Cycle) è una rappresentazione semplificata di un processo software che specifica come sono definiti e organizzati i vari task coinvolti nella progettazione, costruzione e manutenzione di un prodotto. Più precisamente è uno schema di riferimento che specifica in modo astratto e generale quali task svolgere e quando svolgerli. Ogni modello rappresenta un processo da una particolare prospettiva, e quindi fornisce solo delle informazioni parziali su di esso. Questi modelli sono descrizioni astratte di alto livello dei processi software, che possono essere utilizzate per spiegare i diversi approcci allo sviluppo del software.

I modelli generici più utilizzati sono: Tutti i modelli utilizzano come fasi i PROCESSI PRIMARI

1. Modello a cascata
2. Sviluppo incrementale
3. Integrazione e configurazione
4. Sviluppo evolutivo (prototyping)
5. Sviluppo formale
6. Extreme Programming: sviluppo e consegna (costanti) di piccoli incrementi
7. Spiral: approccio generale (più che ciclo di vita), si parte dalle attività che hanno livello di rischio più alto. Un processo è rappresentato da una spirale e ogni ciclo nella spirale rappresenta una fase nel processo

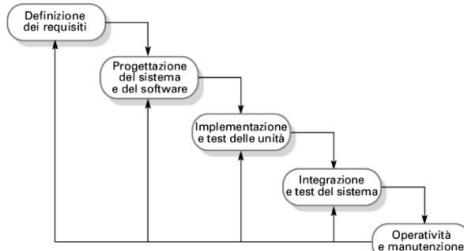
Si noti che i sottosistemi all'interno di un sistema possono essere sviluppati utilizzando modelli diversi. Ad esempio, le sotto-parti ben chiare possono essere sviluppate con un approccio plan-driven (modello a cascata ad es.), mentre quelle più generali, ancora ambigue in fase di progetto, possono essere sviluppate con un approccio incrementale (approccio agile).

Modello a cascata (waterfall model)

In questo modello il processo di sviluppo del software è costituito da un certo numero di stadi. A causa del susseguirsi di una fase dopo l'altra, il modello è conosciuto come modello a cascata. Questo è un esempio di processo software plan-driven. Almeno in linea di principio occorre pianificare tutte le attività di processo prima di iniziare lo sviluppo del software.

I principali stadi del modello a cascata riflettono direttamente le attività di sviluppo del software fondamentali:

1. Analisi e definizione dei requisiti + documento specifiche
2. Progettazione del sistema e del software
3. Implementazione e test delle unità
4. Integrazione del software e test del sistema
5. Operatività e manutenzione



In teoria il risultato di ogni stadio nel modello a cascata è costituito da uno o più documenti approvati, necessari affinché la fase successiva possa iniziare (processo document-driven). In pratica il processo software non è mai un modello lineare semplice, ma richiede una sequenza di feedback da uno stadio all'altro. Nella pratica il cliente chiede le modifiche a cose che sono cambiate e quindi i CAMBIAMENTI possono avvenire IN OGNI FASE

Il fatto di dover stabilire inizialmente gli obiettivi e di dover rilavorare il sistema quando vengono apportate modifiche rende il modello a cascata appropriato solo per alcuni tipi di sistemi software:

- Sistemi integrati dove il software deve interfacciarsi con l'hardware tipicamente meno flessibile.
- Sistemi critici dove sicurezza e protezione sono requisiti chiave.
- Grandi sistemi software che fanno parte di sistemi più complessi sviluppati da più società.

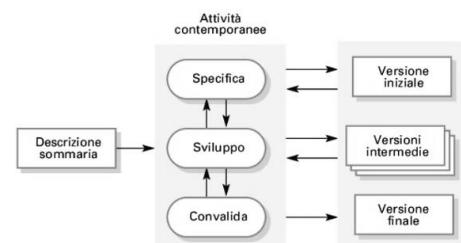
Il modello a cascata NON è appropriato in quei casi in cui sia consentita una comunicazione informale nei team e quando i requisiti del software non sono molto chiari e cambiano rapidamente.

Sviluppo incrementale

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, finché non si ottiene il sistema richiesto. Le attività di specifica, sviluppo e convalida sono intrecciate anziché separate, con molti feedback veloci tra le varie attività. Questo approccio può essere plan-driven, agile, o una combinazione dei due. In un approccio plan-driven gli incrementi sono pianificati precedentemente, nell'approccio agile invece dipendono dalla situazione generale di sviluppo. Questo approccio è pensato per quei software in cui i requisiti e le specifiche di sistema variano costantemente. Ciascun incremento (versione) del sistema apporta qualche funzionalità aggiuntiva richiesta dall'utente. Il modello di sviluppo incrementale può essere visto come una combinazione tra il modello waterfall e il modello evolutivo (prototyping).

Lo sviluppo incrementale offre tre grandi vantaggi:

1. Costo dell'implementazione e delle modifiche dei requisiti ridotto.
2. Maggiore feedback da parte dell'utente.
3. Possibilità di consegnare in anticipo una versione non completa ma funzionante.



Dal punto di vista gestionale, l'approccio incrementale ha due problemi:

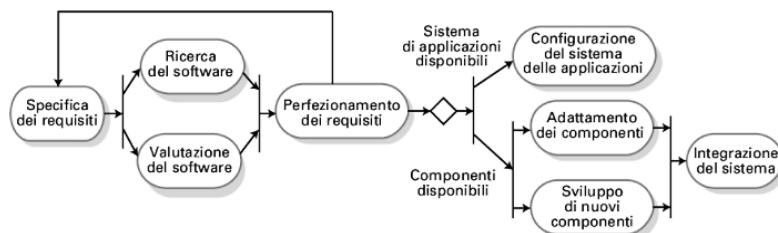
1. Il processo e lo stadio di avanzamento non sono facilmente visibili e tracciabili dai manager.
2. La struttura tende a degradarsi quando vengono aggiunti nuovi incrementi (a causa della mancanza di pianificazione ben strutturata). È necessario attuare periodicamente ristrutturazioni del software come il refactoring.

Esiste una variante del modello incrementale chiamata **sviluppo iterativo**: consiste nel partire con un sistema completo e poi iterare miglioramenti e aggiunte di funzionalità, invece che svilupparle e aggiungerle una ad una come nel modello incrementale.

Integrazione e configurazione (modelli orientati al riuso)

Nella maggior parte dei progetti software si è soliti riutilizzare software. Tre tipologie di componenti/software sono frequentemente riutilizzate/i:

- Sistemi di applicazioni indipendenti che sono configurabili per essere utilizzati in un particolare ambiente.
- Collezioni di oggetti, componenti, API etc.
- Servizi web sviluppati in conformità con gli standard.



In questo approccio le fasi di sviluppo sono:

1. Specifica dei requisiti;
2. Ricerca e valutazione del software riusabile;
3. Perfezionamento e adattamento dei requisiti al software trovato;
4. Configurazione del sistema delle applicazioni (se è stato trovato un sistema pronto all'uso);
5. Adattamento e integrazione dei componenti trovati (se non è stato trovato un sistema pronto all'uso ma esistono componenti riusabili);

L'ingegneria del software orientata al riuso, configurazione e integrazione ha l'ovvio vantaggio di abbattere i costi di sviluppo del software (meno software da sviluppare) e portando a consegne più rapide. Sono però inevitabili dei compromessi nei requisiti per adattare il sistema.

Sviluppo evolutivo (prototyping) → -EVOLUTIONARY -THROW AWAY

Questo approccio è noto anche in altri ambiti dell'ingegneria e si adotta spesso per problemi difficili e di natura nuova. È caratterizzato da una fase iniziale in cui si costruiscono e si valutano diverse versioni successive del sistema a bassa fedeltà e basso costo (prototipi) al fine di avere più informazioni e idee sul risultato finale e sul come ottenerlo. Lo sviluppo evolutivo è quindi un modello di sviluppo software di natura iterativa in cui specifica dei requisiti, validazione e sviluppo si intrecciano di continuo per produrre varie versioni del sistema. I vantaggi dello sviluppo evolutivo sono bassi costi di implementazione al variare dei requisiti e un elevato livello di feedback da parte degli stakeholder. Gli svantaggi invece sono una scarsa visibilità dello stato di sviluppo etc. Applicabile per piccoli e innovativi progetti.

Modelli formali

Un modello di sviluppo formale si basa sull'idea di trasformare una specifica matematica in una versione eseguibile del software. Questo processo è incrementale con piccole trasformazioni che vengono matematicamente dimostrate essere corrette. È un'idea molto complessa da attuare se non di fatto a volte impossibile. È tipica di sistemi critici, dove la fidatezza è un parametro fondamentale (software di pilotaggio ad esempio).

Tutte le fasi del ciclo di vita del SW sono assistite dai CASE TOOL, specifici tool informatici che aiutano a svolgere varie attività

A differenza dell'incrementale, i prototipi che sviluppo mano a mano alla fine li butto e mi servo per trovare solo i requisiti

Nell'Ev: evolvo con l'utente (partendo dai requisiti chiari) fino al sistema finale usando i prototipi

Nel Th: mi concentro solo sui requisiti non chiari e cerco di chiarirli con prototipi.

Attività di processo

Vediamo ora più in dettaglio le quattro attività fondamentali dello sviluppo software:

1. **Specifiche del software**
2. **Sviluppo del software**
3. **Convalida del software**
4. **Evoluzione del software**

Specifiche del software

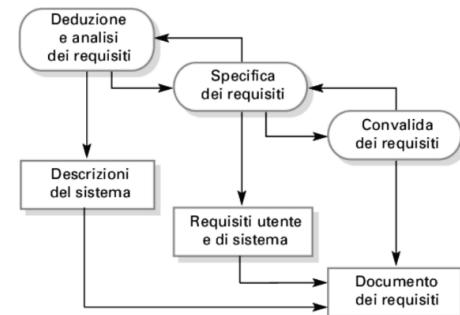
La creazione delle specifiche del software, o dell'**ingegneria dei requisiti** è il processo che mira a capire e definire quali servizi sono richiesti dal sistema, e per identificare i vincoli all'operatività e allo sviluppo del sistema. L'ingegneria dei requisiti è uno stadio particolarmente critico del processo software, poiché gli errori in questa fase portano inevitabilmente a problemi successivi nella progettazione e implementazione del sistema. Il processo di ingegneria dei requisiti porta alla produzione di un documento di caratteristiche concordate che specifica un sistema che soddisfa i requisiti degli **stakeholder**. I requisiti di solito sono presentati in due livelli di dettaglio: gli utenti finali e i clienti hanno bisogno di una formulazione in linguaggio naturale, mentre gli sviluppatori vogliono avere una specifica più dettagliata.

Stakeholder identifica ogni persona o gruppo di persone che sarà influenzato dal sistema; sono i cosiddetti portatori di interesse. Possono essere end user, manager, proprietari o esterni

Le fasi principali del processo di ingegneria dei requisiti sono tre:

1. **Deduzione e analisi dei requisiti**
2. **Specifiche dei requisiti**
3. **Convalida dei requisiti**

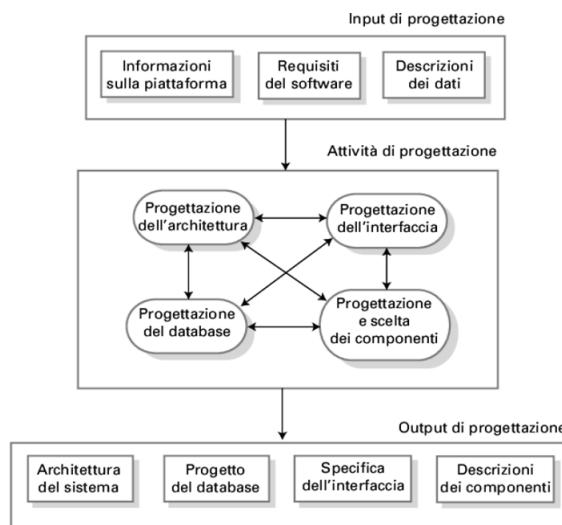
Questi concetti vengono approfonditi nel [capitolo 4](#).



Sviluppo del software (progettazione e implementazione)

Lo stadio di sviluppo software include la progettazione e l'implementazione ed è il processo di conversione delle specifiche in un sistema eseguibile e consegnabile al cliente. Include processi di progettazione e programmazione ma non sempre: se viene adottato un approccio agile le due fasi risultano intrecciate, e non producono documenti formali di progettazione durante il processo.

Questi concetti vengono approfonditi nel [capitolo 3](#) (sviluppo agile), [capitolo 6](#) e [capitolo 7](#).

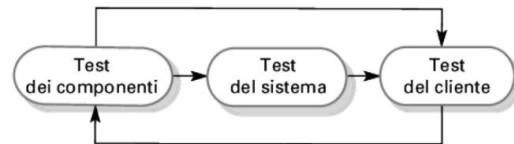


Convalida del software

La convalida del software o, più genericamente, la **verifica e convalida (V&V, Verification & Validation)** è intesa a mostrare che un sistema è conforme alle sue specifiche e soddisfa le aspettative del cliente. Il test dei programmi, dove il sistema viene eseguito utilizzando dati di prova simulati, è la tecnica principale di convalida del software. La convalida può richiedere anche processi di controllo, come le ispezioni e le revisioni, a ogni stadio del processo software, dalla definizione dei requisiti dell'utente allo sviluppo del programma. I sistemi non dovrebbero essere testati come un'unica entità monolitica.

Gli **stadi del processo di test** sono principalmente tre:

1. **Test dei componenti** (o delle unità)
2. **Test del sistema**
3. **Test del cliente**



Questi concetti vengono approfonditi nel [capitolo 8](#).

Evoluzione del software

I grandi software di solito hanno una lunga vita, durante la loro vita devono cambiare se vogliono restare utili. I sistemi software quindi si adattano e si evolvono durante il loro ciclo di vita, dal rilascio della prima release fino all'ultima versione.

Questi concetti vengono approfonditi nel [capitolo 9](#).

Project Management

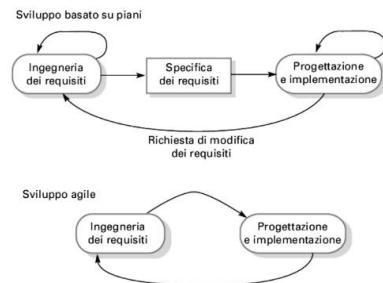
La **gestione del progetto (project management)** del software è una disciplina in cui i progetti vengono pianificati, monitorati e controllati.

Questi concetti vengono approfonditi nel [capitolo 19](#)

3. Sviluppo agile del software

Oggi la **rapidità dello sviluppo e della consegna sono i requisiti più critici della maggior parte dei sistemi software aziendali**. Poiché queste aziende operano in un ambiente in continua evoluzione, **spesso è praticamente impossibile ottenere un insieme completo di requisiti stabili**. I requisiti cambiano perché per il cliente è impossibile prevedere come un sistema influenzerà le pratiche operative, come interagirà con gli altri sistemi e quali operazioni aziendali andrà a impattare.

I **processi di sviluppo software plan-driven che si basano sulla completa specifica dei requisiti non sono adatti allo sviluppo rapido del software**, poiché se i requisiti cambiano o creano problemi, la progettazione e documentazione del sistema deve essere eseguita e verificata nuovamente.



Lo **sviluppo rapido del software** acquisì notorietà come **"sviluppo agile"** e con **"metodi agili"**; questi metodi sono **pensati per produrre rapidamente software utile**. Tutti i metodi agili che sono stati proposti hanno alcune caratteristiche in comune:

- I **processi di specifica, progettazione e implementazione sono intrecciati**. Non c'è una specifica di sistema dettagliata, e la **documentazione dei progetti è minimale**.
- Il sistema viene sviluppato in una serie di incrementi (**sviluppo incrementale**). Gli utenti finali e gli stakeholder sono coinvolti nella specifica e valutazione di ogni incremento.
- Una ricca gamma di strumenti supporta lo sviluppo agile: **test automatici**, strumenti per la gestione della configurazione, strumenti per l'integrazione del sistema, produzione automatica di interfacce utente etc.

I metodi agili sono metodi dello sviluppo incrementale. **I clienti vengono coinvolti nel processo di sviluppo per raccogliere feedback di ogni piccolo incremento. La documentazione è ridotta al minimo**, ricorrendo a comunicazioni informali. **Gli approcci agili considerano la progettazione e l'implementazione attività centrali nel processo software**. In un approccio agile requisiti e progettazione vengono sviluppati assieme, anziché separatamente come nei processi plan-driven.

Metodi agili

I metodi agili possono essere rappresentati dal manifesto dello sviluppo agile del software:

<< Stiamo scoprendo modi migliori di creare software, sviluppando e aiutando gli altri a fare lo stesso. Grazie a questa attività siamo arrivati a considerare importanti:

*Gli individui e le interazioni più che processi e strumenti
Il software funzionante più che documentazione esaustiva
La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano*

*Ovvero, fermo restando il valore delle voci a destra,
consideriamo più importanti quelle a sinistra >>*

Tutti i metodi agili suggeriscono che il software deve essere sviluppato e consegnato in modo incrementale. Questi metodi si basano su processi agili differenti, ma condividono alcuni principi ispirati al manifesto e dunque hanno molto in comune. I **metodi agili sono stati particolarmente utili per due tipi di sviluppo di sistemi**:

- **Software di piccole o medie dimensioni**;
- Sviluppo personalizzato all'interno di un'organizzazione dove c'è un chiaro impegno del cliente di essere coinvolto nel processo di sviluppo.

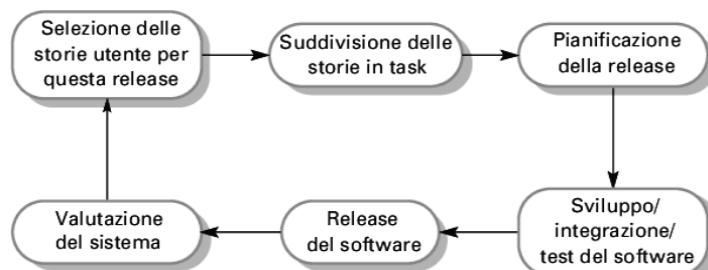
Tecniche di sviluppo agile

La **programmazione estrema** (**eXtreme Programming**) è forse l'approccio più significativo all'innovazione dello sviluppo software agile. Nella programmazione estrema, i **requisiti sono espressi come scenari** (o **storie utente**), che sono implementati direttamente come una serie di **compiti** (**task**). I **programmatori lavorano a coppie** e **sviluppano test** per ogni compito prima di scrivere **il codice**. Tutti i test devono essere completati con successo prima di integrare il nuovo codice nel sistema. Nella pratica reale, la programmazione estrema come proposta non è facilmente applicabile. Per questo, le aziende che adottano i metodi agili selezionano solo alcune pratiche XP.

Le pratiche e metodologie XP più note sono:

- **Proprietà collettiva**: ogni sviluppatore è responsabile di tutto il codice; chiunque può cambiare qualsiasi cosa quando necessario.
- **Integrazione continua**: appena un task è concluso (implementato), viene integrato nel sistema.
- **Pianificazione incrementale**: viene seguita una pianificazione dei task da implementare in una determinata release.
- **Cliente on-site**: un rappresentante dell'utente finale del sistema (il cliente) è sempre presente/disponibile ai programmati per fornire feedback.
- **Programmazione a coppie**: gli sviluppatori lavorano a coppie, verificano reciprocamente il lavoro svolto.
- **Refactoring**: pulizia e miglioramento continuo del codice per aumentarne la manutenibilità.
- **Progettazione semplice**: deve essere svolta una progettazione volta a soddisfare i requisiti correnti, niente di più.
- **Piccole release**: release piccole e frequenti.
- **Ritmo sostenibile**: non sono accettati grandi ritardi, poiché l'effetto finale è una minore qualità del codice e della produttività.
- **Sviluppo con test iniziali**: viene utilizzato un ambiente automatico di test delle unità per provare una nuova parte di funzionalità.

Ciclo di rilascio dell'approccio XP:



Segue una descrizione delle più importanti tecniche/pratiche introdotte dalla XP:

Storie utente

I requisiti del software cambiano sempre. Per gestire questi cambiamenti, i metodi agili non hanno un'apposita attività di ingegneria dei requisiti, ma integrano la deduzione dei requisiti con lo sviluppo. Per semplificare questo approccio, fu sviluppata l'idea delle **storie utente**, dove **una storia è uno scenario d'uso** in cui potrebbe trovarsi un utente finale del sistema.

Le storie utente possono essere incluse nella pianificazione delle iterazioni del sistema. Una volta sviluppate le **story card** (rappresentazioni concrete delle storie utente), il team di sviluppo le suddivide in singoli task e stima le risorse e gli sforzi necessari per implementare ciascun task individuato. Ovviamente i requisiti variano e quindi le storie utente possono essere scartate o modificate nel tempo.

Refactoring

Un precezzo fondamentale dell'ingegneria del software è che si dovrebbe sempre progettare per il cambiamento; questo significa che dovremmo essere in grado di prevedere i cambiamenti futuri dei progetti e del software, in modo che tali cambiamenti possano essere facilmente implementati. L'XP però ha scartato questo principio, considerando che la progettazione per il cambiamento spesso è uno sforzo inutile. Ovviamente, in pratica, le modifiche dovranno essere sempre apportate al codice che si sta sviluppando. Per semplificare queste modifiche, gli sviluppatori XP suggeriscono che **il codice che si sta sviluppando debba essere costantemente rifattorizzato (refactoring)**. Il refactoring **richiede che il team di programmazione ricerchi possibili miglioramenti del software e che li implementi immediatamente**. Il refactoring **migliora la struttura e la leggibilità del software**, evitando così il deterioramento strutturale che si verifica naturalmente quando si modifica il software. Esempi di refactoring sono la riorganizzazione della gerarchia delle classi per eliminare codice duplicato, riordino e ridenominazione degli attributi e dei metodi, sostituzione di sezioni di codice simili con chiamate ai metodi definiti in una libreria etc.

Sviluppo con test iniziali

Come detto all'inizio di questo capitolo, **una delle più importanti differenze tra lo sviluppo incrementale e lo sviluppo agile è come il software viene testato**. La XP ha sviluppato un approccio al test dei programmi per superare alcune difficoltà dei test senza specifica. I **test sono automatizzati e sono centrali nel processo di sviluppo**, e lo sviluppo non può procedere finché tutti i test non sono stati superati con successo. **Anziché scrivere il codice e poi i test per il codice, l'XP prevede il contrario. Questo significa che possono essere eseguiti i test durante l'implementazione del programma**.

Nello sviluppo con test iniziali, chi implementa i task deve capire bene le specifiche, in modo che possa prima scrivere i test per il sistema. Ciò significa che le ambiguità e le omissioni nelle specifiche devono essere chiarite prima che inizi l'implementazione. L'automazione dei test è essenziale per lo sviluppo con test iniziali. È tuttavia difficile giudicare la completezza di un insieme di test.

Si veda lo sviluppo guidato da test approfondito [qui](#).

Programmazione a coppie

Una pratica introdotta dall'XP consiste nel fatto che **i programmatore operino in coppia per sviluppare il codice**. Questo porta a una serie di vantaggi:

1. Supporta l'idea della proprietà e della responsabilità comune del sistema.
2. Funge da processo di revisione informale, in quanto ogni linea di codice è controllata da due persone.
3. Incentiva il refactoring per migliorare la struttura del software.

Gestione agile della progettazione

In qualsiasi società software, i manager hanno bisogno di sapere che cosa sta accadendo, se un progetto potrà raggiungere i suoi obiettivi e se il software sarà consegnato in tempo ed entro il budget previsto. Gli approcci allo sviluppo guidati da piani si sono evoluti per supportare queste esigenze. La pianificazione informale e il controllo del progetto che furono proposti dai metodi agili si sono scontrati con questa esigenza di visibilità da parte dei manager.

Scalabilità dei metodi agili

I metodi agili furono sviluppati per piccoli team di programmazione che avrebbero potuto lavorare insieme nella stessa stanza e comunicare in modo informale. Le piccole società senza processi formali o burocrazia furono i primi utilizzatori di questi metodi. Ovviamente l'esigenza di creare software più adatto alle necessità dei clienti e di doverlo consegnare più velocemente vale anche per i sistemi e le società più grandi.

La scalabilità dei metodi agili ha due sfaccettature:

- Potenziare questi metodi per poter gestire lo sviluppo di grandi sistemi.
- Estendere questi metodi da un uso specializzato dei team di sviluppo a un uso più ampio in una grande società.

Tuttavia, l'uso degli approcci agili per grandi sistemi di lunga durata presenta alcuni problemi:

1. **L'informalità dello sviluppo agile è incompatibile con l'approccio legale alla definizione dei contratti che di solito si usano nelle grandi società;**
2. **I metodi agili sono più indicati per lo sviluppo di nuovo software, non per la manutenzione;**
3. **I metodi agili sono ideati per piccoli team vicini.**

L'uso di metodi agili non è compatibile con la manutenzione a causa di questi problemi:

- **Mancanza di documentazione dettagliata e aggiornata del prodotto.**
- **Mantenere il coinvolgimento dell'utente dopo la release non è facile.**
- **Continuità nel team di sviluppo dopo la release.**

La documentazione formale rende più comprensibile il sistema a chi si trova a modificarlo. Nei metodi agili però questa raramente viene aggiornata.

Metodi agili guidati da piani

Una condizione fondamentale per la scalabilità dei metodi agili è la loro integrazione con gli approcci guidati da piani. Piccole società di startup possono lavorare con una pianificazione informale a breve termine, ma le società più grandi devono avere piani a lungo termine e budget per gli investimenti, lo staff e lo sviluppo. Per risolvere i problemi dei metodi agili per grandi sistemi, molti dei più grandi progetti di sviluppo agile combinano le pratiche tipiche degli approcci plan-driven. Alcuni sono più agili, altri più plan-driven, ma entrambi con caratteristiche dell'altra categoria. Per decidere come bilanciare l'approccio da un lato o dall'altro occorre rispondere ad alcune domande che riguardano il sistema da sviluppare, il team di sviluppo e le risorse disponibili, domande come:

- Quanto è grande il sistema?
- Che tipo di sistema si sta sviluppando?
- Qual è la vita prevista del sistema?
- Il sistema è soggetto a norme/vincoli esterni?

Le risposte a queste domande dovrebbero guidare la scelta di un metodo più o meno agile/plan-driven. I metodi agili pongono sul team di sviluppo una grande responsabilità per cooperare e comunicare durante le fasi di sviluppo. In realtà però qualche pianificazione potrebbe essere richiesta per organizzare nel modo migliore il lavoro delle persone.

- Approfondimento:
https://it.wikipedia.org/wiki/Metodologia_agile
<https://www.agilealliance.org/agile101/>

I requisiti sono rivolti a più destinatari, ci sono destinatari NON TECNICI, MEDI, ESPERTI.

Alcuni metodi AGILI dicono che produrre documenti dei requisiti di sistema articolati non convenga in quanto i requisiti cambiano molto frequentemente -> i metodi AGILI usano i REQUISITI INCREMENTALI

Dagli OBIETTIVI (formula sintetica e astratta degli scopi e intenzioni del cliente)
ai REQUISITI (derivano dagli obiettivi e sono di maggior dettaglio e tradotti in modo verificabile)
alle SPECIFICHE (derivano dai requisiti, hanno ulteriore dettaglio)

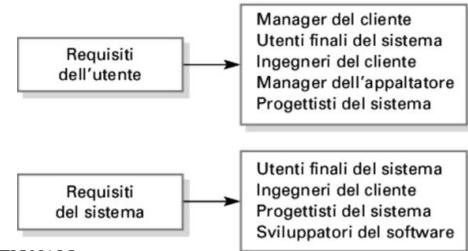
Andrea Mansi UNIUD 2019-2020
Riassunto concetti chiave Ingegneria del Software (ver. 1.1 - 5/2/2020)

4. Ingegneria dei requisiti

I requisiti di un sistema sono la descrizione dei servizi che il sistema deve fornire e dei suoi vincoli operativi. Il processo di ricerca, analisi, documentazione e verifica di questi servizi e vincoli è chiamato **ingegneria dei requisiti** (RE, Requirements Engineering).

I requisiti dell'utente indicano requisiti astratti di alto livello, i requisiti del sistema danno la descrizione dettagliata di quello che il sistema dovrebbe fare, le specifiche del software sono dettagli tecnici da utilizzare per lo sviluppo e progettazione del software.

- **Requisiti dell'utente**: dichiarano nel linguaggio naturale e corredati da diagrammi, quali servizi il sistema dovrebbe fornire e i vincoli sotto cui deve operare.
- **Requisiti di sistema**: sono descrizioni più dettagliate delle funzioni, dei servizi e dei vincoli operativi del sistema software, a volte chiamati **specifiche funzionali**, dovrebbero definire esattamente che cosa deve essere implementato. si dividono in **FUNZIONALI** e **NON FUNZIONALI**
- **Specifiche software**: sono descrizioni molto dettagliate del software che vengono usate come base per la fase di progettazione e implementazione del software.



Diversi livelli di specifiche del sistema sono utili, perché comunicano le informazioni che riguardano il sistema a diversi tipi di utilizzatori dei documenti. Un requisito utente può essere espanso in diversi requisiti del sistema.

Il requisito dell'utente è generico. I requisiti del sistema forniscono informazioni specifiche sui servizi. Le specifiche software forniscono descrizioni molto dettagliate sul software a livello tecnico.

L'ingegneria dei requisiti è di solito presentata come la prima fase del processo di ingegneria del software. Tuttavia, occorre capire un po' meglio i requisiti del sistema prima di prendere una decisione per procedere con l'acquisizione o lo sviluppo di un sistema, di conseguenza è una fase iterativa. Questa fase iniziale dell'ingegneria dei requisiti fornisce una vista generale di ciò che il sistema dovrebbe fare e dei benefici che dovrebbe apportare.

Requisiti funzionali e non funzionali

I requisiti dei sistemi software sono spesso divisi in requisiti **funzionali** e **non funzionali**:

1. **Requisiti funzionali**: sono definizioni di servizi che il sistema deve fornire; indicano come il sistema dovrebbe reagire a particolari input e come dovrebbe comportarsi in particolari situazioni. In alcuni casi possono stabilire esplicitamente che cosa il sistema non dovrebbe fare.
2. **Requisiti non funzionali**: sono vincoli sulle funzioni o servizi offerti dal sistema. Includono vincoli temporali e sul processo di sviluppo e vincoli imposti dagli standard. I requisiti non funzionali di solito si applicano al sistema completo, non a singole funzioni o servizi. Possono essere vincoli di utilizzo memoria, vincoli legati a standard, usabilità, di processo etc.
3. Requisiti di DOMINIO: riguardano il dominio applicativo dell'utente

Esempio: un requisito dell'utente riguardante la sicurezza, come una definizione che limita l'accesso agli utenti autorizzati, potrebbe sembrare un requisito non funzionale. Tuttavia, quando questo requisito viene sviluppato nei dettagli, può produrre altri requisiti che sono chiaramente funzionali come, per esempio, la necessità di includere un sistema di autenticazione degli utenti.

Questo esempio dimostra che i requisiti non sono indipendenti e che un requisito spesso genera o limita altri requisiti. I requisiti del sistema pertanto non specificano semplicemente le funzioni o i servizi che servono al sistema, ma anche le funzionalità necessarie per garantire che questi servizi/funzioni siano realizzati in modo efficace.

Segue una descrizione più dettagliata dei requisiti funzionali e non funzionali:

Regola delle 3C per i REQUISITI (misurano il concetto di qualità):
CORRETTI (devono esprimere le richieste dei committenti),
COMPLETI (includere tutte le richieste)
CONSISTENTI (non devono esserci conflitti o contraddizioni tra i requisiti)

Andrea Mansi UNIUD 2019-2020
Riassunto concetti chiave Ingegneria del Software (ver. 1.1 - 5/2/2020)

Requisiti funzionali

I **requisiti funzionali** descrivono ciò che il sistema dovrebbe fare. Questi requisiti dipendono dal tipo di software in via di sviluppo, dai futuri utenti e dall'approccio generale adottato dall'organizzazione durante la scrittura dei requisiti. Se espressi come **requisiti dell'utente**, i requisiti funzionali devono essere scritti nel linguaggio naturale, in modo che gli utenti del sistema e i manager possano capirne il significato. I **requisiti funzionali del sistema** estendono i requisiti dell'utente e sono scritti per gli **sviluppatori del sistema stesso**; essi descrivono dettagliatamente le funzioni del sistema, input output e le eccezioni ad essi correlati.

I requisiti funzionali variano da descrizioni generiche che riguardano che cosa il sistema deve fare a definizioni specifiche che rispecchiano i modi locali di lavorare o i sistemi esistenti di un'organizzazione.

Esempi di requisiti funzionali:

1. Un utente deve essere in grado di cercare gli appuntamenti nelle liste di tutte le cliniche.
2. Il sistema deve generare ogni giorno, per ciascuna clinica, la lista degli appuntamenti.

I requisiti funzionali possono essere scritti a diversi livelli di dettaglio. **Ricapitolando**, i **requisiti funzionali indicano tipicamente cosa il sistema deve fare**. In linea di principio le specifiche dei requisiti funzionali di un sistema dovrebbero essere complete e coerenti. In pratica questo non è facile in sistemi grandi, dove diversi stakeholder hanno diverse opinioni, i requisiti non sono semplici etc.

Requisiti non funzionali Se non vengono rispettati posso buttare via l'intero sistema, spesso vengono trasformati in funzionali (anche quelli di dominio) I **requisiti non funzionali**, come suggerisce il nome, sono requisiti che non riguardano direttamente specifici servizi forniti dal sistema. Di solito **specificano o limitano le caratteristiche del sistema nel suo complesso**. Possono riferirsi a proprietà del sistema, come l'affidabilità, i tempi di risposta e l'utilizzo delle risorse. **Possono anche definire vincoli sull'implementazione del sistema**. I requisiti non funzionali sono spesso più critici dei singoli requisiti funzionali. L'implementazione di questi requisiti può interessare il sistema per due ragioni:

- I requisiti non funzionali possono influire sull'intera architettura di un sistema, anziché su singoli componenti.
- Un singolo requisito non funzionale potrebbe generare vari requisiti funzionali tra loro correlati che definiscono nuovi servizi del sistema, che li rendono necessari se il requisito non funzionale deve essere implementato. (vedi ad esempio un requisito non funzionale sulla sicurezza che genera un requisito funzionale sulla necessità di autenticare gli utenti).

I **requisiti non funzionali** derivano dalle necessità degli utenti, dai limiti del budget, dalle politiche organizzative, dalla necessità di interoperabilità con gli altri sistemi software o hardware esistenti, fattori esterni di sicurezza o legislazione.

I **requisiti non funzionali** possono suddividersi in tre sottocategorie:

- **Requisiti del prodotto** Il prodotto finale deve comportarsi in un determinato modo, es. velocità di esecuzione
- **Requisiti organizzativi** Va usato uno specifico linguaggio o delle tecniche e procedure standard per sviluppare il sw
- **Requisiti esterni** Requisiti legislativi esterni al sistema sw

Un **problema dei requisiti non funzionali** è che gli stakeholder propongono i requisiti come **obiettivi principali**, come: "facilità d'uso, capacità di ripristino del sistema, risposta rapida". Questi **obiettivi vaghi causano problemi agli sviluppatori**, perché danno adito alla **libera interpretazione** e alla successiva discussione quando il sistema viene consegnato. Quando è possibile, bisognerebbe descrivere i requisiti non funzionali quantitativamente, in modo che possano essere verificati in maniera oggettiva. È possibile così misurare queste caratteristiche quando il sistema viene testato per controllare se soddisfa o meno i suoi requisiti non funzionali.

I cambiamenti dei requisiti possono avvenire lungo tutto il progetto in qualsiasi fase.

Processi di ingegneria dei requisiti

L'ingegneria dei requisiti è formata da tre attività chiave:

- **Deduzione e analisi**: scoperta dei requisiti attraverso l'interazione con gli stakeholder.
- **Specificazione** (specificazione): conversione dei requisiti in un formato standard (le specifiche).
- **Convalida**: controllo che i requisiti definiscano realmente ciò che vuole il cliente.

Queste attività sono spesso illustrate come attività sequenziali, in realtà l'ingegneria dei requisiti è un processo iterativo in cui queste attività sono interacciate. Le attività sopra citate possono essere viste come un **processo iterativo a spirale**. La quantità di tempo e gli sforzi dedicati a ogni attività in una interazione dipende principalmente dallo stadio generale del processo. L'output del processo di ingegneria dei requisiti è un documento dei requisiti del sistema.

Segue una spiegazione più dettagliata delle tre fasi appena citate:

Deduzione e analisi dei requisiti

L'obiettivo principale del processo di **deduzione dei requisiti** consiste nel capire il lavoro svolto dagli **stakeholder** e come essi potrebbero utilizzare un nuovo sistema a supporto del loro lavoro/attività. Durante la fase di deduzione dei requisiti, gli **stakeholder** descrivono le loro **esigenze**, sulle prestazioni richieste dal sistema, sui vincoli hardware e così via.

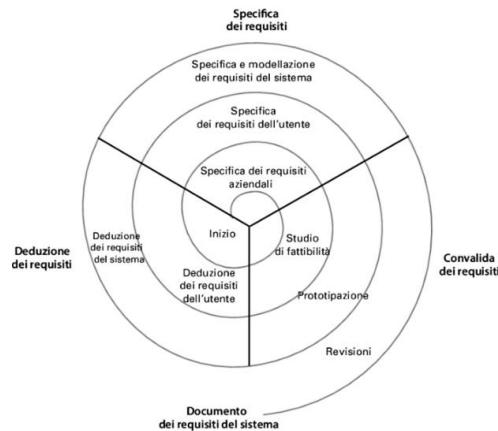
È da notare che spesso **gli stakeholder non sanno cosa vogliono da un sistema informatico oppure non sanno cosa un sistema informatico è in grado di fare o meno**. Possono esserci difficoltà nell'esprimere i requisiti, che gli stakeholder esprimono in termini e con conoscenza implicita del loro lavoro e che gli ingegneri dei requisiti non sempre comprendono.

Le attività del processo di deduzione dei requisiti sono le seguenti:

1. **Scoperta e comprensione dei requisiti** (elicitazione dei requisiti)
2. **Classificazione e organizzazione dei requisiti**
3. **Negoziazione e priorità dei requisiti**
4. **Documentazione dei requisiti**

Esistono più **tecniche di deduzione dei requisiti**: quelle fondamentali sono:

- **Interviste**, ovvero **parlare con gli stakeholder**: le interviste formali o informali con gli stakeholder del sistema sono parte della gran parte dei processi di ingegneria dei requisiti. Le interviste possono essere principalmente di due tipi:
 - o **Chiuse**: lo stakeholder risponde a un **insieme predefinito di domande**;
 - o **Aperte**: dove **non c'è niente di predefinito**, stakeholder e ingegneri analizzano il problema e deducono informazioni/necessità.
- **Etnografia**, **osservare le persone mentre svolgono il loro lavoro**: i sistemi software non sono mai isolati, vengono utilizzati in un contesto sociale e organizzativo, e i requisiti del sistema possono essere derivati o vincolati da questo contesto. Sono quindi fondamentali le analisi del contesto d'uso di un software prima di cominciare la progettazione.
- **Storie e scenari**: storie e scenari sono essenzialmente la stessa cosa. Sono **descrizioni di come il sistema può essere utilizzato per svolgere particolari compiti**. Descrivono che cosa fanno le persone, quali informazioni utilizzano e producono, e quali sistemi possono utilizzare durante questo processo. La differenza è nel modo in cui le descrizioni sono strutturate e nel livello di dettaglio.



Le specifiche diventano l'input per la fase di progettazione. —> nel documento delle SPECIFICHE si dice COSA VA FATTO, non COME (quello si fa nel DESIGN)

Per aiutare il linguaggio naturale, nel documento delle specifiche, si possono utilizzare delle tabelle che servono a specificare determinati elementi o alternative.

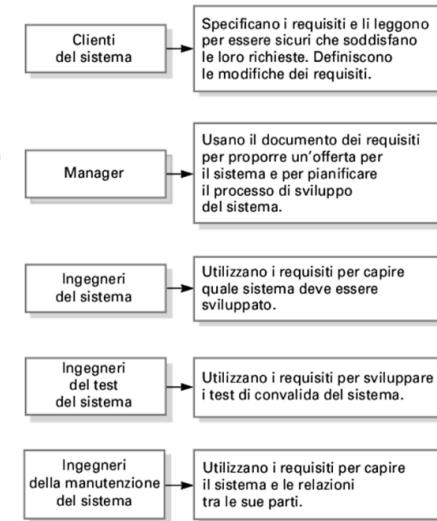
Per visualizzare meglio gli stati e le azioni del sistema si può utilizzare un MODELLO GRAFICO, es. dFd, Reti di Petri..

- **Le storie** sono scritte come [REDACTED] del modo in cui il sistema è utilizzato dall'utente;
- **Gli scenari** di solito sono strutturati con **informazioni specifiche raccolte come input e output**.

Storie e scenari possono essere utilizzati per agevolare la discussione con gli stakeholder che potrebbero avere modi differenti di raggiungere lo stesso risultato.

Specifiche dei requisiti

La **specificazione dei requisiti** è il **processo di scrivere i requisiti dell'utente e del sistema in un documento formale**. Il documento prodotto in questa fase è a volte chiamato **specificazione dei requisiti del software** (o SRS, Software Requirements Specifications) ed è una definizione ufficiale di **quello che gli sviluppatori del sistema dovrebbero implementare**; può includere requisiti dell'utente, requisiti di sistema. Teoricamente, i requisiti dell'utente e del sistema dovrebbero essere chiari, senza ambiguità, facili da capire, completi e coerenti. In pratica questo è impossibile. **Gli stakeholder interpretano i requisiti in modi differenti e spesso i requisiti includono incoerenze e conflitti intrinseci**. I requisiti dell'utente dovrebbero descrivere i requisiti funzionali e non funzionali in modo che siano comprensibili dagli utenti del sistema che non hanno conoscenze tecniche. **Il documento dei requisiti non dovrebbe includere dettagli sull'architettura o sulla progettazione del sistema**. Di conseguenza, non dovrebbero essere presenti gerghi tecnici, notazioni strutturate o formali. **I requisiti del sistema sono versioni espansive dei requisiti dell'utente**; sono utilizzati dagli ingegneri del software come base di partenza per la **progettazione del sistema**; aggiungono dettagli e spiegano come il sistema dovrebbe realizzare i **requisiti dell'utente**. In teoria, i requisiti del sistema dovrebbero descrivere soltanto il comportamento esterno del sistema e i suoi vincoli operativi. Non dovrebbero riguardare la progettazione o l'implementazione del software. Tuttavia, questo non è sempre possibile.



Convalida dei requisiti

La convalida dei requisiti è il **processo che verifica se i requisiti definiscono il sistema realmente voluto dal cliente**. Si sovrappone alla deduzione e all'analisi perché cerca di scoprire i problemi nei requisiti. **La convalida è importante perché gli errori nel documento dei requisiti possono portare a costi di rilavorazione molto alti se tali problemi vengono scoperti durante lo sviluppo o dopo la messa in servizio del sistema**. Per risolvere un problema nei requisiti, di solito è molto più costoso modificare il sistema che correggere gli errori di progettazione o di codifica. Una modifica dei requisiti di solito implica anche che la progettazione e l'implementazione del sistema devono essere corrette; inoltre, occorre eseguire nuovi test sul sistema. Esistono diverse tipologie di controllo dei requisiti da effettuare durante il processo di convalida:

Per valutare i requisiti si usano i parametri delle 3C

- **Controlli di validità**: verificano se i **requisiti riflettono le reali esigenze degli utenti** del sistema.
- **Controlli di consistenza**: tra i **requisiti non dovrebbero esserci conflitti**, ovvero non dovrebbero esserci vincoli in contraddizione.
- **Controlli di completezza**: il **documento dei requisiti non deve ommettere alcuna necessità** degli utenti.
- **Controlli di realismo/fattibilità**: sfruttando le conoscenze delle tecnologie esistenti, si dovrebbero controllare i **requisiti per assicurarsi che possano essere realmente implementati**.
- **Verificabilità**: i **requisiti del sistema dovrebbero essere scritti sempre in modo da essere verificabili**.

È possibile utilizzare diverse tecniche di validazione, singolarmente o combinandole insieme.

- **Revisione dei requisiti**: i requisiti vengono analizzati sistematicamente da un team di revisori, che controllano errori e incoerenze.
- **Prototipazione**: gli utenti finali e i clienti provano un modello eseguibile del sistema per vedere se esso soddisfa le loro necessità aspettative.
- **Generazione di test case**: i requisiti dovrebbero essere testabili.

Segue ora una quarta fase dell'ingegneria dei requisiti: la gestione dei requisiti.

Modifica dei requisiti (gestione dei requisiti)

Durante tutto il ciclo di vita del progetto i requisiti possono cambiare, possono emergere sempre nuovi requisiti

Poiché un sistema non può essere completamente definito, anche i requisiti del software restano indefiniti. Durante il processo di sviluppo del software, la conoscenza di un problema da parte degli stakeholder cambia in continuazione. I requisiti del sistema devono dunque evolversi per riflettere la mutua conoscenza del problema. Durante l'evoluzione dei requisiti, occorre tenere traccia dei singoli requisiti e conservare i collegamenti tra i requisiti dipendenti, in modo che si possa stabilire l'impatto delle modifiche dei requisiti. È dunque importante la pianificazione della **gestione dei requisiti**: ovvero quel processo che gestisce e controlla le modifiche dei requisiti; pianificare la gestione dei requisiti significa stabilire come dovrà essere gestita una possibile serie di cambiamenti dei requisiti. Durante la fase di **pianificazione della gestione dei requisiti**, occorrerà prendere decisioni su vari argomenti:

- **Identificazione dei requisiti**: ogni requisito deve essere univocamente identificato; in modo che possa avere un riferimento incrociato con altri requisiti e possa essere identificato nelle definizioni di tracciabilità.
- **Processo di gestione delle modifiche**: pianificare un insieme di attività che stabilisce l'impatto e il costo delle modifiche;
- **Politiche di tracciabilità**: definiscono le relazioni tra ciascun requisito, e tra i requisiti e il progetto del sistema; Posso rappresentare le relazioni tra requisiti con un matrice della tracciabilità, così è più facile rilevare le varie dipendenze
- **Supporto degli strumenti**.



Esistono due tipi di REQUISITI:

- **STABILI**: derivanti dal fulcro del sistema dello stakeholder, ovvero requisiti che non cambieranno mai (es. in un ospedale ci saranno sempre dottori)
- **VOLATILI**: requisiti che cambiano spesso.

5. Modelli di sistema

Rappresentare in fase di analisi il dominio su cui stiamo per operare e rappresentare le caratteristiche del sistema sw che stiamo per costruire.

La **modellazione dei sistemi** è il **processo che sviluppa modelli astratti di un sistema, dove ogni modello rappresenta una differente vista o prospettiva del sistema**. Oggi per la modellazione dei sistemi di solito s'intende la rappresentazione di un sistema utilizzando qualche tipo di notazione grafica basata sui diagrammi **UML (Unified Modeling Language)**.

Uso dei modelli di sistema:

- i modelli sono **utilizzati durante il processo di ingegneria dei requisiti per facilitare la deduzione dettagliata dei requisiti per un sistema**;
- durante il processo di progettazione **per descrivere il sistema agli ingegneri che lo devono implementare**, e dopo l'implementazione per documentare la struttura e il funzionamento del sistema stesso.

È possibile sviluppare modelli sia di sistemi esistenti sia di nuovi sistemi.

1. I modelli di un sistema esistente si usano durante l'ingegneria dei requisiti. Servono a chiarire che cosa fa il sistema e possono essere utilizzati per focalizzare la discussione degli stakeholder.
2. I modelli di un nuovo sistema si usano durante l'ingegneria dei requisiti per descrivere con maggiore chiarezza i propositi ad altri stakeholder del sistema. Gli ingegneri utilizzano questi modelli per discutere le proposte di progettazione e per la documentazione del sistema.

È importante capire che il modello di un sistema non è una rappresentazione completa del sistema. Il **modello tralascia deliberatamente i dettagli per rendere più comprensibile il sistema**. Un **modello** è **un'astrazione che si sta studiando, non una rappresentazione alternativa del sistema**.

È possibile sviluppare vari modelli per rappresentare il sistema da differenti prospettive; per esempio:

1. **una prospettiva esterna**: viene **modellato il contesto o l'ambiente in cui opera il sistema**;
2. **una prospettiva di interazioni**: vengono modellate le interazioni tra il sistema e il suo ambiente, o tra i componenti del sistema;
3. **una prospettiva strutturale**: viene **modellata l'organizzazione del sistema** o la struttura dei dati elaborati del sistema;
4. **una prospettiva comportamentale**: vengono **modellati il comportamento dinamico del sistema** e le sue risposte agli eventi.

Architettura guida da modelli

L'architettura guidata da modelli è una tecnica che si basa sui modelli per progettare e implementare il software che usa un sottoinsieme di modelli UML per descrivere il sistema. I modelli sono creati a diversi livelli di astrazione. Da un modello di alto livello, indipendente dalla piattaforma, teoricamente è possibile generare un programma funzionante senza interventi manuali.

Il metodo dell'architettura guidata da modelli (**MDA – Model driven architecture**) consiglia di produrre tre tipi di modelli astratti per un sistema:

- Modelli indipendenti dal calcolo (CIM – Computation independent model)
- Modelli indipendenti dalle piattaforme (PIM – Platform independent model)
- Modelli specifici dalle piattaforme (PSM – Platform specific model)

L'ingegneria basata sui modelli consente agli ingegneri di considerare i sistemi con un alto livello di astrazione, senza preoccuparsi dei dettagli della loro implementazione. Questo riduce le probabilità di errore, accelera il processo di progettazione e implementazione, e consente la creazione di modelli di applicazioni riusabili e indipendenti dalle piattaforme. Il concetto fondamentale dell'MDA è che le trasformazioni fra modelli possono essere definite e applicate automaticamente da strumenti software

Tipologie di modelli:

- **CONTESTO**: adottano una visione esterna e si suddividono in:

- *DIAGRAMMI DI CONTESTO*: evidenziano i confini di sistema. Con ENVIRONMENT si definisce tutto ciò che è esterno al SW e non abbiamo controllo su di esso.
- *MODELLI DI PROCESSO*: evidenziano i confini dei processi, ovvero quali processi sono eseguiti dal SW e con quali esterni interagisce).

- **COMPORTAMENTALI**: descrivono come funziona e come si comporta il sistema. Si suddividono in:

- *DATA PROCESSING (DFD)*: Analizza come vengono processati i DATI lungo l'esecuzione del sistema.
=> visualizza i passaggi della trasformazione dei dati.
I processi vanno trattati come black box.
Non confondere DFD con DIAGRAMMI DI FLUSSO, nei dfd sugli archi ci sono DATI mentre nei flow chart ci sono operazioni.
Visione puramente FUNZIONALE.
- *MACCHINA A STATI*: Descrive il comportamento del sistema al verificarsi di determinati eventi.
Si passa da uno stato all'altro. Gli stati sono i NODI e gli eventi sono ARCHI.
Sono limitanti quando il sistema inizia ad avere molteplici configurazioni...come facciamo a sapere in che configuraz. siamo?
- *RETI DI PETRI*: Servono a superare la limitazione delle macchine a stati, riusciamo a rappresentare le configurazioni istantanee.

- **SEMANTICI**: descrivono la semantica dei dati, caratteristiche delle varie entità e rappresentare le relazioni. Si suddividono in :

- *MODELLI E-R*: stabilisce le entità del sistema e le loro relazioni.
- *DATA DICTIONARY*: è un glossario, una lista dei termini utilizzati dentro al sistema.

- **A OGGETTI**: baruzzo

Nei diagrammi di contesto l'ENVIRONMENT (contesto) comprende tutte le entità esterne al sw su cui non abbiamo controllo —> ci deve essere chiaro cosa è dentro e cosa è fuori.

I modelli di contesto vengono utilizzati per evidenziare i confini del mio sw e quali sono le entità esterne

I diagrammi di contesto adattano una visione esterna, servono a mostrare i confini del sw.

I DFD di livello 0 hanno un'entità singola (il sistema sw) e tutto il resto è esterno —> quindi i DFD di liv 0 (i DFD sono modelli comportamentali) sono modelli di contesto.

Andrea Mansi UNIUD 2019-2020

Riassunto concetti chiave Ingegneria del Software (ver. 1.1 - 5/2/2020)

DFD – Data Flow Diagram

Sono modelli comportamentali

I DFD servono a identificare i processi di elaborazione/trasformazione dei dati elaborati in un sistema software. Nei DFD vengono specificati dati in input e dati prodotti in output (ossia flussi in ingresso e flussi in uscita). Un DFD può essere costruito a vari livelli gerarchici: un singolo processo che è caratterizzato da precisi input e output può essere scomposto nei suoi sotto processi, che globalmente avranno gli stessi input e output (eventualmente descritti ad un livello di dettaglio più alto – corrispondenza dei flussi). Questo processo di scomposizione è detto scomposizione funzionale.

Notazione:

- I dati vengono associati ai flussi e devono venir descritti mediante nomi che indichino dati (staticità) e non processi (dinamicità).
- I flussi vengono rappresentati dalle frecce (con estremità "piena"). Vengono anche chiamati pipeline.
- I processi vengono associati alle elissi e devono venir descritti mediante nomi che indichino processi/attività (dinamicità e non staticità).
- Gli archivi di dati possono venir indicati da linee parallele orizzontali.
- Le entità esterne possono venir rappresentate alle estremità delle frecce entranti o uscenti del diagramma, utilizzando rappresentazioni iconiche, rettangoli o semplicemente un nome che le denota.

Notazione base dei DFD

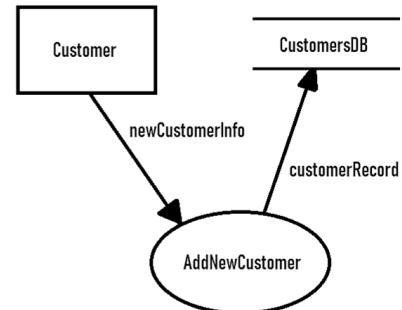
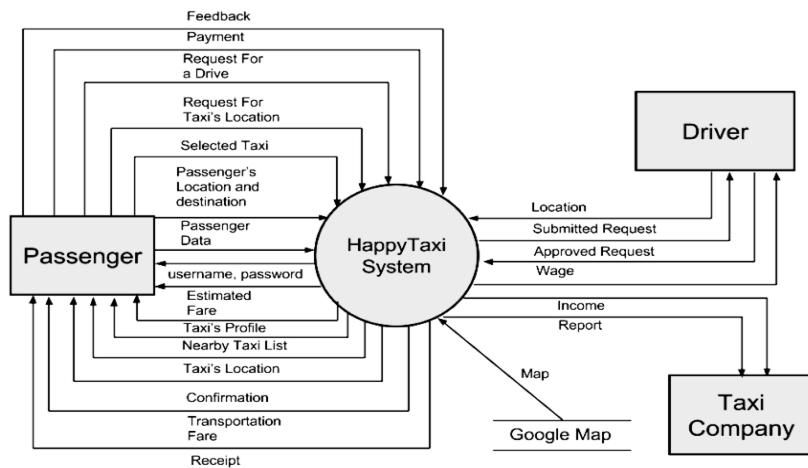


Diagramma di contesto e DFD di livello 0

Un diagramma DFD di livello 0 corrisponde ad un **diagramma di contesto**. Nel livello 0 il sistema è rappresentato con un singolo nodo (processo) e con i flussi di dati che vengono scambiati con unità esterne al sistema. Quindi: **c'è una solta entità** che rappresenta il processo centrale dello scope del sistema. Mostra cosa il sistema riceve e cosa restituisce.

Esempio diagramma di contesto (DFD livello 0):



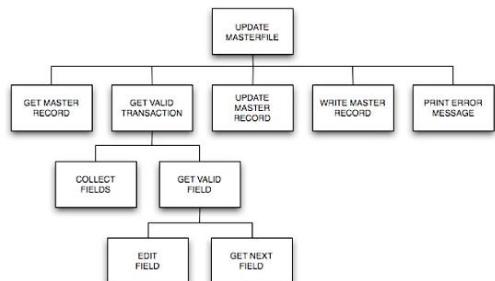
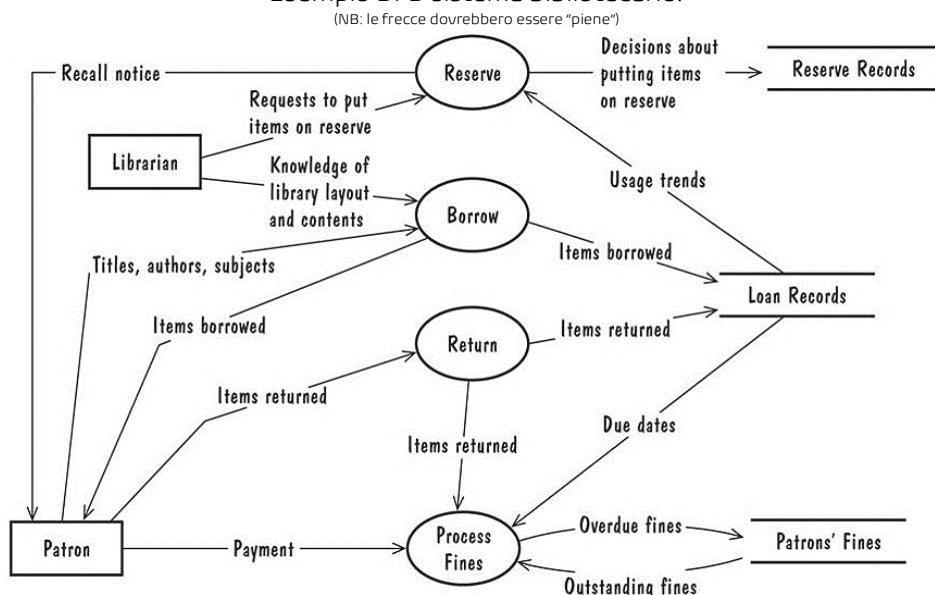
Non vanno confusi i DFD con i Flow Chart. I DFD descrivono le trasformazioni sui dati e il relativo flusso di trasformazione da input a output. I Flow chart (Diagrammi di flusso) invece descrivono il controllo, quali operazioni vengono fatte prima e quali dopo, e quali test eseguire per scegliere fra flussi diversi.

Linee guida

Linee guida per la stesura di un DFD

- **Ignorare** condizioni e operazioni di inizializzazione, terminazione, gestione degli errori. (si pensi quindi alla situazione stabile, non al transitorio, inizializzazione, all'avvio, alla terminazione).
- **Ignorare** il **flusso** (la logica) di controllo e la sincronizzazione delle fasi.
- **Ignorare** i **dati di controllo** (è un dettaglio implementativo; eccessivo).
- **Ignorare** il **funzionamento interno** di un processo (considerarlo una black box) ma modellarne solo gli aspetti funzionali ed i relativi flussi di I/O.
- **VERIFICARE** la coerenza di dati e risultati e viceversa.

Esempio DFD sistema bibliotecario:



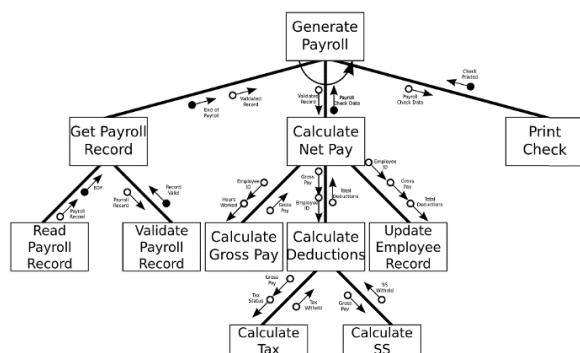
Schema HIPO: Hierarchical Input Process Output

Ogni funzione è rappresentata da un box rettangolare e può essere descritta più a fondo in un diagramma HIPO: ogni funzione è strutturata in sotto funzioni, livello dopo livello. È tipico disporre a sinistra i moduli input, in centro di processing e a destra di output.

Structure chart (SC)

Un diagramma strutturale (SC) è un diagramma che mostra la suddivisione di un sistema ai suoi livelli gestibili più bassi e lo scambio di dati. Ogni modulo è rappresentato da una casella, che contiene il nome del modulo. La struttura ad albero visualizza le relazioni tra i moduli.

Notazione:



Reti di Petri

Le reti di Petri servono a rappresentare la dinamica di un sistema in cui risulta complesso utilizzare gli automi, ed in particolare un unico stato globale (negli automi la situazione specifica in cui si trova l'intero sistema viene rappresentata da un'unica entità astratta detta stato).

Nelle reti di Petri, lo stato viene rappresentato suddiviso/distribuito nelle sue componenti significative, ognuna delle quali è rappresentata da un **posto**. La dinamica del sistema viene descritta con granularità fine, attraverso **passaggi di stato** (transizioni) riferiti solo ad una parte del sistema (singoli posti o insiemi di posti). In tal modo risulta possibile rappresentare più processi indipendenti tra loro, che però possono caratterizzarsi per varie forme di sincronizzazione. Graficamente, una RdP è rappresentabile come un grafo bipartito.

Una rete di Petri (RdP) è composta da quattro elementi fondamentali:

- Un insieme di **posti P**, rappresentati graficamente da un cerchietto.
- Un insieme di **transizioni T**, rappresentati graficamente da una barretta:



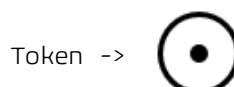
- Una **funzione di input I** ($I:P \rightarrow N$) che associa una collezione di posti (posti di input) ad una transizione. I è rappresentata graficamente da un arco orientato da uno o più posti ad una transizione.



- Una **funzione di output O** ($O:T \rightarrow P$) che mappa una transizione in una collezione di posti chiamati posti di output. O è rappresentata graficamente da un arco orientato da una transizione a uno o più posti.



- Un'altra primitiva delle RdP è il **token** rappresentato graficamente da un puntino (interno ad un posto).



- Una **marcatura** è un assegnamento di token sui posti della rete: zero, uno o più token per ciascun posto della rete. Serve a rappresentare lo stato corrente del sistema.

I posti rappresentano stati parziali; la marcatura ci permette di identificare lo stato corrente del sistema, che risulta quindi distribuito, scomposto in stati parziali. Le funzioni di input e di output associati ad una specifica transizione rappresentano rispettivamente i posti da cui è possibile attivare la transizione ed i posti a cui si transita al completamento della transizione. I token sono

astrazioni interpretabili in diversi modi in base alla specifica situazione: possono identificare la quantità di risorse presenti in un posto, oppure se una condizione (associata al posto) è o meno verificata (in un determinato momento), contatori, il numero di volte in cui un posto è stato usato etc.

L'**evoluzione** di una RDP è condizionata dal numero e dalla distribuzione di token nella rete. L'evoluzione è caratterizzata dallo scatto delle transizioni (che rappresenta un passaggio da uno stato del sistema ad uno stato successivo). Una transizione scatta "consumando" dei token da tutti i suoi posti di input, e "mettendone" altri in tutti i suoi posti di output. Le RDP sono quindi event-driven.

L'evoluzione della rete rappresenta l'evoluzione del sistema, da una situazione ad una successiva, ciascuna modellata da più stati parziali (uno stato globale distribuito).

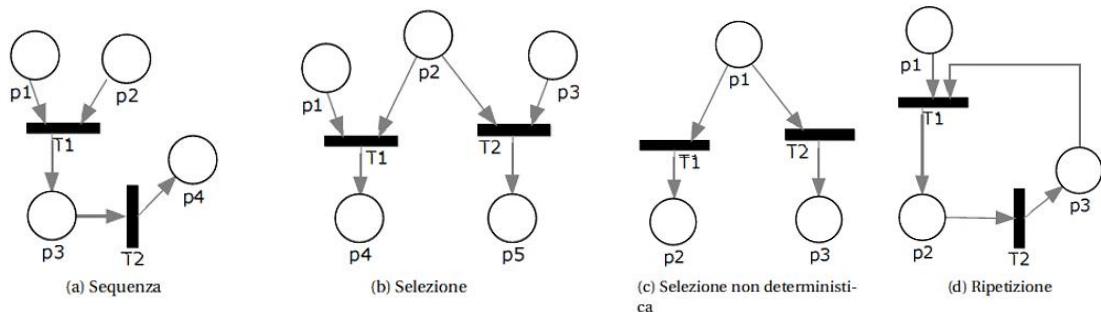
Lo **scatto della transizione** avviene se si verificano le seguenti condizioni:

- **Una transizione è abilitata allo scatto** se tutti i posti in input contengono token da prelevare. Allo scatto verranno prelevati token dai posti in input alla transizione stessa.
- **Una sola transizione per volta può scattare**; ciò per evitare che due transizioni abilitate prelevino lo stesso token. La transizione che è scattata è scelta in modo non deterministico.

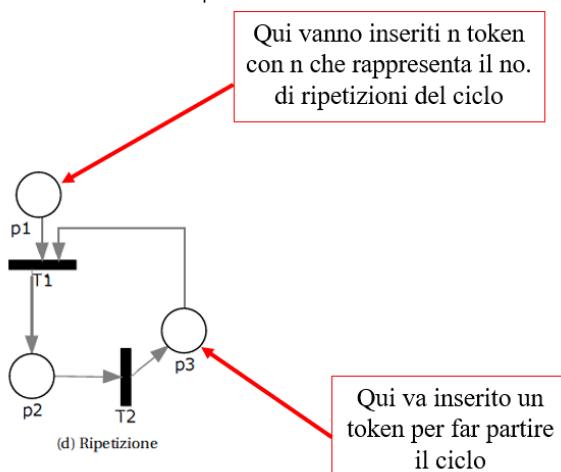
In una visione più generale, un arco può avere associato un peso (positivo). Tale peso se associato ad un arco che connette un posto di input ad una transizione, specifica il numero minimo di token necessari affinché il posto in esame possa dare il proprio contributo ad abilitare la transizione.

Il peso specifica anche quanti token saranno tolti dal posto da cui si diparte l'arco verso la transizione che scatta. Viceversa, se il peso si riferisce ad un nodo di output, il numero specifica quanti token verranno messi nel posto una volta scattata la transizione.

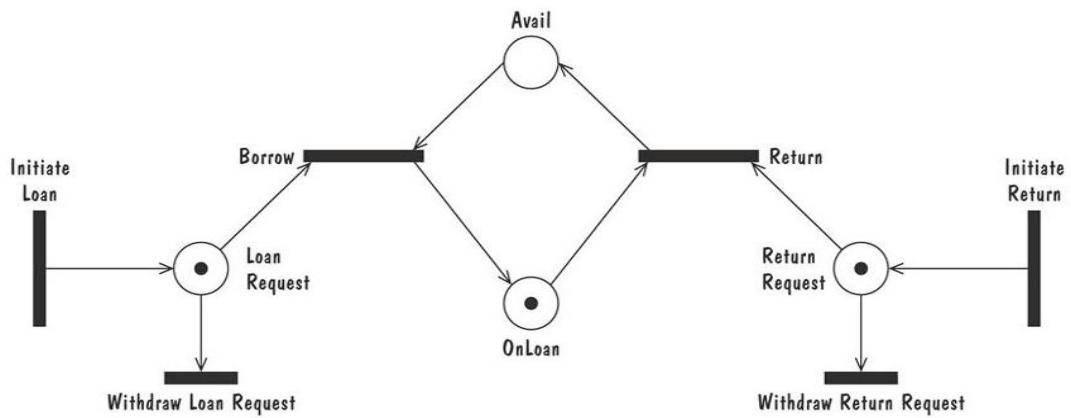
Esempi di configurazioni RDP che rappresentano costrutti di controllo elementari:



Esempio di ciclo:



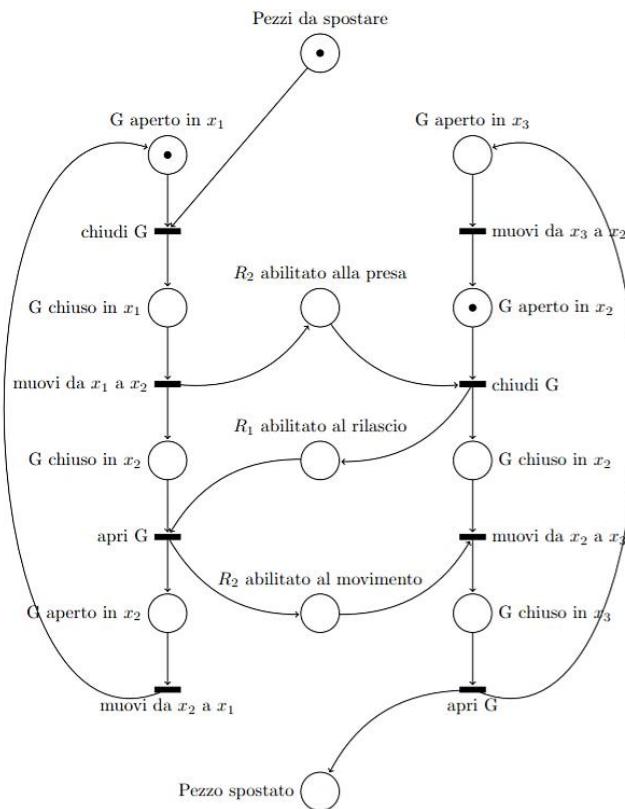
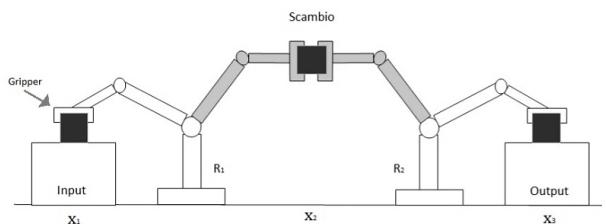
Esempio RdP sistema bibliotecario:



es. Reti di Petri

In figura sono rappresentati due robot, che eseguono in modo coordinato l'operazione PICK-&PLACE di un oggetto. Ogni robot, per tale operazione, è dotato di una pinza (gripper G) che può essere aperta o chiusa. L'operazione di PICK & PLACE può essere a sua volta scomposta in quattro sotto operazioni:

1. operazione di PICK: R1 effettua il prelievo dell'oggetto dalla stazione di input;
2. trasporto dell'oggetto dalla stazione di input al punto di incontro;
3. scambio dell'oggetto da R1 a R2;
4. trasporto dell'oggetto dal punto di incontro alla stazione di output.



trasformazioni fra modelli possono essere definite e applicate automaticamente da strumenti software.

UML

Modellazione software (software modeling): approcci e motivazioni

- è un approccio pratico, focalizzato sulle tecnologie e strumenti largamente accettati e utilizzati nell'industria.
- permette di **costruire modelli indipendenti dal linguaggio** (language-independent).
- dà vita al paradigma dello sviluppo guidato da modelli (model-driven development).
- ha un grande impatto nella fase di analisi, design e documentazione.

Un modello software (similmente a un modello di sistema) è una semplificazione della realtà, è una descrizione accurata e possibilmente parziale di un sistema in fase di studio ad un qualche livello di astrazione. Un modello software:

- è formato da diversi sotto modelli che descrivono una certa vista del sistema;
- non ha bisogno di essere completo;
- è espresso in qualche linguaggio a qualche livello di astrazione;
- è più di una descrizione: è una rappresentazione analogica di quello che vuole modellare.

I modelli software ci aiutano a **visualizzare un sistema così com'è**; ci permettono di specificare sia la **struttura che il comportamento del sistema**. Ci forniscono un template che ci guida durante l'intera costruzione del sistema e inoltre documentano le decisioni prese dal team.

L'**UML** è un **linguaggio di modellazione** standardizzato (ISO) che ci permette di **visualizzare, specificare, costruire e documentare artefatti software**. UML nasce **come strumento di comunicazione nelle fasi di progettazione**: un fallimento di comunicazione durante il processo di sviluppo può portare a conseguenze negative. UML è indipendente dalle metodologie di sviluppo.

L'**UML** non è un **linguaggio di programmazione**: è **indipendente dai dettagli di basso livello** (è machine independent). UML serve come mezzo di discussione dei problemi (requisiti, analisi, design). UML fornisce diverse **viste di diverso livello di dettaglio dello stesso artefatto**.

Segue una lista dei principali modelli software utilizzati, in ambito del linguaggio UML

Casi d'uso

Un **caso d'uso** è una **descrizione di una serie di sequenze di azioni** (comprese le loro varianti), che un sistema esegue per produrre un risultato di valore osservabile per un attore. Un caso d'uso **descrive cosa fa un sistema ma non specifica come lo fa**. Un caso d'uso solitamente rappresenta un **pezzo principale di funzionalità** che porta valore tangibile all'utente. Un caso d'uso è la descrizione di uno scenario (o di set di scenari strettamente correlati) nel quale il sistema interagisce con l'utente. I casi d'uso sono descritti da scenari narrativi e modelli grafici.

Un **attore** è un **agente esterno che deve/vuole interagire con il sistema** e che rappresenta uno specifico ruolo. Gli attori NON fanno parte del sistema.

Notazione dei casi d'uso:

- Caso d'uso: ovale
- Attore: stickman



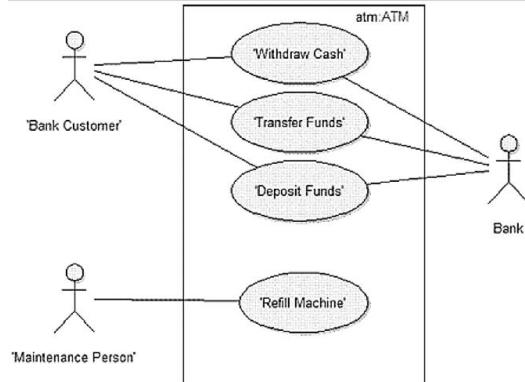
ha senso usarlo solo quando c'è interazione con l'utente.
come vedo se è INTERATTIVO? con DFD liv.0 vedo quali sono gli attori.

Un diagramma dei casi d'uso è un modo eccellente per comunicare alla gestione, ai clienti e alle altre persone che non sviluppano il software cosa farà il sistema quando sarà completato.

I diagrammi dei casi d'uso sono usati per modellare il **contesto** e i **requisiti** di un sistema. Inoltre, forniscono la prospettiva dell'utente del sistema.

Un'associazione tra caso d'uso e attore indica che l'attore e il caso d'uso comunicano l'uno con l'altro, possibilmente mandando e ricevendo messaggi. L'associazione descrive il "canale" di interazione. È importante notare che un'associazione:

- non modella un flusso di dati;
- non indica una direzione del flusso di interazione, dunque l'associazione è raramente ornata da una freccia terminale;
- l'interazione è descritta da un testo fuori dal diagramma.



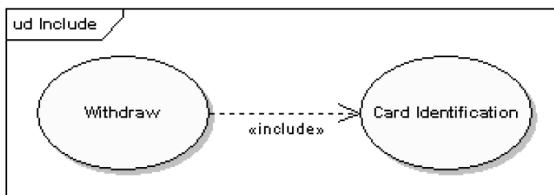
È possibile notare delle associazioni dall'esempio di diagramma di casi d'uso ATM sovrastante.

Relazioni tra casi d'uso

La relazione tra casi d'uso <<include>> specifica che il caso d'uso d'origine **incorpora esplicitamente il comportamento del caso d'uso incluso**. Si ha che:

- il caso d'uso d'origine specifica il punto esatto in cui il flusso è sospeso e il caso d'uso è iniettato.
- Alla fine dell'esecuzione del caso d'uso incluso, il caso d'uso d'origine continua la sua esecuzione dal punto in cui è stato sospeso.
- Tutte queste descrizioni sono di natura testuale e quindi posizionate al di fuori del modello grafico.

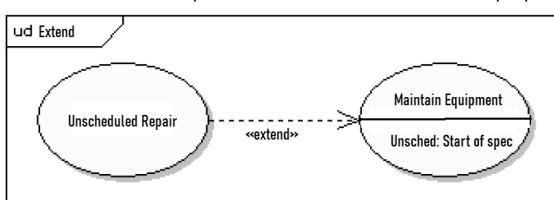
Un esempio è il seguente: "card identification" è incluso in "withdraw"



La relazione tra casi d'uso <<extend>> specifica che il caso d'uso di partenza **estende il comportamento del caso d'uso di arrivo, aggiungendo una logica personalizzata eccezionale in una posizione specificata dall'origine**. Si ha che: NON CENTRA CON EREDITARIETÀ

- il punto di estensione specifica dove il caso d'uso d'origine è sospeso; la condizione specifica la regola che avvia l'attivazione del caso d'uso target.
- La ripresa dal caso d'uso d'origine è simile al caso di inclusione.

Un esempio è il seguente: "unscheduled repair" estende "maintain equipment"

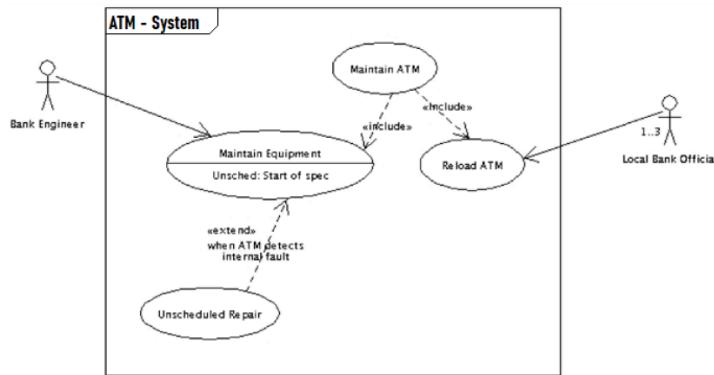


es. su Amazon se voglio accedere mi apre la pagina di LOGIN. Se non sono registrato e seleziono "registriati" allora mi apre temporaneamente sopra al login un form di registrazione (una volta finito torna al form di login) => la registrazione e' stata iniettata dentro la finestra di login.

Il caso d'uso EXTEND viene chiamato solo in alcuni casi specifici, mentre il caso INCLUDE viene richiamato sempre.

Notazione: si noti che le frecce di <<include>> ed <<extend>> sono tratteggiate e aperte all'estremità.

Esempio complessivo di caso d'uso:

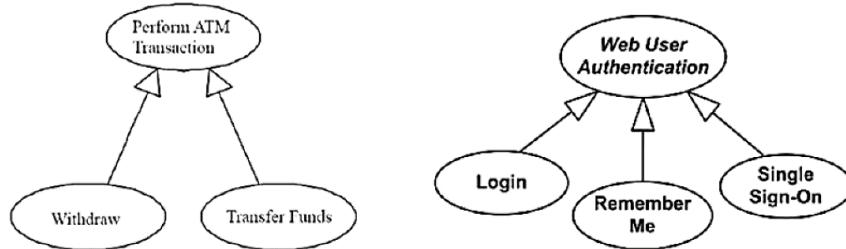


Sempre ritornando alla relazione extend, in relazione all'esempio:

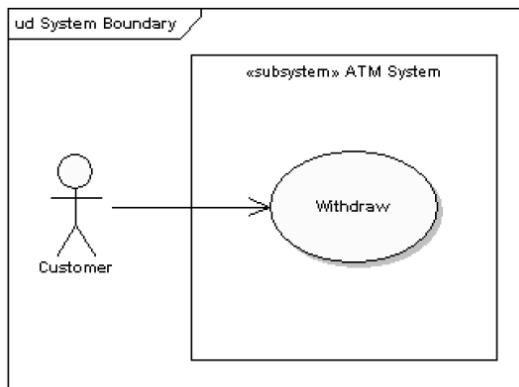
- "when ATM detects internal fault" è la **condition** per la relazione extend;
- "unsched: start of spec" è l'**extension point** per la relazione extend;

La relazione tra casi d'uso di **generalizzazione** specifica che **un caso d'uso (figlio)** eredita la struttura, il comportamento e le relazioni di un altro **attore padre**. Il figlio è il caso d'uso più specializzato, mentre il padre è quello più astratto della relazione. La notazione è una freccia completa con l'estremità chiusa dal caso figlio al caso padre

Esempi di generalizzazione:



I **confini del sistema** sono usati per definire confini concettuali, aiutano a raggruppare elementi correlati logicamente e aiutano a mostrare cosa **risiede dentro il sistema** (le funzionalità) e cosa **sta fuori** (l'utente).



Guidelines

Segue una serie di linee guida per i diagrammi dei casi d'uso:

- Iniziare il **nome dei casi d'uso con un verbo** (affermare il valore reale per l'attore)
- Cercare di **evitare verbi vaghi** come gestire, processare ed eseguire
- Sfruttare il vocabolario del dominio
- Fare attenzione a non usare troppi livelli nidificati di casi d'uso di inclusione, questo significa **focalizzarsi sui dettagli di implementazione** (come) che non è lo scopo dei casi d'uso (cosa).
 - in linea generale: **max. un livello nidificato**.

Diagramma delle classi

Una **classe** è un **template** per la definizione di tutte le caratteristiche di un oggetto, come attributi e **metodi**. Una classe **definisce** una serie di oggetti con la stessa struttura comune e lo stesso **comportamento**.

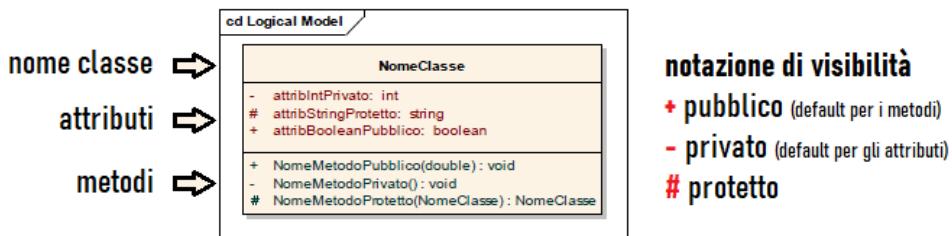
Un **attributo** descrive un'**informazione relativa allo stato interno di un oggetto** (istanza della classe). Ogni oggetto mantiene un valore per ciascun attributo per la corrispondente classe di appartenenza.

Un **metodo** definisce il **comportamento di un oggetto per uno specifico evento**. Il comportamento di un oggetto può essere indipendente dallo stato o dipendente dallo stato.

Un **diagramma delle classi** descrive le tipologie di oggetti presenti in un sistema e le relazioni statiche che esistono tra di loro. È una tecnica di modellazione basata sui principi dell'orientazione agli oggetti. È la più ricca notazione in UML.

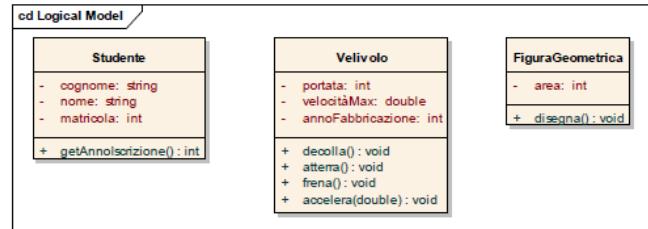
Segue la notazione grafica UML del diagramma delle classi:

Una **classe** è una descrizione di un insieme di oggetti che hanno attributi, operazioni, relazioni e comportamenti simili. Tipicamente una classe descrive un'entità del dominio del problema.



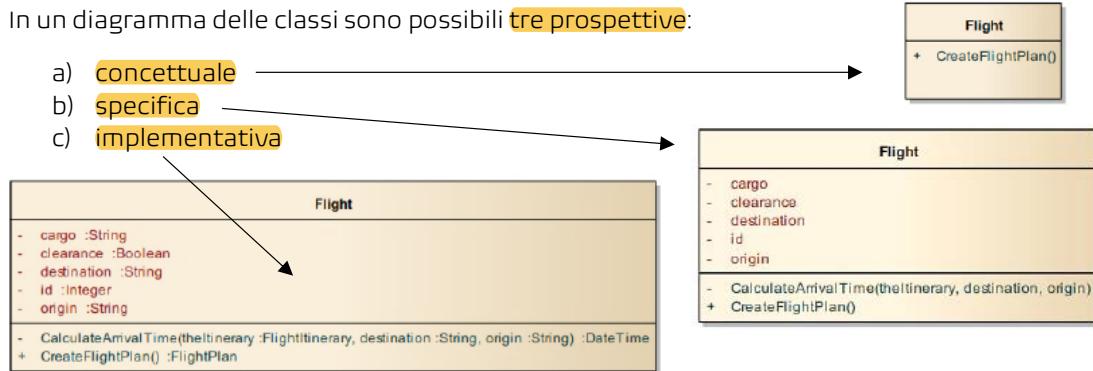
convenzioni (linee guida) sui nomi:

- **Classe:** iniziale maiuscola per ogni parola
- **Attributi:** iniziale minuscola, maiuscola per ogni singola parola successiva
- **Metodi:**
 - C++: maiuscola anche la prima
 - Java: come gli attributi



In un diagramma delle classi sono possibili **tre prospettive**:

- a) **concettuale**
- b) **specifica**
- c) **implementativa**



La relazione di **associazione** è una **relazione "semantica"** tra due o più classi che specifica **una connessione tra le loro istanze**. È una relazione strutturale che specifica che due oggetti di una classe sono connessi a oggetti di un'altra classe (possibilmente anche la stessa).

Esistono relazioni mono o bi-direzionali:

- **origine** e **target** (classe di origine e destinazione)
- gli end-point possono includere delle etichette con numeri o descrizioni.



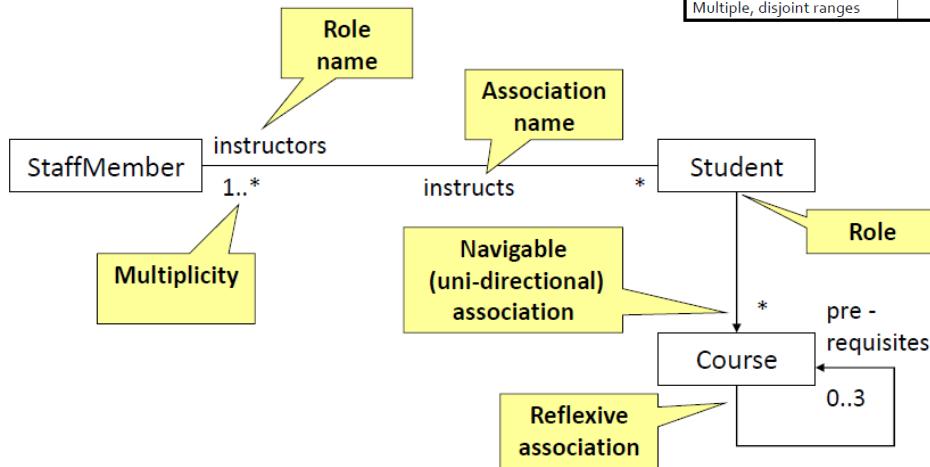
Un'**associazione** tra due classi specifica che l'oggetto di un **end-point** conosce l'oggetto dell'**altro end-point**, e sono capaci di **scambiarsi messaggi**. Notazioni optionali di un'associazione:

- **nome dell'associazione** (opzionale): etichetta posizionata a metà della relazione che rappresenta un verbo che fornisce l'implicita azione della relazione.
- **nome del ruolo** (opzionale): etichetta posizionata alla fine degli end-point che specifica il ruolo dell'end-point nel contesto dell'associazione.

Molteplicità: indica quanti oggetti di una classe possono far riferimento ad ogni oggetto dell'altra. Permette di indicare se un'associazione è obbligatoria o meno, e il lower e upper bound del numero di istanze.

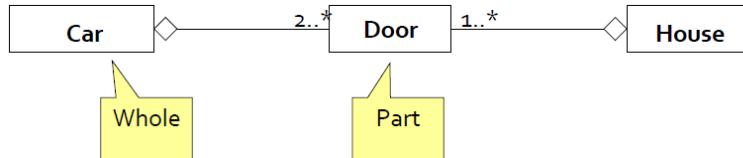
Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

Esempio:



Aggregazione: è una forma speciale di associazione che modella una relazione "whole-part" tra un aggregato (whole – il tutto) e le sue parti. Modella il concetto di "è parte di" e "contiene".

Esempio: car-door; house-door;

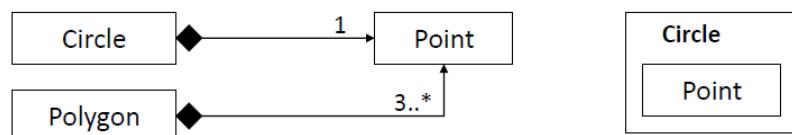


Composizione: è una forte forma di contenimento (più forte dell'aggregazione).

- il "whole" è l'unico proprietario di ogni parte.
- le molteplicità nel "whole" devono essere 1 o 0.

Mentre in AGGREGAZIONE posso avere INSTANZE CONDIVISE con COMPOSIZIONE le INSTANZE sono DISTINTE E SERIALI

Esempio:

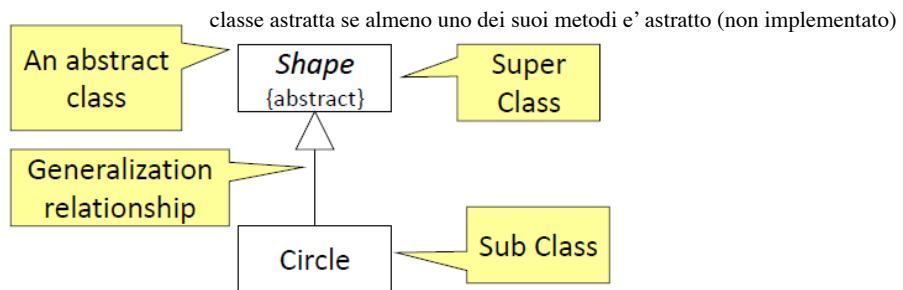


Generalizzazione: indica che oggetti di una classe specializzata (sottoclasse – subclass) sono sostituibili da oggetti di una classe generale (classe generale – superclass).

Una sottoclasse eredita dalla superclasse: attributi, operazioni e relazioni.

Una sottoclasse può aggiungere operazioni e relazioni, rifinire (sostituire – override) le operazioni ereditate dalla superclasse.

{abstract} is a tagged value that indicates that the class is abstract. The name of an abstract class should be italicized



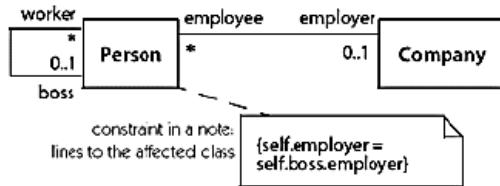
Una relazione di **realizzazione** indica che una classe implementa un comportamento specifico di qualche interfaccia. Un'interfaccia può essere realizzata da più classi. Una classe può realizzare più interfacce. Con interfaccia ci si riferisce al concetto di "insieme di comportamenti visibili".

Un figlio può estendere solo un padre



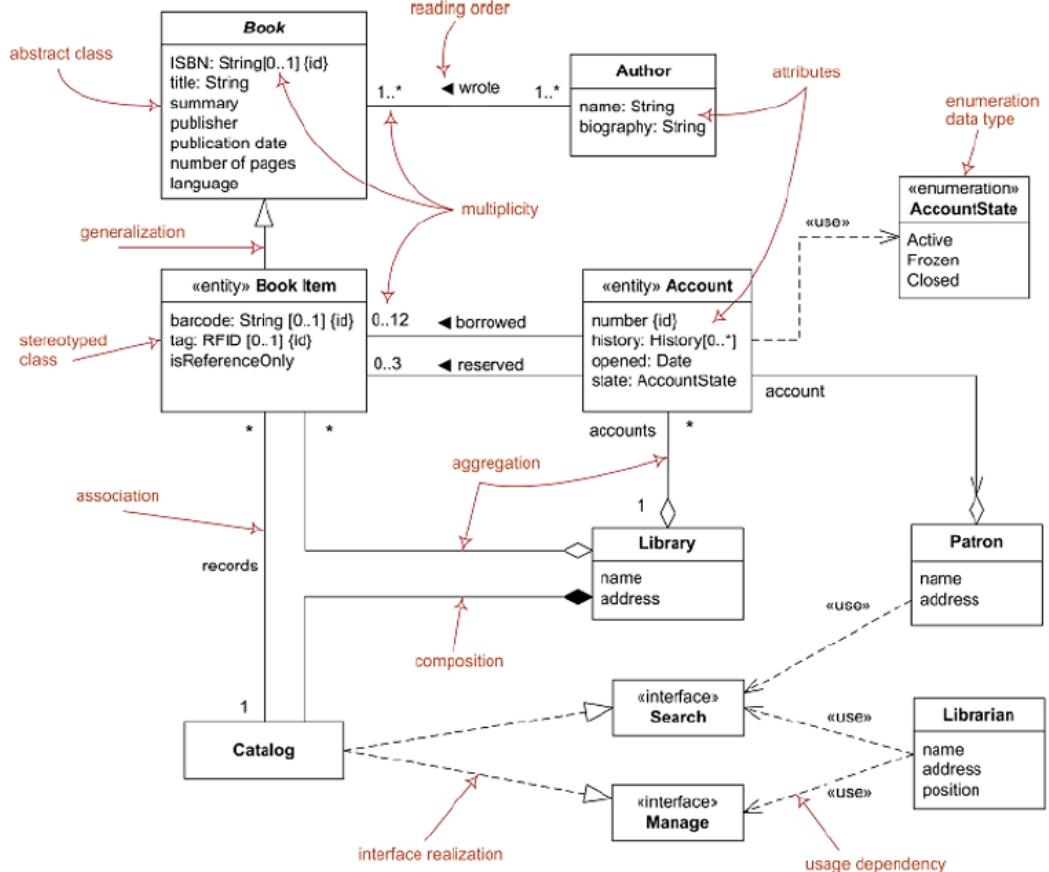
Una **dipendenza** indica una relazione tra due classi sebbene non ci sia un'esplicita associazione tra di loro. È una forma più debole di accoppiamento.



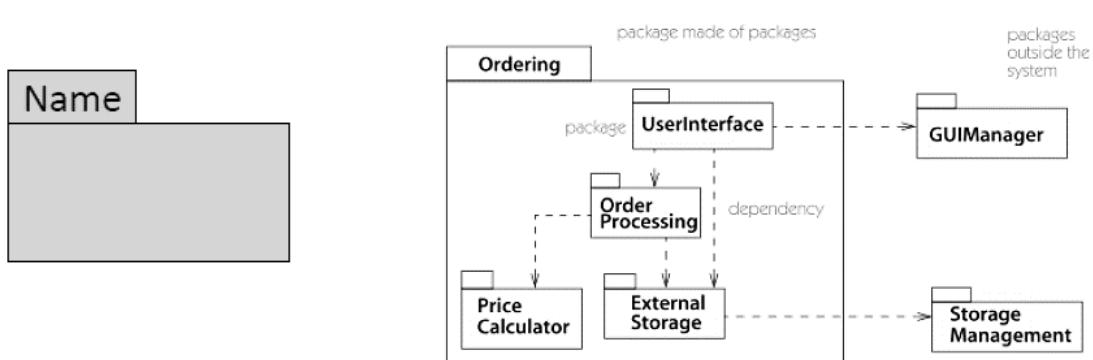


I **vincoli** (constraints) sono note che pongono restrizioni semantiche annotate come boolean expression. I vincoli sono utilizzati per rappresentare assunzioni, descrivere invarianti, descrivere pre e post condizioni dei metodi.

Esempio riassuntivo della notazione:



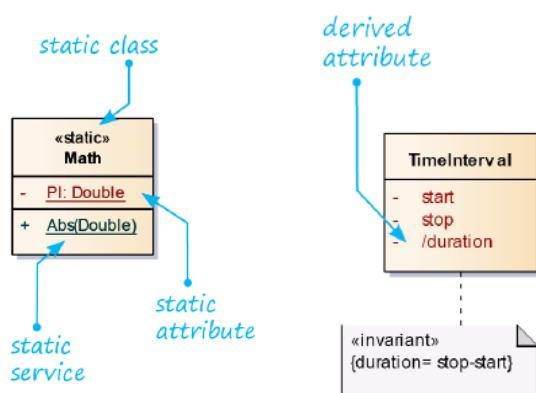
Un **package** è un meccanismo di raggruppamento general purpose. Viene tipicamente utilizzato per specificare l'architettura logica del sistema. Un package non si traduce per forza in un sottosistema fisico.



Static, derived, red only, frozen:

Vincoli:

- **{readOnly}** è un vincolo che precisa che il valore dell'attributo a cui è riferito non può essere modificato dall'esterno.
- **{frozen}** è un vincolo che precisa che il valore dell'attributo a cui è riferito non può cambiare per tutta la vita di un oggetto. (costante)



Diagrammi di sequenza

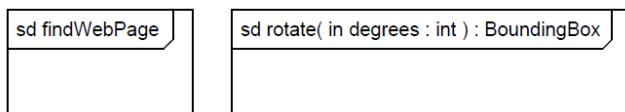
Un **diagramma di sequenza** descrive le **interazioni tra gli oggetti** coinvolti in uno specifico **scenario**. Gli oggetti collaborano per portare a termine un **task** o un **caso d'uso**. Gli oggetti collaborano **scambiandosi messaggi**. L'intero insieme di oggetti e le loro mutue interazioni in uno specifico scenario viene chiamato **collaborazione**. Il meccanismo di scambio messaggi nel paradigma OO è analogo a quello delle chiamate a funzioni nel paradigma procedurale.

I diagrammi di sequenza sono utili **per modellare**:

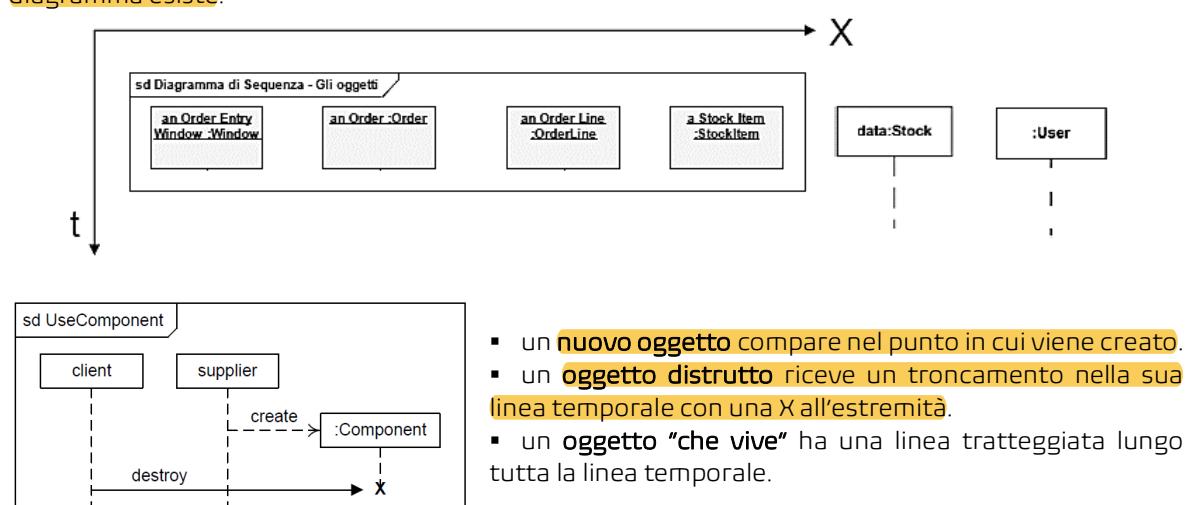
- interazioni in fase di progettazione "mid-level";
- le interazioni tra un prodotto e il suo ambiente;
- le **interazioni tra componenti di un sistema** nella progettazione architetturale.

Segue la notazione:

Un **frame** è un **rettangolo** con un **pentagono** nella parte superiore sinistra che indica il **nome del** **compartimento** (contesto).



Gli individui che partecipano sono disposti nel diagramma sotto forma di linee temporali. La **linea** **verticale** indica il **tempo**; la **linea** **tratteggiata** indica il **periodo di tempo** in cui un certo individuo del diagramma esiste.



Frecce:

- **Sincrono**: il mittente sospende l'esecuzione finché il messaggio non è completato.



- **Asincrono**: il mittente continua l'esecuzione dopo aver inviato il messaggio.



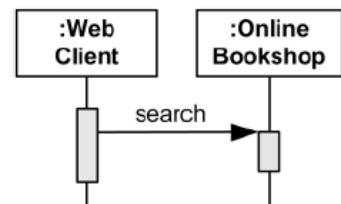
- Il messaggio sincrono ritorna o viene creata un'istanza.



Chiamate sincrone:

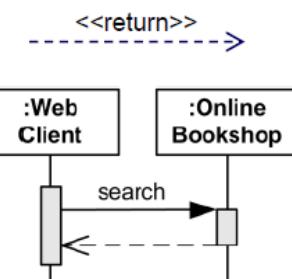
Frecce dall'estremità chiusa dal chiamante al chiamato. La freccia è adornata da una tabella con il nome del servizio chiamato (opzionalmente è possibile mostrare argomenti e valori di ritorno). Il chiamante aspetta la terminazione del servizio chiamato prima di continuare (sincrono). Il tempo di attivazione è mostrato da un rettangolo sopra la corrispettiva linea temporale dell'oggetto.

l'oggetto WebClient richiama il metodo search dell'oggetto OnlineBookshop

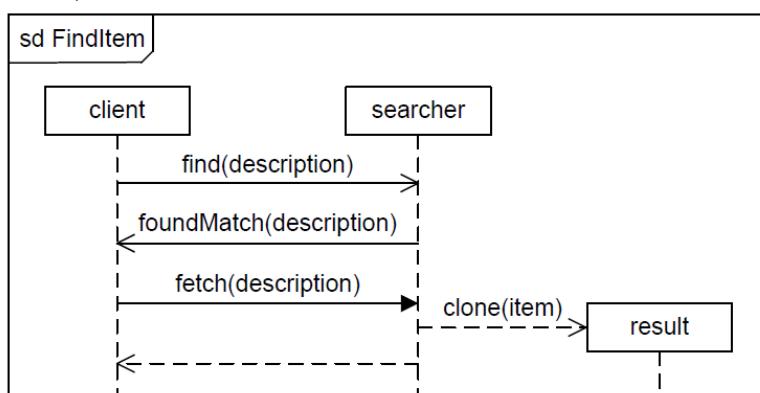


Messaggio di ritorno:

Il messaggio di ritorno è mostrato con una freccia tratteggiata dall'estremità aperta. Il messaggio di ritorno è **opzionale**, se non mostrato, la terminazione dell'esecuzione è determinata dalla fine del box di attivazione.

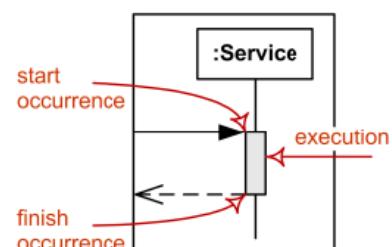


Esempio:



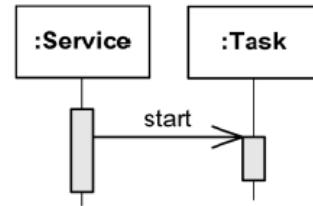
Istanze di esecuzione:

Una operazione è in esecuzione quando qualche processo sta eseguendo il proprio codice. Una operazione è sospesa quando invia un messaggio sincrono ed è in attesa del return. Una operazione è attiva quando sta eseguendo o è sospesa. Il periodo in cui un oggetto è attivo può essere mostrato usando un'istanza di esecuzione: un rettangolo sopra la linea temporale:



Chiamate asincrone:

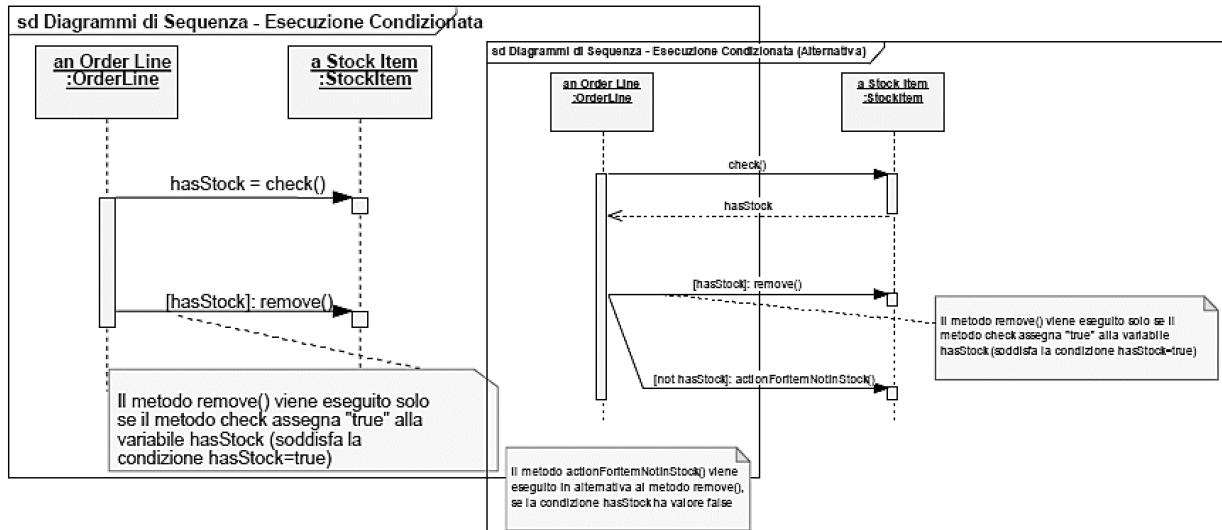
Adatte per **interazioni concorrenti**. La notazione è una freccia con l'estremità aperta dal chiamante al chiamato. La freccia è adornata dal nome del servizio invocato, e opzionalmente gli argomenti di input e i valori ritornati. Il chiamante può procedere l'esecuzione dopo aver inviato il messaggio.



Chiamate condizionali:

L'invio di un messaggio può essere soggetto a una condizione che viene controllata a run-time:

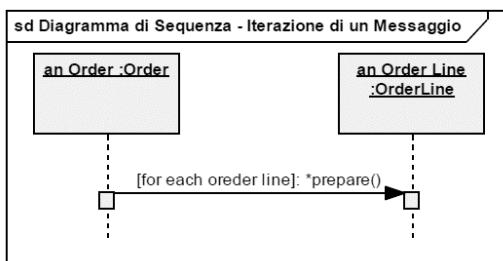
Sintassi: **(condizione) : nomeMetodo()**



Messaggio di interazione:

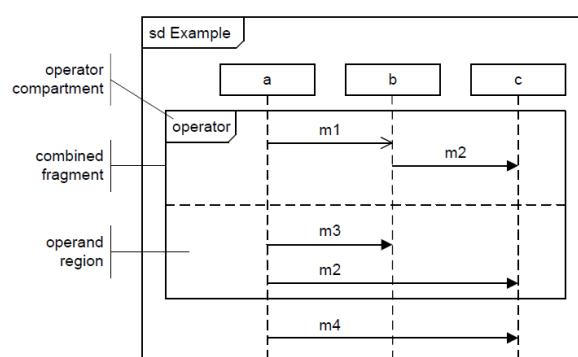
Esecuzione ciclica di un solo messaggio; la "guardia" specifica quando il loop deve terminare.

Sintassi: **(condizione) : * nomeMetodo()**

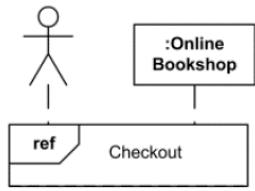


Frammenti combinati (combined fragments):

È una parte marcata di una interazione che mostra: rami, loop, esecuzioni concorrenti etc.



:Web Customer



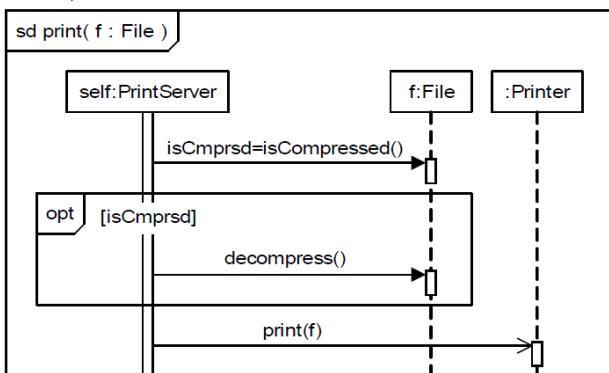
L'**uso di interazione** è un frammento che permette di usare (o chiamare) un'altra interazione. L'uso di interazione è un frammento combinato con l'operatore: **ref**.

I **frammenti opzionali** sono una porzione di interazione che potrebbe non essere eseguita; sono equivalenti alle dichiarazioni condizionali. L'operatore è: **opt**, con un solo operando con una guardia.

Una **guardia** è un'espressione booleana racchiusa tra parentesi quadre in un formato non specificato da UML.

Esempio:

IF

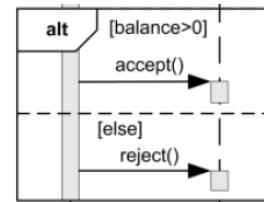
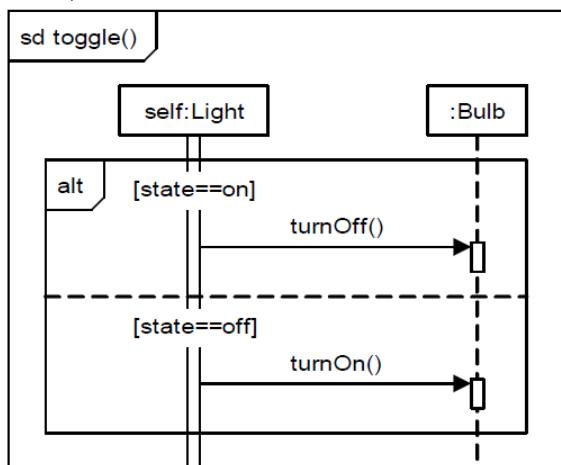


Post comments if there were no errors

I **frammenti alternativi** con una o più guardie come operandi sono mutuamente esclusivi. L'operatore è: **alt**.

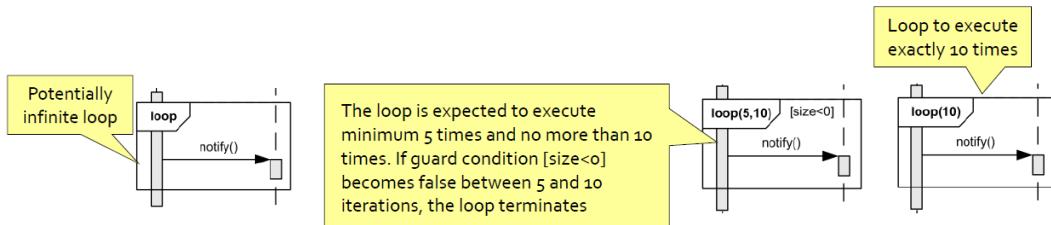
Esempio:

SWITCH



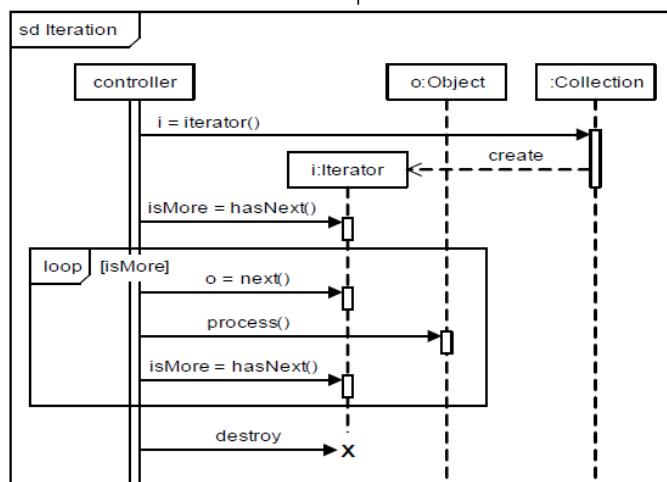
Call accept() if balance > 0, call reject() otherwise

I **frammenti di loop** sono corpi singoli che possono avere un operando guardia. L'operatore ha la seguente forma: **loop (min, max)** dove i parametri sono opzionali.



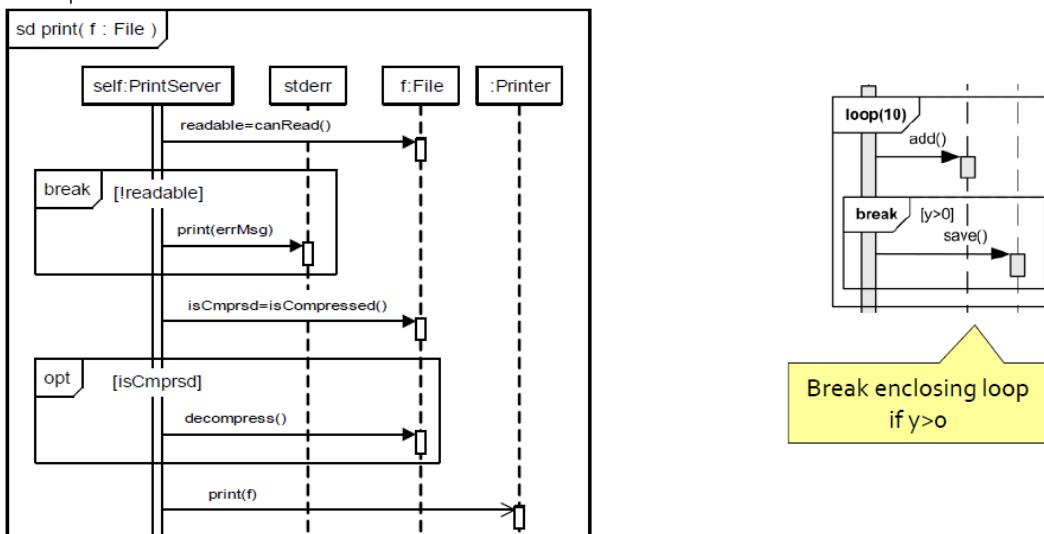
Il **corpo del loop** viene eseguito almeno **min** volte e al massimo **max** volte. Se il corpo del body è stato eseguito almeno min volte ma meno di max volte, viene eseguito solo se la guardia è true.

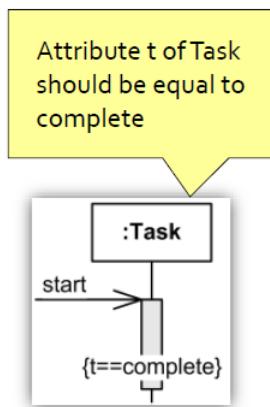
Esempio:



I **frammenti di rottura** (break fragment) sono frammenti combinati in cui un operando ha eseguito al posto di un operando o diagramma racchiuso se la guardia è vera. L'operatore è: **break**.

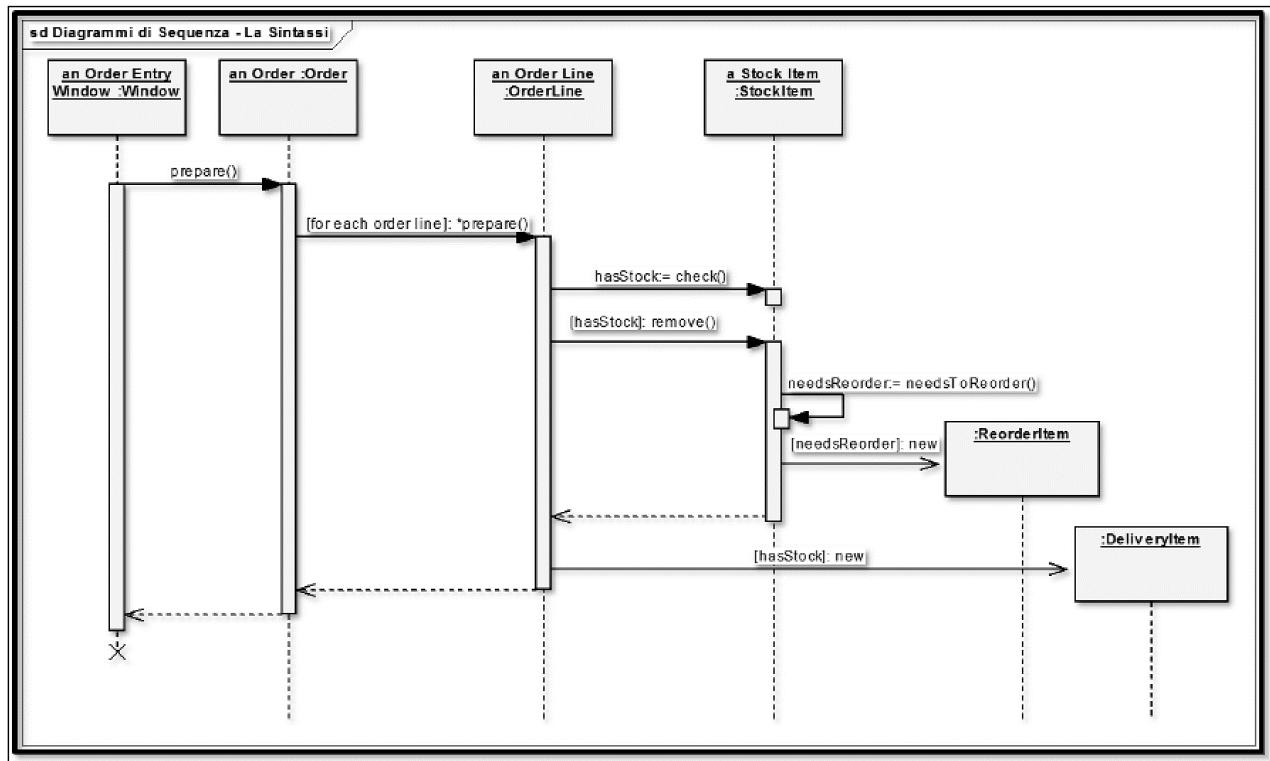
Esempio:



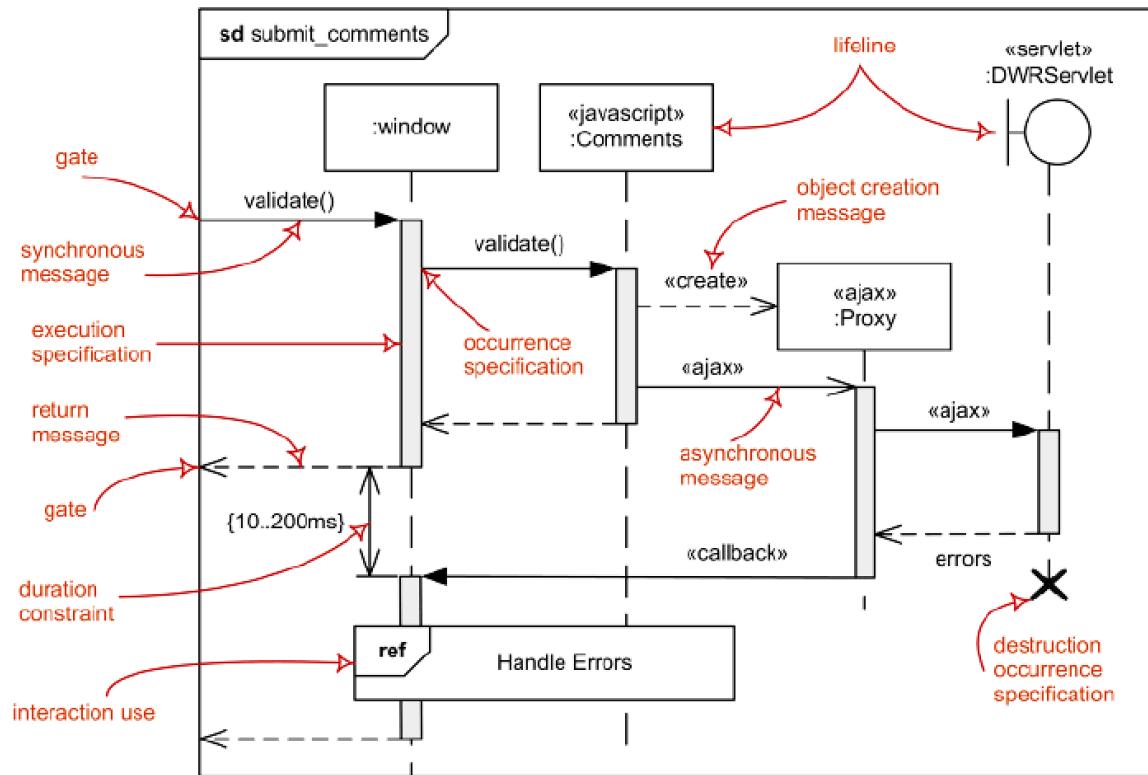


Le **invarianti di stato** sono un **frammento di interazione** che rappresentano **vincoli run-time** per il **partecipante dell'interazione**. Possono essere usati per specificare differenti **tipologie di vincoli**, come valori di attributi/variabili o stati interni/esterni etc. Il **vincolo** è valutato **immediatamente** **precedentemente** all'esecuzione della prossima occorrenza specificata così che tutte le azioni non esplicitamente modellate sono state eseguite.

Esempio di un diagramma di sequenza completo:



Riassunto sintassi:



Storie utente (user story mapping)

Una storia utente (user story) è una descrizione informale nel linguaggio naturale di una o più funzionalità del sistema software. Le storie utente sono scritte dalla prospettiva dell'utente finale del sistema. Sono spesso registrate in quelle che vengono chiamate "story card".

Una storia utente è una descrizione di una funzionalità che ha un valore sia per l'utente o acquirente del sistema. È una piccola parte di una funzionalità.

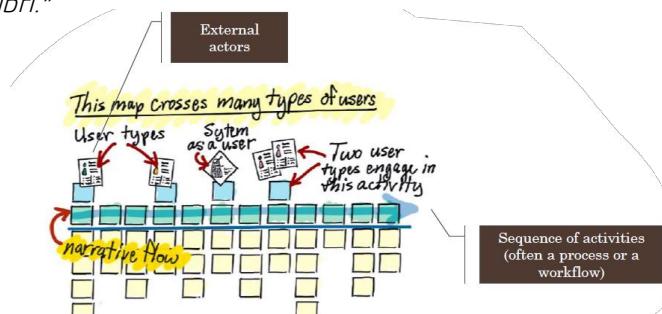
Il formato di una storia utente è: **ruolo – azione – scopo**

ovvero: **In quanto [ruolo] vorrei [azione] così che [scopo]**

esempio: *"In quanto utente, vorrei poter effettuare il login con la mia password, così da poter accedere al mio catalogo dei libri."*

Proprietà di una buona storia utente:

- Indipendente
- Negoziabile
- di Valore
- Stimabile
- Piccola
- Testabile



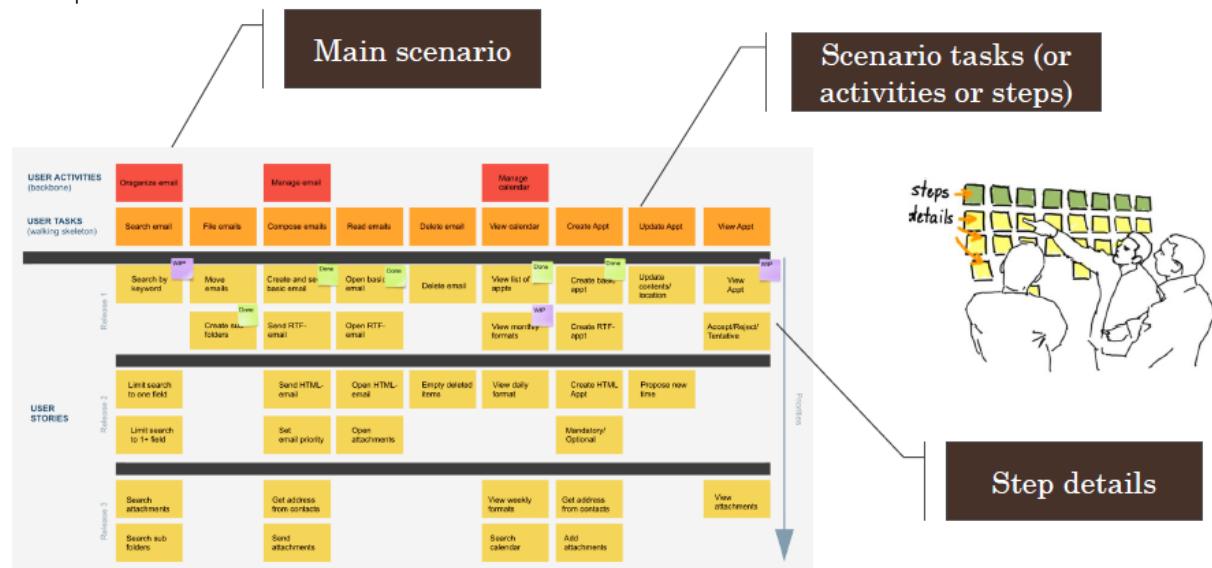
Dalle user story alle user story mapping

Una **user story map** è un grafico; una rappresentazione bidimensionale di una storia.

In cima alla mappa ci sono le **"epics"** (storie utente generiche), temi o attività.

Verticalmente, sotto le **"epics"** (epopee in italiano) abbiamo le story cards, collocate in ordine di priorità. La prima riga orizzontale è chiamata **"walking skeleton"** e mostra il flusso delle storie in un processo.

Esempio:



Mappare una storia aiuta a creare una comprensione condivisa del problema tra i membri del team. Le user stories a differenza dei casi d'uso sono informali.

Esercizio su UML e OOD (vecchio esame con svolgimento del prof. A. Baruzzo)

Si consideri un sistema di biglietteria elettronica in funzione all'interno di una metropolitana. Gli utenti comprano i biglietti attraverso i distributori automatici collocati lungo la rete metropolitana e collegati al sistema centralizzato che aggiorna i costi. Da ciascun distributore automatico, un utente può effettuare tre tipi di acquisto: un biglietto per una singola corsa, un biglietto di abbonamento settimanale e un biglietto di abbonamento mensile. Tutti e tre i biglietti sono riferiti a una specifica linea della metropolitana, coprendone tutte le tratte. Un acquisto può fallire per quattro motivi: l'acquirente ha impiegato troppo tempo per completare l'operazione; l'acquirente ha annullato l'operazione; il distributore ha esaurito la carta; il distributore ha esaurito il resto.

- (i) Si costruisca un modello dei requisiti mediante la tecnica dei casi d'uso.
- (ii) Si costruisca poi un modello delle classi per il dominio descritto.
- (iii) Si costruisca infine un diagramma di sequenza per l'acquisto di un biglietto di abbonamento mensile alla linea C.
- (iv) Per quanto riguarda la valutazione qualitativa del design, si calcolino le metriche di Lakos CCD, ACD, NCCD per il diagramma di classe elaborato al punto (ii) commentandone i risultati in termini di manutenibilità e testabilità.

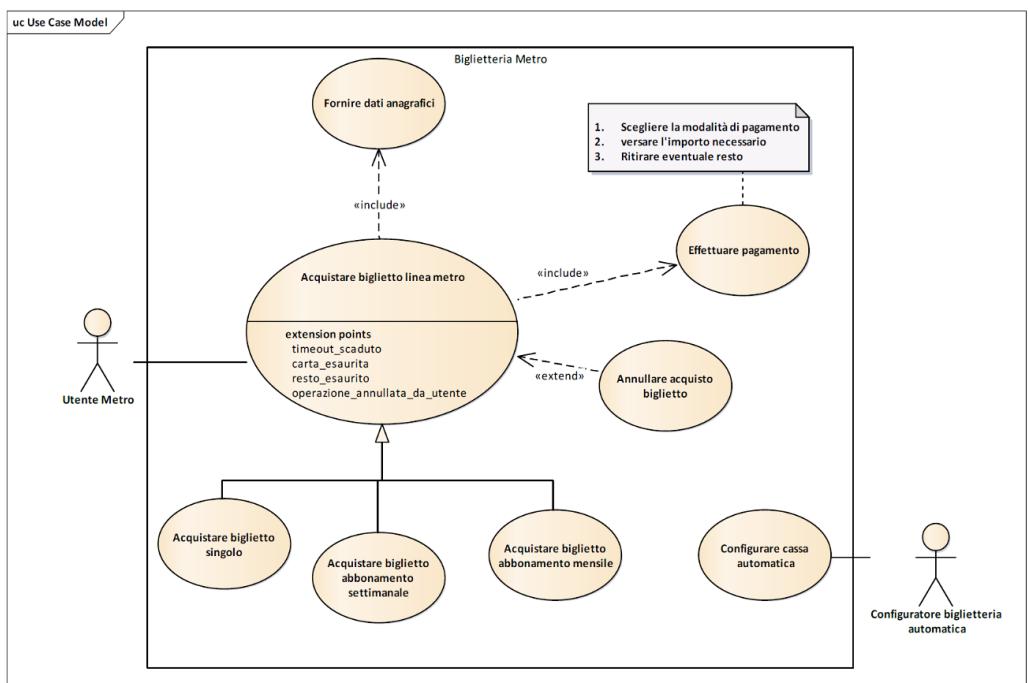
es. Casi d'uso

Come prima cosa identifichiamo il confine tra il sistema e l'ambiente esterno, introducendo un elemento "boundary" che chiamiamo Biglietteria Metro. Al suo interno vengono collocati tutti i casi d'uso, mentre al suo esterno vengono posizionati gli attori.

L'attore principale è l'utente della metropolitana (Utente Metro) il cui compito principale è di acquistare, mediante questo sistema, dei biglietti. L'acquisto è visto come un'operazione astratta che può essere istanziata nelle tre modalità previste, corrispondenti a casi d'uso concreti.

L'annullamento dell'acquisto è visto come un'estensione del caso d'uso principale, scatenato da una delle quattro possibilità, modellate come singoli punti di estensione. L'acquisto infine è scomposto in due use case inclusi, relativi all'inserimento dei dati anagrafici e all'effettuazione del pagamento. La nota associata al pagamento evidenzia tre passi chiave di tale caso d'uso.

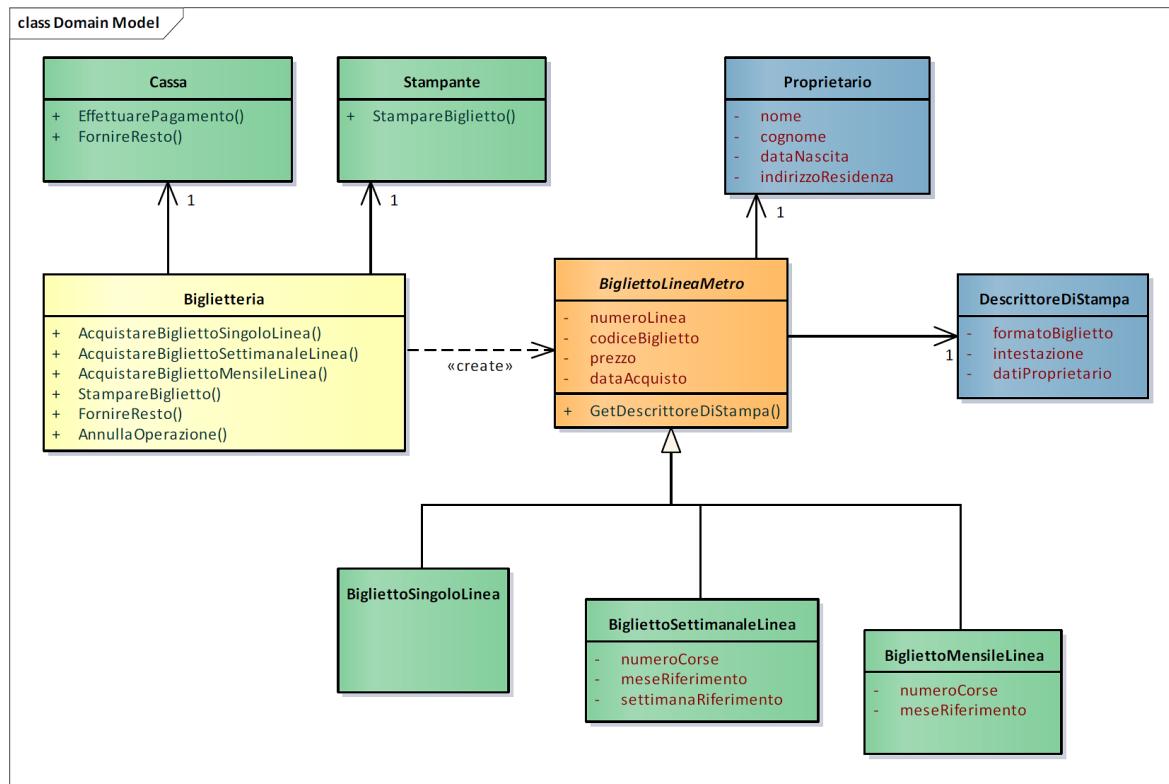
Un attore secondario è il configuratore della biglietteria automatica, con l'omonimo compito di impostare tutti i parametri che permettono alla biglietteria di funzionare (ad esempio, il messaggio di benvenuto, il taglio e la divisa di monete supportate, la tipologia di biglietti stampabili, la tipologia di strumenti di pagamento disponibili, etc.).



es. Diagramma delle classi

Per costruire un modello delle classi è necessario identificare i concetti chiave espressi nel contesto del problema e associarli poi tra loro mediante opportune dipendenze. Si viene così a formare un reticolo di classi legate da associazioni di idee, strutture gerarchiche di ereditarietà e strutture gerarchiche di contenimento. La Figura 2 illustra un possibile reticolo per il modello di dominio della biglietteria della metropolitana mediante la notazione dei diagrammi di classe UML.

In tale modello sono previste le operazioni di acquisto dei vari tipi di biglietto, il pagamento e la stampa del biglietto acquistato. Anche l'operazione di annullamento dell'acquisto è stata inserita come funzionalità offerta dalla biglietteria.

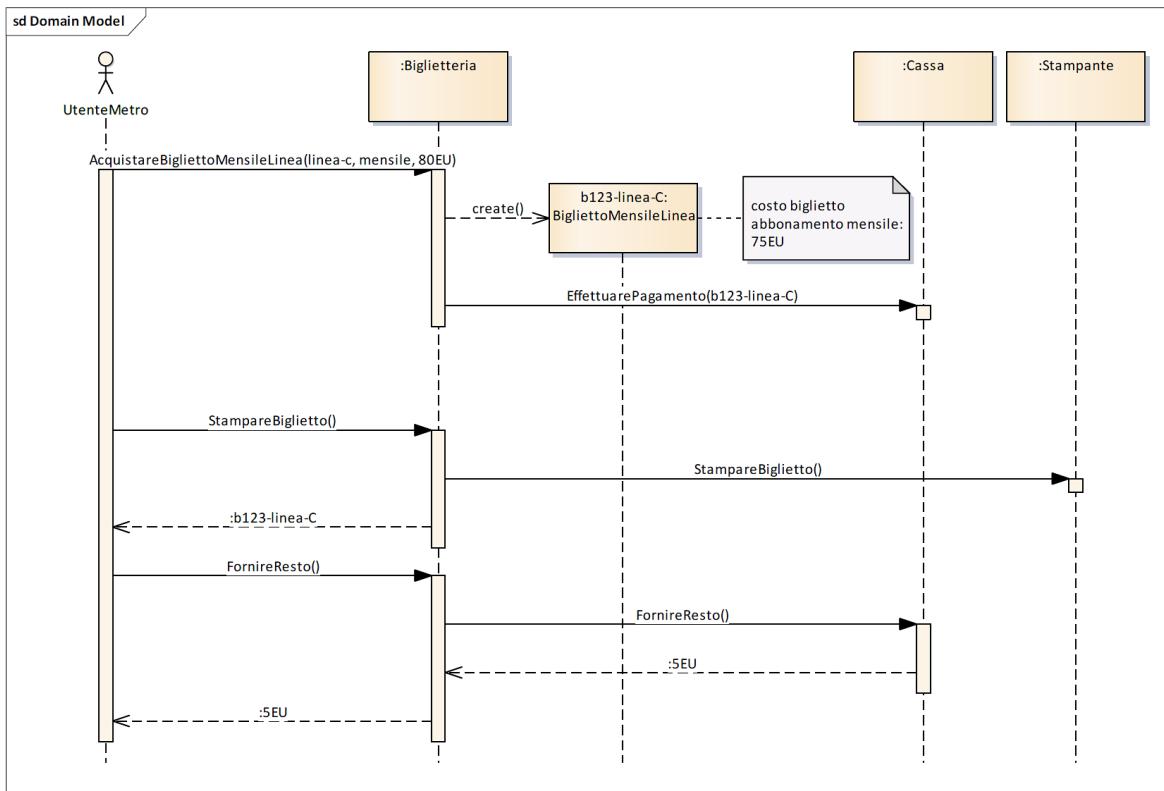


es. Diagramma di sequenza

Per costruire un modello dinamico con i diagrammi di sequenza è necessario individuare tutti i passi dello scenario di esecuzione che si vuole descrivere. Per ogni passo vanno individuati gli oggetti (istanze di classi nel diagramma di classe) e i comportamenti (metodi definiti per ciascun oggetto) necessari. Nel caso richiesto, la traccia d'esecuzione è relativa all'acquisto di un biglietto di abbonamento mensile per la linea C. Tale comportamento può essere scomposto nei seguenti passi:

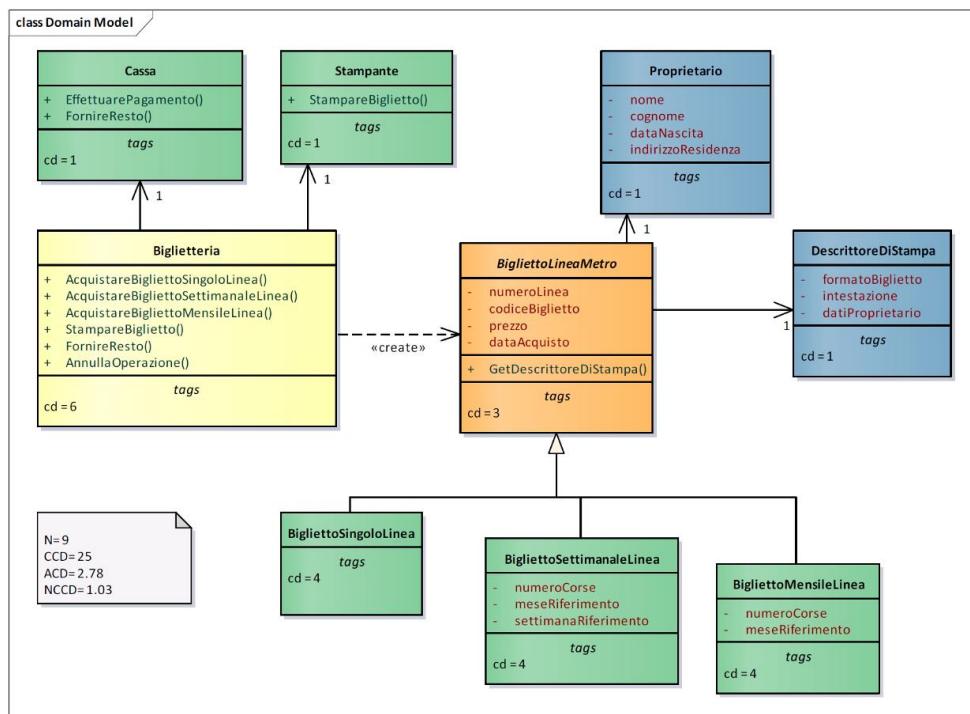
- L'utente della biglietteria accede alla biglietteria attraverso la funzione di acquisto biglietto per la tipologia di interesse (acquistareBigliettoMensileLinea);
- L'utente effettua il pagamento;
- L'utente ritira l'eventuale resto;
- L'utente stampa il biglietto.

Per completare questo scenario, è necessario inserire nel diagramma di sequenza gli oggetti relativi alla Biglietteria, al BigliettoMensileLinea, alla Stampante e alla Cassa. Il diagramma di Figura 3 illustra lo scenario richiesto.



es. Metriche di Lakos

Chiudiamo l'esercizio con il calcolo delle metriche di accoppiamento di Lakos, utili per valutare la manutenibilità e la testabilità del design object-oriented. La Figura 4 illustra le metrice CD (per ogni singola classe), CCD, ACD e NCCD (per l'intero design). Poiché le due condizioni di design ottimale $ACD < 10$ e $NCCD < 7$ sono entrambe soddisfatte, a parità di altri fattori la progettazione in esame è da ritenersi in linea con i dettami della metodologia ad oggetti. Ne consegue una manutenibilità e testabilità adeguata (almeno per quanto è possibile decretare dall'analisi delle sole metriche).



La fase di PROGETTAZIONE e' il processo di conversione delle specifiche di sistema in un documento con il progetto.

La fase di IMPLEMENTAZIONE e' il processo di conversione del progetto in un programma eseguibile, e' composto da PROGRAMMAZIONE e DEBUGGING (sempre interleaved)

Spesso queste due fasi possono essere inter-leaved.

La Progettazione si suddivide in:

- Design Architetturale: identifichi i sottosistemi e le relazioni tra essi e arrivi a definire qual e' l'architettura del sistema sw
- Design Interfaccia: definisci le interfacce tra i vari componenti di sistema
- Design dei componenti: progetti come dovrebbe funzionare ogni singolo componente
- Design del database: progetti le strutture dei dati

Andrea Mansi UNIUD 2019-2020

Riassunto concetti chiave Ingegneria del Software (ver. 1.1 - 5/2/2020)

La progettazione ha degli elementi di creatività ma nell'affrontare un problema specifico il progettista ha a disposizione degli schemi di RIFERIMENTO, sviluppati nel corso degli anni.

6. Progettazione architetturale

La progettazione architetturale si occupa dell'organizzazione di un sistema software e della progettazione della sua struttura complessiva. Nel modello del processo di sviluppo del software descritto nel capitolo 2, la progettazione architetturale è il primo stadio del processo di progettazione del software. È il **collegamento critico tra progettazione e ingegneria dei requisiti**, in quanto **identifica i principali componenti strutturali di un sistema e le loro relazioni**. L'output del **processo di progettazione architetturale è un modello architetturale che descrive come il sistema è organizzato in funzione dei componenti di comunicazione**.

In pratica, c'è una significativa sovrapposizione tra i processi di ingegneria dei requisiti e la progettazione architetturale. In teoria, una specifica del sistema non dovrebbe includere alcuna informazione sulla progettazione; questo è irrealizzabile, tranne per i sistemi molto piccoli. Occorre identificare i componenti architetturali principali, perché questi riflettono le caratteristiche di alto livello del sistema. È possibile progettare l'architettura del software a due livelli di astrazione, che possono essere chiamati:

- **Architettura in piccolo**: riguarda l'architettura dei singoli programmi.
- **Architettura in grande**: riguarda l'architettura di sistemi complessi che includono sottosistemi, programmi e componenti di programmi.

L'architettura del software è importante perché influisce sulle prestazioni, robustezza, distribuzione e manutenzione di un sistema. I **singoli componenti implementano i requisiti funzionali di un sistema, ma è l'architettura che ha un'influenza predominante sulle caratteristiche non funzionali di un sistema**.

Esistono **tre principali vantaggi** della progettazione e documentazione esplicita dell'architettura del software:

1. **Comunicazione tra gli stakeholder**: l'architettura è una presentazione di alto livello del sistema che può essere utilizzata per concentrare i temi di discussione fra diversi stakeholder e ingegneri.
2. **Analisi del sistema**: rendere esplicita l'architettura di un sistema, in un primo stadio di sviluppo, richiede alcune analisi. Le decisioni di progettazione architetturale influenzano profondamente la conformità del sistema ai requisiti critici come prestazioni, affidabilità e manutenibilità. Permette di capire se verifica i requisiti funzionali e non funzionali.
3. **Riutilizzo su vasta scala**: un modello architetturale di un sistema descrive in modo compatto e gestibile come è organizzato un sistema. L'architettura è spesso la stessa per sistemi con requisiti molto simili. Quindi, favorisce il riuso del software su vasta scala.

Le **architetture dei sistemi sono spesso modellate in modo informale utilizzando semplici diagrammi a blocchi**. Ciascun box nel diagramma rappresenta un componente. I box all'interno di un altro box rappresentano sottocomponenti. Le frecce indicano che i dati o i segnali di controllo passano da un componente all'altro nella direzione della freccia. Questo è un buon modo di supportare la comunicazione tra le persone coinvolte nel processo di progettazione del software, essendo tali diagrammi intuitivi. Tuttavia, a livello teorico, l'architettura di un sistema deve essere documentata nei dettagli, è meglio quindi utilizzare una notazione più rigorosa.

Decisioni di progettazione architetturale

La progettazione architetturale è un processo creativo in cui si progetta un'organizzazione che soddisfa i requisiti funzionali e non funzionali di un sistema. Non esiste sempre un processo di progettazione architetturale sperimentato. Il processo dipende dal tipo di sistema che si sta sviluppando, dal background e dall'esperienza del team. Il processo di progettazione architetturale è quindi più una serie di decisioni da prendere che una serie di attività ben definite.



Per quanto ogni sistema software sia unico, i sistemi nello stesso dominio di applicazione hanno spesso architetture simili che ne riflettono i concetti fondamentali.

L'architettura di un sistema software può basarsi su un particolare **schema** o **stile architetturale**. Uno schema architettonico è una descrizione dell'organizzazione del sistema, per esempio un'organizzazione client-server o un'architettura a strati. Gli schemi architettonici esprimono l'essenza dell'architettura che è stata utilizzata in diversi sistemi software.

Per scomporre le unità strutturali del sistema, bisogna scegliere la strategia di scomposizione dei componenti in sottocomponenti; infine, nel processo di modellazione del controllo, occorre sviluppare un modello generale delle relazioni di controllo tra le varie parti del sistema e decidere come controllare l'esecuzione dei componenti. A causa della stretta relazione tra le caratteristiche non funzionali del sistema e l'architettura hardware sottostante, la scelta dello schema architettonico e della struttura dipende pesantemente dai requisiti non funzionali del sistema. Come ad esempio: Parametri importanti che sono conflittuali tra loro (se favorisco uno, sfavorisco l'altro)

- **Prestazioni**: Ridurre il numero di sottofunzioni in modo da ridurre il numero di passaggi di parametri tra le funzioni
- **Protezione**: Più stratifico e' più sono sicuro che le cose a basso livello siano inaccessibili, pero' ne risento di prestazioni
- Sicurezza
- Disponibilità
- Manutenibilità

Tutti concetti da tenere in considerazione durante la progettazione architetturale. Valutare la bontà di un progetto architettonico è difficile perché il vero test è controllare in che modo il sistema soddisfa i requisiti funzionali e non funzionali una volta che il sistema viene utilizzato; tuttavia è possibile confrontare il proprio progetto con schemi architettonici di riferimento.

Viste architettoniche

È impossibile rappresentare tutte le informazioni relative all'architettura di un sistema in un singolo diagramma, in quanto un modello grafico può mostrare soltanto una vista o prospettiva del sistema completo. Di solito bisogna fornire più viste dell'architettura del software per i vari stakeholder:

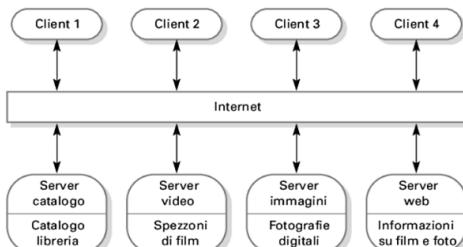
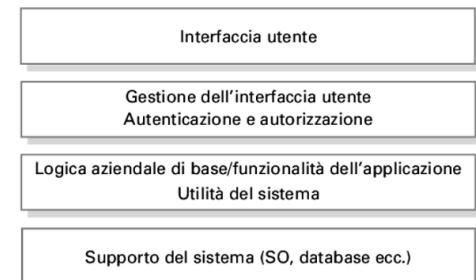
- **Vista logica**: mostra le astrazioni chiave nel sistema come oggetti o classi di oggetti.
- **Vista dei processi**: mostra come, a runtime, il sistema è scomposto da processi interattivi.
- **Vista di sviluppo**: mostra come il software viene scomposto per lo sviluppo; ovvero mostra la suddivisione del software nei suoi componenti che sono sviluppati da un singolo programmatore o gruppo.
- **Vista fisica**: mostra l'hardware del sistema e come i componenti del software sono distribuiti tra i componenti hardware.

Schemi architetturali

Come suddividere il sistema in SOTTOSISTEMI ? Ho vari stili da cui scegliere

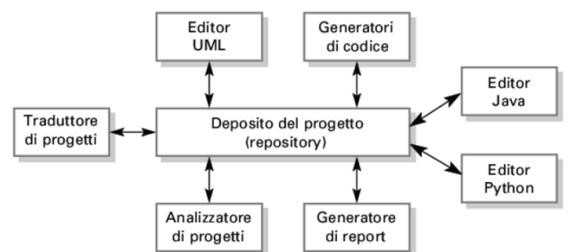
L'idea è quella di presentare, condividere e riutilizzare le conoscenze dei sistemi software e delle loro architetture. Uno **schema architettonico** può essere immaginato come una descrizione stilizzata di una buona pratica, che è stata provata e verificata in vari sistemi e ambienti (si veda l'analogia con i modelli di processo per lo sviluppo del software). Pertanto, uno schema architettonico dovrebbe descrivere l'organizzazione di un sistema che ha avuto successo; dovrebbe includere le informazioni su quando il suo utilizzo è appropriato e i dettagli su svantaggi e svantaggi del suo utilizzo. Seguono alcuni esempi significativi di schemi architettonici:

- **MVC (Model View Controller)**: presentazione e interazione separate dai dati del sistema. Il sistema è strutturato in tre componenti logiche che interagiscono tra loro. Il componente **Model** gestisce i dati del sistema e le operazioni associate. Il componente **View** definisce e gestisce il modo in cui i dati sono presentati all'utente. Il componente **Controller** gestisce l'interazione degli utenti e passa queste interazioni ai componenti Model e View. Consente ai dati di cambiare indipendentemente dalla loro rappresentazione e viceversa. Supporta la presentazione degli stessi dati in modi differenti.
- **Architettura a strati**: è un altro modo di ottenere la separazione e l'indipendenza tra i componenti. Le funzionalità del sistema sono organizzate in strati separati del sistema, ciascuno dei quali si basa sui servizi offerti dallo strato immediatamente sottostante. Questo approccio a strati supporta lo sviluppo incrementale dei sistemi. Quando viene sviluppato uno strato, alcuni dei servizi da esso forniti sono accessibili agli utenti. Quest'architettura è anche modificabile e portabile. Consente la sostituzione di interi strati, se l'interfaccia non viene modificata. Le funzioni ridondanti possono essere fornite in ogni strato per aumentare la fidatezza del sistema. Posso sviluppare un singolo strato con tecnologie diverse da quelle degli strati sopra e sotto, l'importante è che riescano a comunicare => MAGGIORE FLESSIBILITÀ



- **Architettura client-server**: lo schema client-server illustra una tipica organizzazione a runtime di sistemi distribuiti. Un sistema conforme allo schema client-server è organizzato come un insieme di servizi e server associati e di client che accedono e usano tali servizi. Le architetture client-server di solito sono concepite come architetture di sistemi distribuiti, ma il modello logico di servizi indipendenti che sono eseguiti su server separati può essere implementato su un singolo computer. Ancora una volta, la separazione e l'indipendenza sono un importante vantaggio. I servizi e i server possono essere modificati senza influire su altre parti del sistema. L'immagine mostra un esempio di architettura client-server per una libreria online di film e immagini. I server forniscono servizi specifici e possono essere DISTRIBUITI lungo la rete, i client utilizzano i servizi offerti e la rete serve a far comunicare i due (la rete puo' fare da collo di bottiglia).

- **Architettura repository**: spiega come una serie di componenti interattivi possono condividere i dati. Tutti i dati di un sistema vengono gestiti in un database centrale (detto repository) che è accessibile da tutti i componenti del sistema. I componenti non interagiscono direttamente, ma soltanto attraverso il database repository. I componenti possono essere indipendenti e le modifiche eseguite su un componente possono



essere rese note agli altri componenti, ma allo stesso tempo un componente non deve necessariamente sapere dell'esistenza degli altri componenti. Il repository è un punto comune di malfunzionamento, nel senso che i suoi problemi influiscono sull'intero sistema. L'immagine mostra un esempio di architettura a repository per un IDE.

- Architettura per sistemi distribuiti: [Capitolo 17](#)
- Architettura orientata ai servizi: [Capitolo 18](#)

Architetture applicative

Modelli architettonici progettati per le tipologie più comuni di applicazioni

I sistemi applicativi sono ideati per soddisfare necessità aziendali o organizzative. Tutte le aziende hanno molto in comune. Ne consegue che i sistemi applicativi utilizzati da queste aziende hanno molto in comune. Questi elementi comuni hanno portato allo sviluppo di architetture software che descrivono la struttura e l'organizzazione di particolari tipi di sistemi software. Le architetture delle applicazioni encapsulano le caratteristiche principali di una classe di sistemi. Un progettista può utilizzare questi modelli di architetture delle applicazioni in vari modi:

1. Come punto di partenza per il processo di progettazione architettonica;
2. Come lista di verifica per il progetto;
3. Come modo di organizzare il lavoro per il team di sviluppo;
4. Come mezzo per valutare la riusabilità dei componenti;
5. Come vocabolario per discutere delle applicazioni.

MODELLI DI CONTROLLO: si occupano di controllare il flusso dei sottosistemi (qual e' il prossimo modulo/funzione che va in esecuzione?) possono essere:

- controllo **CENTRALIZZATO**: Un sottosistema si occupa di assegnare/togliere il controllo ad un sottosistema, si suddivide in:
 - **CALL & RETURN**: programmat. sequenziale in cui quando richiamo un sottosistema l'esecuzione principale termina finche' il sottosistema non termina.
 - **MANAGER**: un programma centrale attiva i vari sottosistemi.
- controllo **EVENT DRIVEN**: Ogni sottosistema puo' rispondere ad eventi (esterni a lui) generati da altri sottosistemi; si attivano **ON DEMAND** i vari sottosistemi.
 - **BROADCAST**: si scatena un evento e si manda in broadcast una richiesta di controllo e il sottosistema che sa gestirlo prende il controllo
 - **INTERRUPT DRIVEN**: si scatena un evento e si va a vedere nel vettore delle interruzioni a che sottosistema affidare il controllo.

Un **SOTTOSISTEMA** e' un sistema con le sue operazioni che sono indipendenti da quelle degli altri sottosistemi.

Un **MODULO** e' una componente di un sistema che provvede dei **SERVIZI**.

l'architettura generale del sistema la scelgo con i pattern architetturali (client-srv,a livelli..) e poi per ogni sottosistema lo divido in moduli con la scomposizione modulare.

SCOMPOSIZIONE MODULARE: come scompongo un sottosistema in moduli? si suddivide in:

- a **OGGETTI**: sistema scomposto in oggetti che interagiscono tra loro
- **FUNZIONALE** (data-flow): sistema scomposto in moduli che trasformano gli inputs in un output (modello pipeline)
 - => applicano delle **TRASFORMAZIONI**
 - e' facile da capire e le funzioni di trasformazione possono essere riutilizzate

L'ARCHITETTURA del DOMINIO puo' essere **GENERICA** (bottom-up) o **SPECIFICA**(top down, sono derivate dallo studio del dominio applicativo al posto che da sistemi pre-esistenti)

7. Progettazione e implementazione

La progettazione e l'implementazione del software sono le fasi nel processo di ingegneria del software in cui viene sviluppato un sistema software eseguibile. Per alcuni sistemi semplici, ingegneria del software significa progettazione e implementazione del software, e tutte le altre attività di ingegneria del software si fondono in questo processo. Per i sistemi complessi, invece, la progettazione del software e l'implementazione sono solo due dei tanti processi di ingegneria del software. Le attività di progettazione e implementazione del software sono inevitabilmente intrecciate. La progettazione del software è un'attività creativa in cui vengono identificati i componenti e le loro relazioni, in base alle richieste di un cliente. L'implementazione è il processo che realizza il progetto sotto forma di programma eseguibile. Progettazione e implementazione sono strettamente collegate, quindi di solito bisogna prendere in considerazione i problemi di implementazione durante la fase di sviluppo del progetto.

Seguono i principali aspetti della fase di implementazione:

- **Riutilizzo:** il riutilizzo del software è possibile a vari livelli:
 - **Livello di astrazione:** a questo livello, non viene riutilizzato direttamente il software, ma le conoscenze di astrazioni nella progettazione del software. Vengono ad esempio riutilizzati schemi architetturali e di progettazione.
 - **Livello degli oggetti:** a questo livello, si riutilizzano direttamente gli oggetti da una libreria, anziché scrivere il codice da zero.
 - **Livello dei componenti:** i componenti sono raccolte di oggetti e classi di oggetti che operano insieme per fornire un servizio. Molte volte si utilizza un componente adattandolo e aggiungendo del proprio codice per non partire da zero.
 - **Livello del sistema:** a questo livello si riutilizza un sistema intero. Questa fase richiede solitamente un processo di configurazione del sistema, che viene fatto configurandolo o modificando il sistema per i propri interessi.

Utilizzando il software esistente si possono sviluppare nuovi sistemi più rapidamente, con minori rischi di sviluppo e a costi inferiori. Poiché il software riutilizzato è già testato esso dovrebbe essere più affidabile di quello implementato da zero. Tuttavia, il riutilizzo comporta alcuni costi: il costo del tempo impiegato per la ricerca del software, in alcuni casi il costo di acquisto del software riutilizzabile (API, interi sistemi etc.), costi di adattamento e riconfigurazione del software riusabile e il costo di integrazione. Come riutilizzare le conoscenze e il software esistente dovrebbe essere la prima cosa cui pensare quando si avvia un progetto di sviluppo del software. Bisogna valutare le possibilità di riutilizzo prima di progettare in dettaglio il software.

- **Gestione della configurazione:** la gestione della configurazione è il nome dato al processo generale di gestione di un sistema software che cambia. Scopo della gestione della configurazione è supportare il processo di integrazione del sistema in modo che tutti gli sviluppatori possano accedere al codice e ai documenti del progetto in modo controllato, trovare le modifiche che sono state apportate, compilare e collegare i componenti per creare un sistema. Questa fase è suddivisa in quattro fasi principali:
 - **Gestione delle versioni:** il supporto è fornito per tenere traccia delle differenti versioni dei componenti software.
 - **Integrazione del sistema:** il supporto è fornito per aiutare gli sviluppatori a definire quali versioni di componenti sono utilizzate per creare le singole versioni del sistema.
 - **Registrazione dei problemi:** il supporto è fornito per consentire agli utenti di segnalare i bug del codice e altri problemi e per consentire agli sviluppatori di tenerne traccia.
 - **Gestione delle release:** le nuove versioni di un sistema software vengono rilasciate ai clienti. La gestione delle release si occupa di pianificare le funzionalità delle nuove release e di organizzare la distribuzione del software.

L'obiettivo è capire che tecnica utilizzare per scomporre in moduli

47

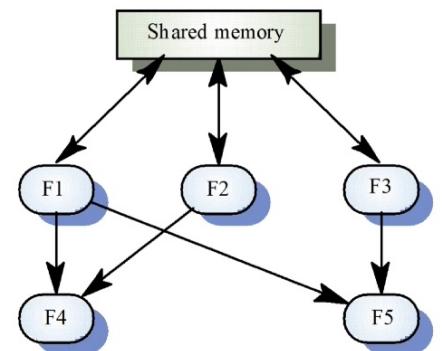
Esistono due tipi di STRATEGIE DI DESIGN :

- design ORIENTATO AGLI OGGETTI: il sistema è visto come una collezione di oggetti che interagiscono tra loro e si scambiano messaggi. Ogni oggetto ha il suo stato (quindi stato decentralizzato).
- design FUNZIONALE: il sistema è visto da una prospettiva funzionale, lo stato è centralizzato e CONDIVISO dalle funzioni.

APPROCIO FUNZIONALE

Utilizzato in sistemi in cui e' presente uno **STATO MINIMO**, ovvero dove tutte le funzioni sono indipendenti tra loro e l'output di una funzione non e' input di nessun'altra.

Le **informazioni sono CONDIVISE** tramite liste di parametri e repository condivise.

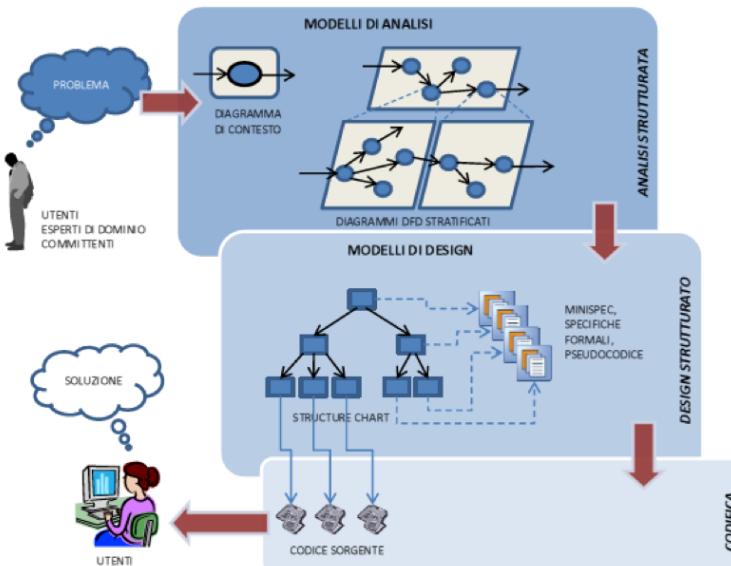


approccio utilizzato nei sistemi legacy o nei sistemi bancari con le transazioni.

FASI:

- design utilizzando i DFD
- trasformo il DFD in funzioni e poi capisco quali sottofunzioni sviluppare per implementare le funzioni (visualizzo le funzioni in un grafo)
- preparo il MINISPEC per i programmatorei
- codifica

il **MINISPEC** e' un **progetto esecutivo** per ciascun modulo della progettazione che viene passato ai programmatorei per la fase **successiva**, ovvero descrivo nel dettaglio le entita' e le loro interfacce. Viene usato anche per creare un data dictionary.



- **Sviluppo host-target:** il software viene sviluppato in un computer (host), ma viene eseguito in una macchina separata (target).

Dopo aver scelto l'organizzazione generale del sistema, si deve decidere l'approccio da usare per la scomposizione dei sottosistemi in moduli. **Un modulo è solitamente un componente di un sistema che fornisce uno o più servizi ad altri moduli, fa uso dei servizi forniti da altri moduli, e non è normalmente considerato un sistema indipendente.** Tipicamente i moduli sono formati da una serie di componenti più piccoli (che sono indipendenti). **Il principale scopo della scomposizione modulare è l'aumento della manutenibilità del sistema e della riusabilità.** Segue ora l'approccio OO; adatto alla scomposizione modulare:

Object-Oriented design

Un **progetto orientato agli oggetti determina quali classi e oggetti sono richieste e specifica come queste debbano interagire tra loro.** Un progetto più di basso livello include anche dettagli su come i singoli metodi debbano completare i propri task. La progettazione OO è quindi quel processo di progettazione che **mira a pianificare un sistema software come un insieme di oggetti che interagiscono tra loro.** Seguono alcune caratteristiche del design OO:

- **Gli oggetti sono indipendenti e autonomi:** sono astrazioni del mondo reale o di entità del sistema e si gestiscono autonomamente. Gli oggetti sono indipendenti e mantengono il loro stato.
- Le funzionalità sono espresse in termini di servizi degli oggetti.
- **I dati condivisi non esistono. Gli oggetti comunicano via scambio di messaggi.**
- Gli oggetti possono essere distribuiti e possono essere eseguiti in sequenza o in parallelo.

I vantaggi dei software/sistemi Object-Oriented sono: elevata manutenibilità e riusabilità.

I metodi di un oggetto dovrebbero essere in armonia; se un metodo di un oggetto sembra fuori luogo, meglio darlo sotto la responsabilità di un altro oggetto.

La progettazione Object-Oriented è applicata a due livelli:

- **Modelli strutturali:** classi, soggetti singoli, architetture di sistema, diagrammi di contesto.
- **Modelli comportamentali:** diagrammi di interazione (sequence diagrams), diagrammi di stato.

Segue il concetto di coesione e accoppiamento tra moduli/componenti:

Coesione e accoppiamento

La **coesione** è una **misura di quanto bene i componenti si incastrano tra loro.** È una misura del livello di correlazione tra diverse funzionalità presenti all'interno di un modulo, ossia il livello di **omogeneità funzionale.**

Un componente dovrebbe implementare una singola entità o funzione logica. Un componente dovrebbe contenere tutto e solo ciò che serve per implementare la relativa.

Esistono vari livelli di coesione:

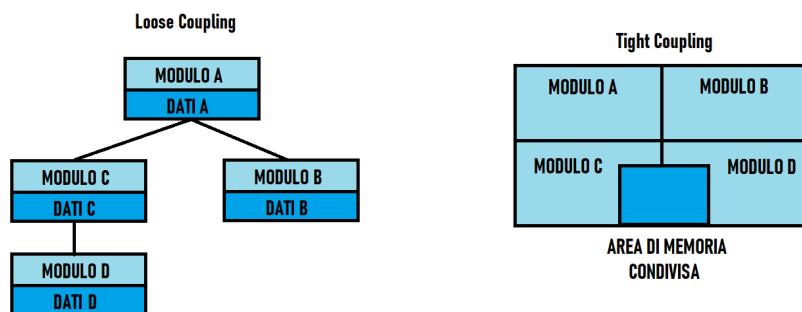
- **Weak (debole):**
 - Coesione coincidente (debole): parti di un componente sono semplicemente raggruppate insieme.
 - Associazione logica (debole): i componenti che svolgono funzioni simili sono raggruppati.
 - Coesione temporale (debole): i componenti attivati contemporaneamente vengono raggruppati.
- **Solid (solida):**
 - Coesione comunicativa (media): tutti gli elementi di un componente operano sullo stesso input o producono lo stesso output.
 - Coesione sequenziale (media): l'output per una parte di un componente è l'input per un'altra parte.

- **Strong (forte):**
 - Coesione funzionale (forte): ogni parte di un componente è necessaria (e sufficiente, tutto e solo, etc.) per l'esecuzione di una singola funzione.
 - Coesione degli oggetti (forte): ogni operazione fornisce funzionalità che consentono di modificare o ispezionare gli attributi degli oggetti.

L'accoppiamento è una misura della forza delle interconnessioni tra i componenti del sistema.

L'accoppiamento "allentato" (loose coupling) - ovvero interrelazioni scarse tra i componenti) significa che è improbabile che i cambiamenti dei componenti influenzino altri componenti. L'accoppiamento allentato può essere ottenuto mediante decentralizzazione di stato (come negli oggetti) e comunicazione tra componenti tramite parametri o passaggio di messaggi (ovvero non tramite variabile condivisa).

Le variabili condivise e il controllo dello scambio di informazioni portano a un accoppiamento stretto (tight coupling).



Ricapitolando: La coesione è una misura del livello di correlazione tra diverse funzionalità presenti all'interno di un modulo, ossia del suo livello di omogeneità funzionale. L'accoppiamento è una misura delle connessioni (dipendenze) tra due componenti del sistema. È importante progettare moduli con forte coesione perché questo indica un elevato livello di manutenibilità del sistema, ogni modifica è più rapida ed economica se svolta in un componente unico, invece che andare a modificare più componenti del sistema con il rischio di dover eseguire molte più modifiche. Un buon metodo per avere alta coesione nei moduli è applicare i principi solidi (principalmente il SRP). Avere basso accoppiamento indica che i moduli del sistema sono prevalentemente indipendenti e hanno scarse relazioni con altri moduli, questo basso livello di accoppiamento fa sì che una modifica non vada a impattare su altri moduli. Decentralizzando lo stato e facendo comunicare i moduli tramite messaggi (e non condividendo variabili) si raggiunge un basso livello di accoppiamento (loose coupling).

Principi SOLIDI del design OO

I principi solidi sono intesi come linee guida per lo sviluppo di software estendibile e mantenibile, in particolare nel contesto delle tecniche di sviluppo agile. Tra i cinque principi SOLIDI troviamo:

- Single Responsibility Principle (SRP)
 - Coesione e Accoppiamento
- Princípio dell'aperto-chiuso (OCP)
- Princípio di sostituzione di Liskov (LSP)
- Princípio di segregazione delle interfacce (ISP)
- Princípio di inversione della dipendenza (IDP)

Single Responsibility Principle (SRP)

Afferma che **ogni classe dovrebbe avere una ed una sola responsabilità**, interamente encapsulata al suo interno. In altri termini, ogni componente e/o modulo deve **implementare una singola funzionalità**, e quindi deve contenere tutto e solo il codice relativo a quella funzionalità e nulla di più.

Principio dell'aperto-chiuso (OCP)

Le **entità software** (classi, moduli, funzioni etc.) dovrebbero essere **aperte per l'estensione ma chiuse per le modifiche**.

Perché? una singola modifica in un modulo può produrre a catena degli effetti indesiderati su tutti i **moduli dipendenti**.

Lo scopo dell'OCP è di ridurre l'impatto di una modifica: idealmente, una modifica corrisponde a un nuovo concetto aggiunto, NON a una modifica a del codice funzionante.

Principio di sostituzione di Liskov (LSP)

I **sottotipi devono essere sostituibili per i loro tipi di base**, in ciascun contesto, per ogni programma.

Principio di segregazione delle interfacce (ISP)

Gli **utenti non dovrebbero essere forzati a dipendere dai metodi che non usano**.

Principio di inversione della dipendenza (DIP)

I **moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Entrambi dovrebbero dipendere dalle astrazioni**.

Progettazione per i test (design for testability - DFT)

La **testabilità** è il **grado in cui un artefatto software** (ad es. software, modulo, progetto, classe etc.) **supporta il testing in un certo contesto**. Se la **testabilità di un artefatto software è alta, trovare difetti nel sistema è più facile**. La testabilità include due fattori:

- **Controllabilità**: l'abilità di applicare input al sistema sotto test e di indurlo a un particolare stato.
- **Visibilità**: l'abilità di osservare gli stati in output.

La testabilità **non è una proprietà intrinseca dell'artefatto software e non può essere misurata direttamente**. Piuttosto, la testabilità è il risultato dell'interdipendenza del software testato e dello scopo finale, dal metodo utilizzato e dalle risorse.

Una **metrica per valutare la testabilità** è quanti test case sono necessari per ciascun caso per formare **un test completo**. Perché un software sia testabile, i requisiti devono essere:

- consistenti
- completi
- non ambigui
- quantitativi
- verificabili

Al fine di introdurre il corretto grado di testabilità in un software, **il software va progettato per la testabilità (design for testability - DFT)**. Le **tecniche DFT mirano ad aumentare l'abilità di un software di essere testato**. Alcune tecniche DFT più comuni sono:

- **Stratificazione**
- **Asserzioni** (test built-in)
- **Inversione di dipendenza**

LSP: Può essere rilevato con:

precondiz. più forte nella classe derivata,
invariante e postcond. più deboli nella classe derivata

50

ISP: lo si ha spesso quando si mettono metodi non comuni nelle interfacce e quindi alcune classi si ritrovano con metodi che non useranno mai.
(tutti i metodi nelle interfacce vanno implementati dalle classi che le sviluppano)

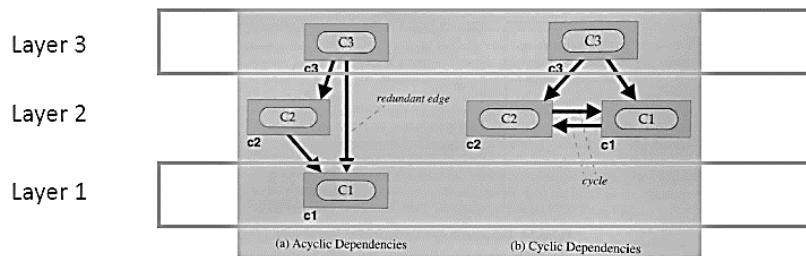
Stratificazione fisica e testing

Il **progetto fisico** (physical design) **descrive il layout delle dipendenze tra i componenti fisici del sistema**. Esso fornisce approfondimenti riguardo la concreta testabilità del sistema. L'idea è di eseguire **test per livello**, in diverse modalità possibili:

- **Testability in isolation**: un singolo livello alla volta, un singolo componente alla volta
- **Testing gerarchico** (hierarchical testing)
- **Testing incrementale** (incremental testing)

Progettazione aciclica

Le tecniche DFT forniscono approfondimenti per la costruzione di sistemi per singoli componenti che sono testabili in isolamento. **Se il grafo delle dipendenze di un sistema è aciclico, c'è almeno un ordine di test di tutti i componenti**: si parte dal layer fisico più basso e si procede fino a coprire ogni layer del sistema.

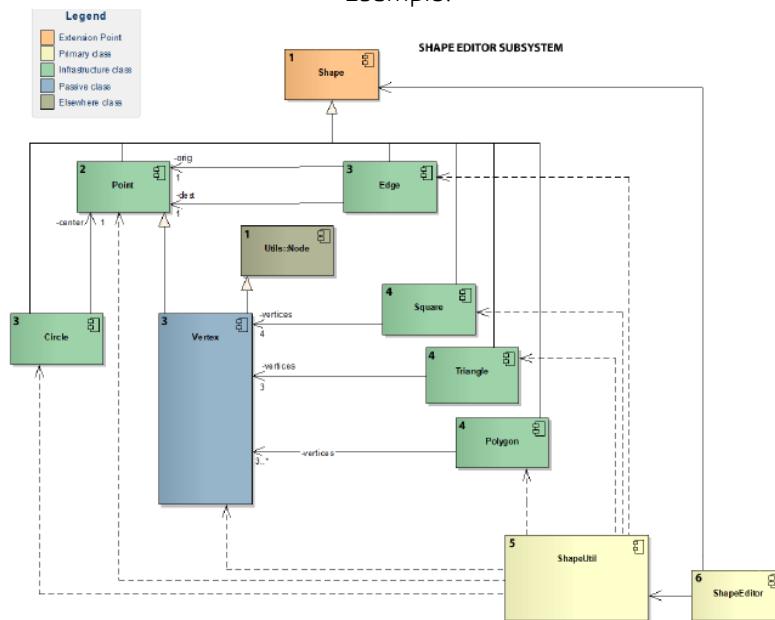


Definizione del numero di livello:

Livello: Il **livello** di un componente è la lunghezza del cammino più lungo dal grafico delle dipendenze **al componente esterno** (liv 0).

- livello 0: componente esterno.
- livello 1 (**componente foglia**): componente interno nel package corrente che non ha dipendenze con altri componenti all'interno del package. Un componente al livello 1 è una foglia e quindi testabile in isolamento
- livello N: componente che dipende almeno da un componente piazzato al livello N-1, ma che non dipende da componenti a livelli >N-1

Esempio:



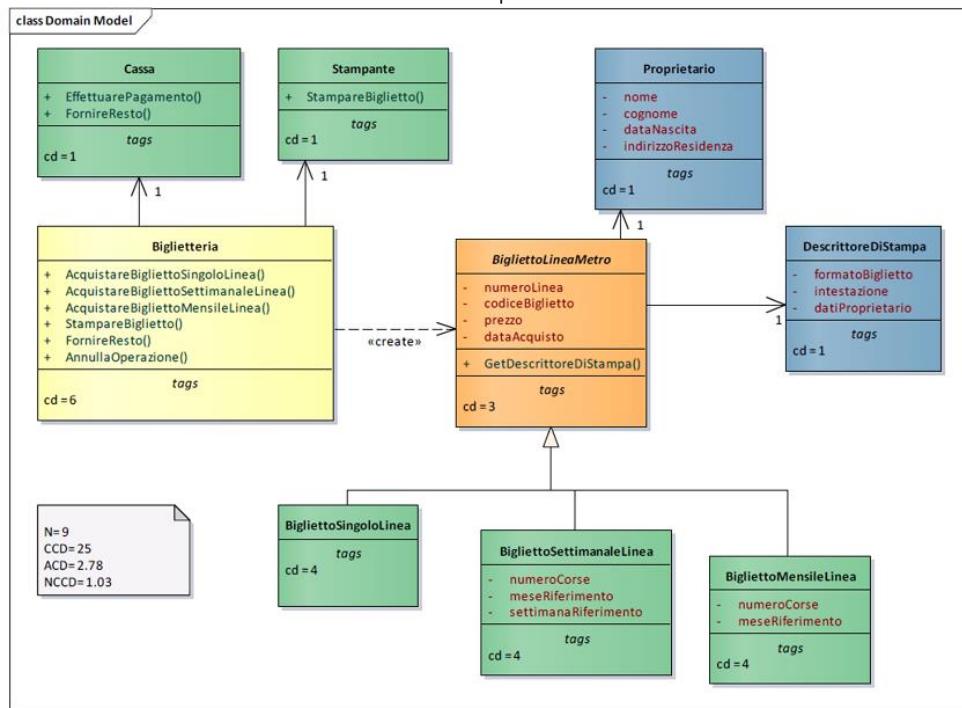
Component dependency and normalized metrics (LAKOS)

Metriche di Lakos:

- **CD**: component dependency: il livello di una classe
- **N**: numero di classi presenti nel sistema
- **CCD**: cumulative component dependency = somma delle CD delle N classi
- **ACD**: acyclic component dependency = CCD/N (la media)
- **NCCD**: normalized cumulative component dependency = CCD/CCD_{BBT}
 - o dove $CCD_{BBT} = (N+1) * \log_2(N + 1) - N$

Ricorda: se NCCD < 1 la struttura è orizzontale, se NCCD >> 1 allora sono presenti dei cicli nel sistema.

Esempio:



N = 9

CD: presenti nelle singole classi

CCD = 25

ACD = 25/9 = 2.78

CCD_{BBT} = (9+1) * log₂(9 + 1) - 9 = 24.22

NCCD = 25/24.22 = 1.03

Complessità ciclomatica CC

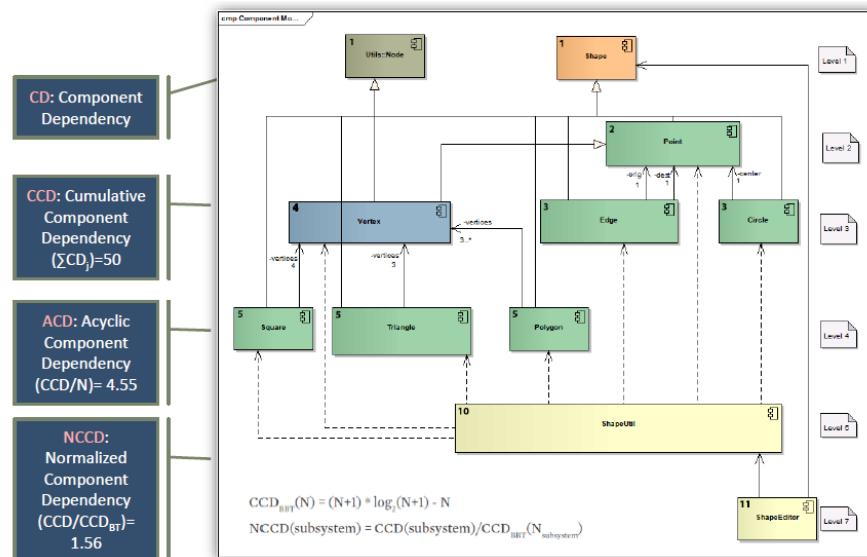
La **complessità ciclomatica** è una **metrica software** utilizzata per misurare la complessità strutturale **di un programma**. Tale metrica **misura direttamente il numero di cammini linearmente indipendenti** che caratterizzano il grafo del flusso di controllo di un programma.

Debito tecnico

La misura della testabilità appena illustrata è utile per la valutazione del debito tecnico.

Il **debito tecnico** è una metafora inventata per descrivere "le possibili complicazioni che subentrano in un progetto, tipicamente di sviluppo software, qualora non venissero adottate adeguate azioni volte a mantenerne bassa la complessità." Il debito tecnico (TD – Technical Debt) causa lavoro extra nel lungo periodo quando si è sotto pressione dalle deadlines.

Il debito tecnico è un aspetto inevitabilmente presente in un qualsiasi sistema software industriale.



Misurare il debito tecnico è importante perché esso aumenta i costi di gestione dei progetti. La fragilità di un progetto può essere ricondotta all'accumulo di debito tecnico poiché nel tempo non sono state prese le dovute contromisure per organizzare meglio la complessità strutturale dei moduli e delle procedure. Il debito tecnico nasce sempre da carenze organizzative, siano esse a livello di personale non qualificato oppure a livello di processo di sviluppo. Le principali cause del debito tecnico sono:

- Esigenze di mercato: deadlines stringenti;
- Mancanza di conoscenza del processo;
- Mancanza di disaccoppiamento: ignorando la proprietà di modularità o comunque senza l'intento di mantenere basso il livello di accoppiamento tra sottosistemi.
- Assenza di testing;
- Assenza di documentazione;
- Mancanza di collaborazione.

8. Test del software

L'obiettivo dei test del software è di **dimostrare che un programma svolge i compiti per i quali è stato realizzato e identificare eventuali errori prima di metterlo in uso**. Per provare il funzionamento di un software, si esegue un programma utilizzando dati artificiali. Si controllano i risultati dei test per verificare che non ci siano errori, anomalie o informazioni su attributi non funzionali del programma.

Il processo di test del software ha **due obiettivi** distinti:

1. **Dimostrare allo sviluppatore e al cliente che il software soddisfa i suoi requisiti**. Per il software personalizzato, questo significa che dovrebbe esserci almeno un test per ogni requisito specificato nel documento dei requisiti.
2. **Scoprire eventuali input o sequenze di input per i quali il comportamento del software è errato**, indesiderato o non è conforme alle sue specifiche. Tutto questo è causato da difetti (bug) nel software.

Il primo di questi obiettivi è il **test di convalida**, che **controlla che il sistema funzioni correttamente utilizzando una serie di test case che riflettono l'uso previsto del sistema**. Il secondo obiettivo è il **test dei difetti**, dove i test case sono progettati per scoprire difetti del software (input particolari etc.).

I test non possono dimostrare che il software sia privo di difetti o che si comporterà correttamente in qualsiasi circostanza. È sempre possibile che un test che viene trascurato possa svelare ulteriori problemi nel sistema. Come ha affermato Edsger Dijkstra:

"i test possono dimostrare soltanto la presenza di errori, non la loro assenza".

I test fanno parte di un processo più ampio: **verifica e convalida del software (V&V, Verification and Validation)**. La verifica e la convalida non sono la stessa cosa, anche se spesso vengono confuse:

- **Verifica**: stiamo realizzando il prodotto correttamente? (il programma è corretto?)
- **Convalida (o validazione)**: stiamo realizzando il prodotto corretto? (i requisiti sono soddisfatti?)

Il ruolo dei processi di verifica e convalida è essenzialmente quello di controllare che il software che si sta sviluppando soddisfi le sue specifiche e offra le funzionalità richieste da chi ha acquistato il software (gli stakeholder). Questi processi di controllo iniziano non appena i requisiti del software si rendono disponibili e proseguono per tutte le successive fasi del processo di sviluppo.

- **La verifica** del software è il processo che controlla se il software soddisfa i requisiti funzionali e non funzionali.
- **La convalida (o validazione)** è un processo più generale; il suo compito è garantire che il software rispetti le attese del cliente. Il suo scopo va ben oltre il semplice controllo di conformità alla specifica per dimostrare che il software fa ciò che il cliente ha richiesto. La convalida è essenziale perché le definizioni dei requisiti non sempre riflettono i reali desideri o necessità dei clienti.

La differenza è quindi che:

- L'attività di verifica fornisce l'obiettiva conferma che i risultati di progetto di un particolare componente soddisfano i requisiti specificati. Al contrario, la convalida del software conferma la conformità del prodotto software finale con i requisiti, aspettative e desideri dell'utente finale.
- La verifica viene eseguita nella fase di sviluppo mentre la convalida viene eseguita dopo che il prodotto è stato sviluppato (cioè dopo la verifica).

L'obiettivo ultimo dei processi di verifica e convalida è stabilire, con un buon grado di confidenza, che il software è adatto al suo scopo. Ciò significa che il sistema deve essere adatto all'uso cui è destinato.

VERIFICA E VALIDAZIONE

Verifica: vedo se il sw che ho costruito e' conforme a quanto richiesto nelle SPECIFICHE.

Validazione: il sw risolve i problemi che aveva l'utente (i veri requisiti), ovvero ho scelto specifiche giuste?

Esse hanno una GRANULARITA' diversa, validazione e' a livello di sistema (o sottosistema), mentre verifica e' a livello variabile (puo' essere a livello di singola linea di codice, o classe o sistema).

Verifica e validazione avvengono durante TUTTO IL CLICLO DI VITA DEL PROGETTO

Esistono due tipi di ATTIVITÀ:

- **STATICA**: viene svolta senza eseguire il codice. Sono le **ISPEZIONI**, che si appoggiano anche alle documentazioni e guardano il codice (senza eseguirlo).

Le ispezioni vengono fatte per trovare anomalie, riconoscere aspetti non corretti e difformita' da standard (grazie all'esperienza passata).

Applicabile in tutte le fasi di sviluppo.

- **DINAMICA**: richiede l'esecuzione del codice. E' il **TESTING** che serve per analizzare il **COMPORTAMENTO** del sw utilizzando dei data-test come input.

Nel testing non si deve correre bug (per questo c'e' il debugging), bensì si correggono errori LOGICI che riguardano incomprensioni con l'utente.

Applicabile solamente quando si ha un eseguibile a disposizione.

immagine

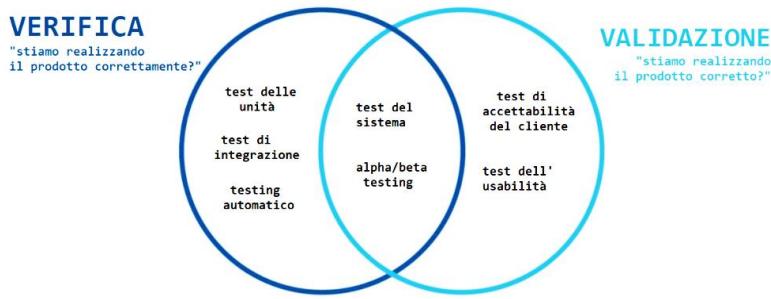
Questa fase non deve per forza eliminare tutti i difetti, deve invece assicurarsi che il sw prodotto sia quello giusto per il cliente. Questa fase può occupare più del 50% della fase di sviluppo, perciò va pianificata.

Il team di ispezione deve essere composto da almeno 4 membri:

- autore del codice che viene ispezionato
- ispettore
- lettore che legge il codice al team
- moderatore

Si usa una checklist di errori comuni per guidare l'ispezione.

Un altro metodo statico sono le **ANALISI STATICHE**. Vengono utilizzati dei tool per processare il codice del programma e rilevare dei bug (riconoscendo dei pattern).



Analogamente al test del software, anche i processi di verifica e convalida possono richiedere ispezioni e revisioni del software. Le ispezioni e le revisioni analizzano e controllano requisiti del sistema, gli schemi di progettazione, il codice sorgente dei programmi e perfino i test proposti per il sistema. Si tratta di tecniche statiche di V&V, nelle quali non occorre eseguire il software per verificarlo. L'ispezione del software ha tre vantaggi rispetto al testing:

1. Durante i test, gli errori possono nascondere altri errori. Gli errori causano altre interazioni erronee per via delle dipendenze di alcuni componenti.
2. Le versioni incomplete di un sistema possono essere ispezionate senza costi aggiuntivi, se invece si adotta il testing, bisogna sviluppare test specifici per il sistema incompleto; e questo ha un costo non trascurabile.
3. Oltre a cercare difetti di un programma, un'ispezione può anche valutare attributi come qualità e leggibilità del codice, conformità agli standard, portabilità e facilità di manutenzione; inefficienze e presenza di scelte errate di implementazione.

Le ispezioni sono un'idea vecchia: ci sono studi che dimostrano che sono più efficaci nella scoperta di bug nel codice. Le ispezioni tuttavia non possono rimpiazzare il test del software. Le ispezioni NON sono adatte a scoprire difetti che si presentano a causa di interazioni inaspettate tra varie parti del programma, i problemi tempistici o quelli relativi alle prestazioni.

Ispezioni e revisioni vengono approfondite [qui](#).

I **test case** sono le specifiche degli input dei test e degli output previsti dal sistema più una descrizione di ciò che si sta provando. I dati di test sono gli input che sono stati previsti per provare il sistema. I dati di test a volte possono essere generati automaticamente, ma non è possibile generare automaticamente un test case. Tipicamente un sistema commerciale è soggetto a **tre stadi di test**:

- **Test dello sviluppo:** include tutte le attività di test che sono svolte dal team di sviluppo di un sistema software. Il tester del software di solito è il programmatore che sviluppa il software. Ci sono tre stadi nei test di sviluppo:
 - **Test delle unità:** vengono provate singole unità di programma o classi di oggetti. Il test delle unità si focalizza sulla verifica delle funzionalità di oggetti e metodi singoli.
 - **Test dei componenti:** vengono integrate più unità per creare componenti complessi e successivamente ci si focalizza sulla verifica delle interfacce dei componenti che forniscono accesso alle funzioni dei vari componenti.
 - **Test del sistema:** alcuni o tutti i componenti del sistema vengono integrati e il sistema viene provato come un unico elemento distinto. Il test si focalizza sulla verifica delle interazioni fra i componenti integrati.

Le tre fasi/stadi nei/dei test di sviluppo vengono approfondite [qui](#).

- **Test della release:** Il test della release è il processo che testa una particolare release di un sistema che dovrà essere utilizzata all'esterno del team di sviluppo. Di norma, la release di un sistema è destinata agli utenti e clienti. Tuttavia, in un progetto complesso, la release potrebbe essere utilizzata da membri di altri team che sviluppano sistemi correlati. Il test della

release è un processo di convalida che controlla se un sistema soddisfa i suoi requisiti ed è sufficientemente buono per essere utilizzato dai clienti/utilizzatori del sistema. L'obiettivo principale del test della release è convincere il fornitore del sistema che questo è sufficientemente buono per essere utilizzato. Esistono tre approcci al test della release:

- **Test basato sui requisiti:** un principio generale della buona pratica di ing. del software è che i requisiti devono essere testabili. Questo significa che un requisito deve essere scritto in modo tale che possa essere progettato un apposito test case, dove si prendono in esame i singoli requisiti e si sviluppa una serie di test per ciascun requisito. Il test basato sui requisiti è una convalida, anziché un test dei difetti – si tenta di dimostrare che il sistema implementa correttamente i suoi requisiti.
- **Test basato sugli scenari:** il test degli scenari è un approccio al test della release dove vengono concepiti alcuni tipici scenari d'uso per sviluppare dei test case per un sistema. Uno scenario è una storia che descrivere il modo in cui il sistema potrebbe essere utilizzato e di conseguenza come le varie componenti interagiscono.
- **Test delle prestazioni:** una volta che il sistema è completamente integrato è possibile testare le proprietà più evidenti, come prestazioni e affidabilità. I test delle prestazioni devono essere progettati in modo tale da dimostrare che il sistema possa elaborare il carico di lavoro previsto. Questo test è particolarmente importante per i sistemi distribuiti.
- **Test degli utenti:** il test degli utenti è una fase del processo di test di un sistema in cui gli utenti o clienti danno i loro input e suggerimenti su test del sistema. Il test degli utenti è essenziale, anche dopo che sono stati completati i test del sistema e della release. I condizionamenti derivati dall'ambiente di lavoro degli utenti possono avere effetti rilevanti sulle prestazioni di affidabilità, utilizzabilità e robustezza di un sistema. Esistono tre test degli utenti principali:
 - **Alpha-test e Beta-test**
 - **Test di accettazione:** sono una parte intrinseca dello sviluppo dei sistemi personalizzati. I clienti di un sistema, utilizzando i loro dati, testano il sistema, e decidono se il software è accettabile o meno. L'accettazione implica che il pagamento finale del software può avvenire. Le sei fasi del test di accettazione sono:
 - Definire i criteri di accettazione;
 - Pianificare i test di accettazione;
 - Derivare i test di accettazione;
 - Eseguire i test di accettazione;
 - Negociare i risultati dei test;
 - Accettare o rifiutare il sistema.

In pratica, il processo di test consiste in un mix di test manuali e automatici e di vario tipo.

Fasi dei test di sviluppo

Oltre a quello detto precedentemente, possiamo dire che il test di sviluppo è un processo il cui scopo principale è scoprire bug del software. Di conseguenza, questo test di solito è interlacciato con il debugging (processo che localizza i problemi nel codice e modifica il programma per risolvere tali problemi). Segue una descrizione più dettagliata delle tre fasi di test nei test di sviluppo:

Test delle unità

I tipi più semplici di componenti sono le singole funzioni, i singoli metodi di un programma. I test dovrebbero eseguire queste routine con differenti parametri di input. Per testare le classi di oggetti, si dovrebbero progettare test in modo da verificare le funzioni di ciascun oggetto. Ove possibile è consigliabile automatizzare il test delle unità. Un test automatico è composto da tre parti:

- Parte di impostazione: viene inizializzato il sistema con il test case, ovvero input e output previsti.
- Parte della chiamata: viene chiamato l'oggetto o il metodo da testare.

- Parte dell'asserzione: viene confrontato il risultato della chiamata con il risultato previsto.

Il test delle unità è un processo lungo e costoso, quindi è importante scegliere test case molto efficienti. Con efficienza si intende:

- I test case devono mostrare che se utilizzato come previsto, il componente svolge correttamente il compito per il quale è progettato.
- Se ci sono difetti nel componente, questi devono emergere con l'uso del test case.

Per questo bisogna progettare due tipologie di test case: il primo di questi dovrebbe riflettere il normale funzionamento del sistema e mostrare che esso funziona. L'altro tipo di test dovrebbe essere basato sull'esperienza che suggerisce dove si presentano i problemi più comuni; ad esempio utilizzando input particolari. Esistono due strategie per la scelta dei test case:

- Test delle partizioni
- Test basati su linee guida

Test delle componenti

I componenti software spesso sono formati da molti oggetti singoli che interagiscono tra di loro. L'accesso alle funzionalità di questi oggetti avviene tramite le interfacce dei componenti. Il test dei componenti composti deve dimostrare che le interfacce dei componenti funzionino correttamente. È molto probabile che gli errori nelle interfacce dei componenti non vengano individuati testando i singoli oggetti, in quanto tali errori riguardano l'interazione tra di essi. Ci sono varie tipologie di interfacce, le più comuni sono:

- Interfacce di parametri;
- Interfacce a memoria condivisa;
- Interfacce procedurali;
- Interfacce a passaggio di messaggi.

Di conseguenza ci possono essere diversi tipi di errori, classificabili in:

- Uso errato dell'interfaccia;
- Incomprensione delle interfacce;
- Errori di tempistica.

I test dei difetti delle interfacce sono difficili perché alcuni errori possono manifestarsi soltanto sotto condizioni inusuali di interazione. Un problema ulteriore può sorgere a causa delle interazioni tra gli errori nei vari moduli o oggetti. Gli errori di un oggetto possono essere individuati soltanto quando qualche altro oggetto si comporta in modo inaspettato. Se il servizio ricevuto è sbagliato, il valore restituito può essere valido, ma errato. Il problema in questo caso non sarà immediatamente rilevato, ma si manifesterà più tardi. In alcuni casi è preferibile al posto dei test, eseguire revisioni statiche per trovare errori nelle interfacce.

Test del sistema

Lo sviluppo di un sistema richiede l'integrazione di più componenti nel sistema e poi il test del sistema integrato. Il test del sistema controlla che i componenti siano compatibili, che interagiscano correttamente e che scambino i dati appropriati al momento giusto tra le loro interfacce. Ovviamente questo processo si sovrappone al test dei componenti. Alcune funzionalità e caratteristiche del sistema diventano evidenti solo quando tutti i componenti vengono assemblati. Bisognerebbe sviluppare i test per controllare che il sistema svolga soltanto i compiti previsti. Il test del sistema dovrebbe focalizzarsi sulle interazioni tra i componenti e gli oggetti che formano il sistema. A causa della loro attenzione alle interazioni, i test basati sui casi d'uso sono molto efficaci per testare un sistema. Il test automatico di un sistema è di solito più difficile da realizzare.

Sviluppo guidato da test

Lo sviluppo guidato da test è un approccio allo sviluppo dei programmi in cui si interallacciano sviluppo e test del codice. Il codice viene sviluppato in modo incrementale, insieme a una serie di test per ogni incremento del codice. Non si potrà procedere allo sviluppo del successivo incremento finché il codice non avrà superato tutti i suoi test. Lo sviluppo guidato da test venne introdotto come parte del metodo di sviluppo agile extreme programming XP. Le fasi di questo processo sono elencate di seguito:

1. Si inizia identificando l'incremento delle funzionalità richieste.
2. Si scrive un test per queste funzionalità e lo si implementa come test automatico.
3. Si esegue il test.
4. Si implementano le funzionalità e si ripete il test finché non va a buon fine.
5. Si passa all'implementazione dell'incremento successivo se tutti i test vanno a buon fine.

Con questo approccio è possibile eseguire centinaia di test distinti in pochi secondi. Lo sviluppo guidato da test aiuta i programmatore a fare chiarezza su cosa un segmento di codice deve effettivamente fare. Per scrivere un test, occorre capire a cosa serve, e questa conoscenza aiuta a scrivere il codice richiesto. Se non si hanno sufficienti conoscenze per scrivere i test, non è possibile sviluppare codice richiesto. Oltre a una migliore comprensione dei problemi, lo sviluppo da test offre altri vantaggi:

1. Copertura del codice;
2. Test di regressione;
3. Debugging semplificato;
4. Documentazione del sistema.

Uno dei più importanti vantaggi dello sviluppo guidato da test è la riduzione dei costi del test di regressione. Il **test di regressione** richiede l'esecuzione di una serie di test che sono stati eseguiti con successo dopo le modifiche apportate a un sistema. Il test di regressione controlla che queste modifiche non abbiano introdotto nuovi bug nel sistema e che il nuovo codice interagisca secondo le previsioni con il codice esistente. Il test di regressione è costoso e a volte impraticabile quando un sistema viene testato manualmente, in quanto i costi in termini di tempo e impegno sono molto elevati. I test automatici riducono notevolmente i costi del test di regressione. I test esistenti possono essere eseguiti nuovamente rapidamente e a un buon prezzo dopo aver apportato una modifica.

Altro sul testing:

Il **black-box testing** è un metodo di test del software che esamina la funzionalità di un'applicazione senza scrutare nelle sue strutture o funzionamenti interni (tipico dei sistemi legacy).

Equivalence partitioning: Il partizionamento di equivalenza o il partizionamento di classe di equivalenza (ECP) è una tecnica di test del software che divide i dati di input di un'unità software in partizioni di dati equivalenti da cui è possibile derivare casi di test. In linea di principio, i casi di test sono progettati per coprire ogni partizione almeno una volta.

Il **profilo operativo** del software riflette come questo verrà utilizzato nella pratica. Consiste in una specifica delle classi di input e nella probabilità che questi avvengano. I **test statistici** consentono di concentrarsi sui test di quegli elementi del sistema che presumibilmente devono essere utilizzati.

TIPI di TESTING:

- **STATISTICAL**: viene fatto per dare una misura approssimativa dell'affidabilita'.

Si stabilisce un **PROFILO OPERATIVO** (ovvero un set di input), si eseguono le misure e si usano i dati raccolti per calcolare indici di predizione (es. ROCOF, MTBF, ...)

- 1. **DEFECT** (verification): per scoprire i difetti nei programmi. Puo' rilevare la presenza di failure, ma NON la loro ASSENZA.

Si progettano dei **TEST-CASE** (e' la coppia <input, output atteso>), si effettuano i test e si verificano quali portano a failure, una volta fatto cio si passano i dati raccolti ai debugger che provano a risolvere le failure.

Il test va a buon fine se rileva dei failure.

Il TEST-CASE e' un'insieme di input che puo' risultare inestato e volto soltanto a rilevare errori (quindi lontano dal comportamento dell'utente).

- 2. **VALIDATION**: dimostro al cliente che il sw rispetta i requisiti (requisiti, non specifiche, quindi caratteristiche funzionali, non tecniche) e che funziona come dovrebbe.

Il test va a buon fine se non rileva dei failure.

Il TEST-CASE e' un insieme di input che riflettono il possibile comportamento dell'utente e di come il sistema andrebbe usato.

Verification e validation testing vengono fatti per trovare dei failure, mentre il debugging viene fatto per localizzare e risolvere gli errori.

Dimostrare con il testing che un sw e' esente da errori e' impossibile, non si puo' sapere se e' esente da errori che non conosciamo (e quindi non testiamo).

1. All'interno del **DEFECT** sono compresi:

- **UNIT/COMPONENT TESTING**: Si inizia a testare le singole componenti analizzando prima le singole funzioni e poi integrandole (funzioni) tra loro testo le componenti.

- **BLACK BOX**: considero il programma come black box e cerco solo di testare le sue funzionalita'.

I TEST-CASE mirano a verificare le funzionalita' e sono sia input normali che input anomali (basandomi sull'esperienza per vedere se riesco a "rompere il sistema").

Con l'**EQUIVALENCE PARTITIONING** divido in classi (di equivalenza) i test-data in modo da visualizzare quali input sono normali e quali borderline (ovvero input che l'utente non immetterebbe mai).

- **STRUCTURAL**:

- **WHITE BOX**: guardo come e' fatto il programma all'interno e cerco di testare tutti i controlli all'interno del programma usando test-case aggiuntivi.

- **PATH**: costruisco un grafo i cui nodi rappresentano le decisioni e gli archi il flusso e cerco di creare dei test-case in modo da coprire ogni cammino possibile del programma almeno una volta. L'indice di complessita' ciclomatica (numero archi+numero nodi+2) rappresenta la complessita' procedurale del sw e quindi il numero di test che devo fare per testare tutti i casi possibili (e quindi non tutte le combinazioni avvenibili -> tolgo combinazioni che non avverrebbero mai).
- **INTERFACE**: testo le interfacce che servono per fare interagire tra di loro i singoli moduli. Testo se la loro integrazione funziona o meno.
Ad es. faccio uno stress testing nello scambio di messaggi tra i moduli (o sottosistemi).
- **STRESS**: Stresso il sistema oltre il suo carico massimo per vedere se emergono difetti.
- **INTEGRATION/SYSTEM/USER TESTING**: Ho tutte le componenti (o la maggior parte) integrate tra loro e testo l'intero sistema (o sottosistema) se funziona correttamente.
 - **INCREMENTAL**: Testo anche le proprieta' emergenti. So che le singole componenti funzionano, a chi do la colpa allora? aggiungo piccoli incrementi alla volta e ri-testo tutto.
 - **REGRESSION**: e' necessario ripetere ad ogni step incrementale tutti i test eseguiti al passo incrementale precedente.
Cio' e' necessario per verificare che i test già fatti in precedenza continuino a non creare problemi con le nuove aggiunte.
 - **TOP DOWN**: parto dal livello alto e integro mano a mano. Per le componenti che mancano ci sono le STUB che simulano quello che c'e' sotto (assumendo che sotto tutto funzioni) anche se magari non e' ancora stato implementato.
 - **BOTTOM UP**: parto dal livello più basso e ogni volta aggiungo una componente e i COMPONENT DRIVER controllano le componenti (di livelli inferiori già passati) passandogli un test case e verificando che sotto tutto funzioni.
- **ACCEPTANCE**: Processo formale in cui lo sviluppatore dimostra di aver realizzato quanto richiesto, eseguendo una serie di test.
 - **RELEASE TESTING**: controllare che tutto il sistema funzioni, di solito viene fatto black box (ovvero i tester sono persone esterne che non conoscono l'implementazione del sistema) ed e' funzionale (ovvero verifico le funzionalita').
 - **PERFORMANCE TESTING**: verifico le proprietà (non funzionali) emergenti del sistema, es. la performance.
 - **BACK TO BACK**: quando un vecchio sistema viene sostituito con un nuovo sistema che ha alcune funzionalita' simili a quelle del precedente.
 - **INSTALLATION**: eseguito quando il sistema collaudato e' stato installato nell'ambiente operativo.

2.All'interno del **VALIDATION** sono compresi:

- **REQUIREMENTS BASED**: per ogni requisito creo un data set da testare.
- **USER**: l'utente mi fornisce degli input e provo a testare il sistema (con quegli input).

Si suddivide in:

- **ALPHA TESTING**: ottica utente ma non inserito nell'ambiente finale.
- **BETA TESTING**: ottica utente ma inserito nell'ambiente finale.

9. Evoluzione del software

I grandi sistemi software di solito hanno una lunga vita. Il costo di questo software è molto elevato, quindi le aziende utilizzano un sistema software per molti anni in modo da avere un ritorno sugli investimenti. Durante la loro vita, i sistemi software devono cambiare se vogliono restare utili. I cambiamenti delle aziende e delle aspettative degli utenti generano nuovi requisiti per il software. Parti del software potrebbero richiedere delle modifiche per correggere gli errori che si sono scoperti durante il loro utilizzo, per essere adattate a nuove unità hardware e piattaforme software, e per migliorare le loro prestazioni o altre caratteristiche non funzionali.

I sistemi software, quindi, si adattano e si evolvono durante il loro ciclo di vita, dal rilascio della prima versione fino all'ultima versione. Alcuni dati statistici indicano che i costi di evoluzione sono compresi tra il 60% e il 90% dei costi totali del software. L'ingegneria del software si può così immaginare come un processo a spirale, in cui la specifica dei requisiti, la progettazione, l'implementazione e i test continuano durante tutta la vita del sistema. Si inizia creando la release 1 del sistema; una volta consegnata, si propongono nuove modifiche e quasi immediatamente inizia lo sviluppo della release 2. In effetti, la necessità di aggiornare il software può verificarsi ancora prima della consegna del sistema, cosicché la successiva release del software può iniziare il suo sviluppo ancora prima che la versione iniziale sia stata rilasciata.

L'evoluzione del software personalizzato, invece, di solito segue un modello differente. Il cliente del sistema potrebbe pagare una società per sviluppare il software e poi assumersi la responsabilità del supporto tecnico e dell'evoluzione utilizzando il proprio personale. In questo caso spesso ci sono discontinuità nel processo evolutivo. I documenti dei requisiti e di progettazione potrebbero non essere trasmessi da una società all'altra; le società potrebbero fondersi o riorganizzarsi, ereditare il software da altre società, e scoprire poi che questo software deve venir modificato. Quando la transazione dallo sviluppo all'evoluzione non è continua, il processo di modifica del software dopo la sua consegna si chiama **manutenzione del software**.

Processi evolutivi

Come tutti i processi software, non esiste un processo evolutivo standard o un processo di modifica standard per il software. Il processo evolutivo più appropriato per un sistema software dipende dal tipo di software che si sta mantenendo, dai processi di sviluppo utilizzati in un'azienda e dall'esperienza delle persone coinvolte nel processo. Le proposte di modifica del sistema, formali o informali, sono la guida per l'evoluzione dei sistemi in tutte le aziende. In una proposta di modifica, uno o più individui possono suggerire modifiche e aggiornamenti per un sistema software esistente. Queste proposte possono includere requisiti esistenti che non sono stati implementati nella precedente release, o nuovi requisiti e correzioni di bug etc. Prima che una proposta di modifica venga accettata occorre eseguire un'analisi del software per stabilire quali componenti devono essere modificati. Questo fa parte del processo generale di **gestione delle modifiche**, che dovrebbe garantire anche che in ogni release del sistema siano incluse le versioni corrette dei componenti. Il processo include le attività fondamentali di analisi delle modifiche, pianificazione delle release, implementazione del sistema e rilascio di un sistema ai clienti. Vengono stimati il costo e l'impatto di una modifica per sapere quanto il sistema viene influenzato dalla modifica e quanto potrebbe costare l'implementazione. Se le modifiche proposte vengono accettate, viene pianificata una nuova release del sistema. Se la specifica dei requisiti e i documenti di progettazione sono disponibili, questi dovrebbero essere aggiornati durante il processo evolutivo per riflettere le modifiche che sono state richieste. Le modifiche proposte possono essere prototipate come parte del processo di analisi delle modifiche, quando vengono definite le implicazioni e i costi per apportare le modifiche.

Sistemi ereditati (legacy systems)

I legacy system sono tutti quei vecchi sistemi ancora in uso che giocano un ruolo critico nella gestione delle aziende. I sistemi ereditati si basano su tecnologie e linguaggi che non sono più utilizzati nello sviluppo di nuovi sistemi. Essi non sono semplicemente sistemi software, ma sistemi sociotecnici più complessi che includono hardware, software, librerie, processi di supporto al software e alla gestione aziendale. Le parti logiche di un legacy system sono:

- Hardware del sistema;
- Software di supporto;
- Software applicativo;
- Dati delle applicazioni;
- Processi aziendali;
- Politiche e regole aziendali.

Questi programmi funzionano ancora oggi in modo efficiente, e le aziende che li usano non sentono la necessità di cambiarli. Tuttavia, una problematica da affrontare è la carenza di programmatore che conoscano questi vecchi linguaggi, di conseguenza le modifiche sono costose; inoltre, sono presenti problemi di vulnerabilità e di sicurezza. Un altro possibile problema è che l'hardware del sistema potrebbe essere obsoleto e quindi, sempre più costoso da mantenere. Le aziende non sostituiscono questi sistemi con sistemi moderni perché tale sostituzione sarebbe troppo costosa e troppo rischiosa. Se un sistema legacy funziona bene, i costi di sostituzione potrebbero superare i risparmi che si otterrebbero dalla riduzione dei costi di supporto del nuovo sistema. Scartare un sistema legacy con uno moderno metterebbe a rischio il corretto funzionamento delle operazioni e il nuovo sistema potrebbe non soddisfare le esigenze di un'azienda.

Manutenzione del software

La manutenzione del software è il processo generale di modifica di un sistema dopo che è stato consegnato. Il termine viene solitamente applicato al software personalizzato, dove operano gruppi di sviluppo separati prima e dopo la consegna. Le modifiche apportate al software possono essere semplici correzioni di errori nel codice, correzioni più consistenti di errori di progettazione o miglioramenti significativi per correggere errori della specifica o per adattare nuovi requisiti. Le modifiche sono implementate modificando i componenti esistenti del sistema, e se necessario, aggiungendone di nuovi.

Ci sono tre tipologie di manutenzione del software:

- Eliminare errori e punti vulnerabili del software (**manutenzione correttiva**)
- Adattare il software a nuovi ambienti operativi e piattaforme (**manutenzione adattiva**)
- Aggiungere nuove funzionalità o supportare nuovi requisiti (**manutenzione perfettiva**)

In pratica, non c'è una netta distinzione tra questi tipi di manutenzione. L'esperienza ha dimostrato che di solito è più costoso aggiungere una nuova funzionalità a un sistema durante la manutenzione, mentre è più conveniente implementare la stessa funzionalità durante la fase dello sviluppo, per le seguenti ragioni:

- Un nuovo team deve capire il programma da manutenere.
- Separare la manutenzione dallo sviluppo significa che il team di sviluppo non è incentivato a scrivere un software facile da manutenere.
- La manutenzione dei programmi è un compito impopolare.
- Quando un programma invecchia, la sua struttura si deteriora e diventa più difficile modificarlo.

Il lavoro svolto durante lo sviluppo per strutturare il software e renderlo più comprensibile e più facile da modificare ridurrà i costi di evoluzione. Le buone tecniche di ingegneria del software, quali le specifiche accurate, lo sviluppo con test iniziali, lo sviluppo orientato agli oggetti e la gestione della configurazione, contribuiscono a ridurre i costi di manutenzione.

Previsione della manutenzione

La previsione della manutenzione si occupa di tentare di stabilire le probabili modifiche che potrebbero essere richieste da un sistema software e identificare le parti del sistema che probabilmente saranno le più costose da modificare. Se si capisce questo, è possibile progettare le componenti software che più probabilmente dovranno essere modificate per renderle più adattabili e mantenibili. Prevedendo le modifiche, inoltre, è possibile stabilire i costi di manutenzione complessivi di un sistema in un dato periodo.

Reingegnerizzazione del software

Per rendere più facile la manutenzione dei legacy systems si può decidere di reingegnerizzare questi sistemi per migliorarne la struttura e la comprensibilità. La reingegnerizzazione del software potrebbe richiedere una nuova documentazione del sistema, l'esecuzione di refactoring dell'architettura, la traduzione di programmi in un linguaggio più moderno, la modifica e aggiornamento della struttura e dei dati del sistema. La reingegnerizzazione del software ha tipicamente costi e rischi inferiori rispetto alla sostituzione del software. Le attività principali della reingegnerizzazione sono:

1. Traduzione del codice sorgente;
2. Reverse engineering;
3. Perfezionamento della struttura del programma;
4. Modularizzazione del programma;
5. Reingegnerizzazione dei dati.

10. Sistemi fidati

Poiché i sistemi che fanno uso intensivo del software sono molto importanti per le aziende pubbliche e private e per i singoli utenti, è necessario che questi sistemi siano affidabili. Il software dovrebbe essere disponibile quando è richiesto e dovrebbe funzionare correttamente senza effetti collaterali indesiderati, come ad esempio la divulgazione non autorizzata di informazioni. In poche parole, dovremmo essere in grado di fidarci dei nostri sistemi software.

Il termine **fidatezza** (dependability) fa riferimento alle quattro principali caratteristiche dei sistemi software (più una aggiunta successivamente alla definizione):

- **Disponibilità**: informalmente, indica la probabilità che un sistema sia attivo e in grado di fornire servizi utili in qualsiasi momento. Il sw funziona quando serve?
- **Affidabilità**: informalmente, è la probabilità che, in un determinato periodo di tempo, un sistema sia in grado di fornire correttamente i servizi secondo le aspettative dell'utente. —> Il sw funziona bene o male, fa o non fa quello che dovrebbe fare?
- **Sicurezza**: informalmente, è la stima delle probabilità che il sistema possa causare danni alle persone o all'ambiente. Il sw funziona senza causare danni a cose o persone?
- **Protezione**: informalmente, è la stima delle probabilità che il sistema possa resistere a intrusioni accidentali o deliberate. Il sw funziona protetto evitando danni ai suoi dati?
- **Resilienza**: informalmente, è la stima delle probabilità che il sistema possa garantire la continuità dei servizi critici al verificarsi di eventi dannosi, come il guasto di un componente o un attacco cibernetico. Questa proprietà è stata aggiunta alle 4 originariamente consigliate. —> L'abilità del sistema di resistere e recuperare da eventi dannosi

Il termine fidatezza (dependability) serve a indicare tutte queste caratteristiche appena citate. La fidatezza è la funzionalità più importante di un sistema software per varie ragioni:

- I guasti di un sistema riguardano un gran numero di persone;
- Gli utenti spesso rifiutano i sistemi non affidabili, non sicuri e non protetti;
- I costi per un guasto di un sistema software potrebbero essere enormi;
- I sistemi inaffidabili possono causare perdite di informazioni.

Sistemi critici: classe di sistemi software i cui guasti potrebbero causare danni alle persone o all'ambiente o ingenti perdite economiche. I sistemi critici sono molto costosi da sviluppare. In questi sistemi la fidatezza è un parametro fondamentale.

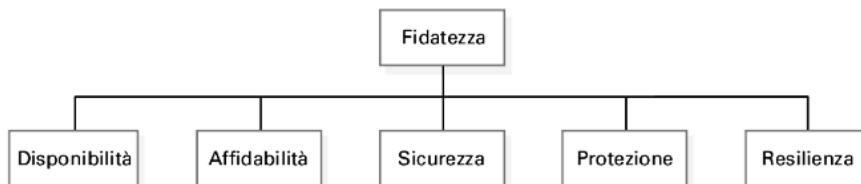
Quando si progetta un sistema fidato, occorre quindi considerare le probabilità che si verifichino:

- Guasti dell'hardware
- Guasti del software
- Errori di utilizzo

Se si progettano separatamente hardware, software e processi operativi, senza tener conto delle potenziali debolezze delle altre parti del sistema, è più probabile che si verifichino degli errori tra le interfacce delle varie parti del sistema.

Fidatezza in quanto proprietà

La fidatezza di un sistema software è una proprietà che rispecchia il livello di **fiducia** che l'utente ripone nel sistema. In questo contesto, l'affidabilità indica essenzialmente il grado di fiducia dell'utente sul corretto funzionamento del sistema e sul fatto che il sistema non fallirà durante il normale utilizzo.



DEPENDABILITY (fideiussa)

Sempre di più la nostra società delega al SW task rilevanti e critici, perciò bisogna garantire una corretta esecuzione.
Per i SISTEMI CRITICI è il parametro più importante.

Ci sono 4 dimensioni di dependability:

- **Disponibilità**: è pronto il sw nel momento del bisogno? (MISURA IN PERCENTUALE) -> probabilità che il sistema in un certo istante sia operativo e funzionante.
- **Affidabilità**: quanto bene svolge il suo compito il sw? fa quanto richiesto, funziona bene o male? (MISURA IN PERCENTUALE)
-> probabilità che un sistema in un certo istante esente da malfunzionamenti.
- **Sicurezza**: causa danni a persone o a cose il mio sw? (STIMA PERCENTUALE)
- **Protezione**: il sw causa danni a se stesso (toccando i dati)? (STIMA PERCENTUALE)

Ci sono ulteriori dimensioni:

- **Resilienza**: l'abilità del sistema di resistere e rigenerarsi da eventi danneggianti.
- **Manutenibilità**: quanto è facile riparare il sistema dopo un guasto? o cambiarlo per introdurre nuove features? (MISURA IN PERCENTUALE)
- **Survivability**: abilità del sistema di continuare a rilasciare il proprio servizio durante un attacco.

Potremmo dire

- I Includendo anche la maintainability, ciò che desideriamo è quindi un **sistema che funzioni nel modo che ci attendiamo (correttamente e consistentemente), senza creare danni all'esterno o a se stesso, per lunghi periodi tra una manutenzione e l'altra**. In più è importante che sia operativo e funzionante nel momento in cui ci serve e che sia riparabile velocemente e facilmente quando evidenzia dei malfunzionamenti.

[**reliability, availability, security, safety, maintainability**]

↳ facilità di riparazione e
modifica

I costi di DEPENDABILITY tendono ad incrementare esponenzialmente, infatti per raggiungere livelli più alti di dependability sono richiesti:

- maggiore livello di testing e validazione -> richiede molto tempo
- uso di tecniche implementative più costose e di hw più performante.

Dependability e performance sono INVERSAMENTE PROPORZIONALI.

Esistono vari tipi di guasti:

- **Malfunzionamenti**: (visione esterna) **evento inatteso**, servizio non conforme a ciò che l'utente si aspetta.
Il **TESTING** serve ad identificare i malfunzionamenti.
- **Errori di sistema**: (visione interna) **comportamento interno diverso da quello atteso**. Errore di sist. provoca malfunzionamento.
- **Difetto**: Anomalia del codice (**bug**), il bug può portare ad errori di sistema. Il **DEBUGGING** serve a trovare i difetti.
- **Errore umano**: **distrazione**, etc. Può portare a difetti.

La cosa importante è la **Affidabilità** che l'utente percepisce.

La Reliability può anche essere vista come la probabilità che un input specifico venga mappato in un output non desiderato.

La **Reliability** (affidabilità) viene raggiunta con:

- **Avoidance**: si cerca di evitare con accorgimenti durante il processo di sviluppo (es. con controlli).
- **Detection e Removal**: si cerca di **rimuovere prima di mettere in produzione** -> bisogna rilevare gli errori.
- **Tolerance**: si cerca di evitare con codice ad hoc nel sistema sviluppato

Rimuovere il x% di guasti non assume che il sistema guadagni x% reliability

-> magari rimuovo bug in sezioni di codice che non vengono mai usate (o usate di rado) dall'utente.

Le proprietà della fidatezza sono caratteristiche complesse che possono essere suddivise in caratteristiche più semplici. Per esempio, la protezione include integrità (garanzia che i programmi e i dati del sistema non vengano danneggiati) e riservatezza (garanzia che le informazioni siano accessibili soltanto alle persone autorizzate). L'affidabilità include la correttezza (la garanzia che i servizi di sistema siano quelli specificati), la precisione (garanzia che le informazioni siano fornite con un corretto livello di dettaglio) e la puntualità (garanzia che le informazioni siano fornite quando richiesto).

Ovviamente, non tutte le proprietà della fidatezza sono critiche per tutti i sistemi.

Altre proprietà strettamente legate alla fidatezza di un sistema software sono:

- Riparabilità;
- Manutenibilità;
- Tolleranza agli errori.

Per sviluppare un software fidato occorre quindi:

1. Evitare di introdurre errori accidentali nel sistema durante la specifica e lo sviluppo del software;
2. Progettare processi di verifica e convalida in grado di identificare errori residui che influiscono sulla fidatezza del sistema;
3. Progettare un sistema tollerante agli errori in modo che possa continuare a operare anche quando qualcosa si guasta;
4. Progettare meccanismi di protezione contro attacchi esterni che possono compromettere la disponibilità o protezione del sistema;
5. Configurare il sistema e il software di supporto correttamente per il loro ambiente operativo;
6. Includere alcune funzionalità in grado di riconoscere e resistere a eventuali attacchi cibernetici esterni;
7. Progettare il sistema in modo che possa riprendere il suo corretto funzionamento dopo un guasto o un attacco cibernetico senza perdere dati critici.

La tolleranza ai guasti prevede ridondanza nel sistema, questo ha un impatto sulle performance. I progettisti devono quindi scegliere un giusto compromesso tra fidatezza e prestazioni. La fidatezza porta a costi aggiuntivi nelle fasi di progettazioni e testing.

Sistemi sociotecnici

I sistemi sociotecnici sono quei sistemi dove non è presente solo l'hardware e il software, ma anche elementi come persone, processi e regolamenti aziendali. La fidatezza di un sistema è influenzata da tutti gli aspetti in un sistema sociotecnico – hardware, software, persone e organizzazioni.

Questi sistemi sono così complessi che è meglio considerarli come un insieme di strati

1. Strato delle apparecchiature;
2. Strato del sistema operativo;
3. Strato delle comunicazioni e gestione dei dati;
4. Strato delle applicazioni;
5. Strato dei processi aziendali;
6. Strato organizzativo;
7. Strato sociale.

Occorre avere, quindi, una visione globale del sistema quando si progetta un software che deve essere affidabile e protetto. Occorre prendere in considerazione le conseguenze dei malfunzionamenti del software su altri elementi del sistema.



Occorre quindi garantire che:

- I malfunzionamenti del software siano, per quanto possibile, contenuti nello stesso strato in cui si verificano, evitando che influiscano sul funzionamento di altri strati dello stack del sistema.
- I guasti e i malfunzionamenti di altri strati dello stack del sistema non influiscano sul software che si sta sviluppando.

Ridondanza

I guasti dei componenti in qualsiasi sistema sono pressoché inevitabili. Le persone commettono errori; bug nascosti nel software provocano comportamenti indesiderati; l'hardware a volte fallisce. Si utilizzano una serie di strategie per ridurre al minimo il numero di errori umani. Tuttavia, non possiamo avere la certezza che queste strategie eliminino completamente il rischio di guasti. Per questo occorre progettare i sistemi in modo che il malfunzionamento di un singolo componente non provochi il blocco di tutto il sistema.

Le strategie per ottenere e migliorare la fidatezza si basano sia sulla **ridondanza** sia sulla **diversità**.

Ridondanza significa che occorre includere delle capacità di riserva in un sistema che può essere utilizzato quando una sua parte fallisce. Diversità significa che i componenti ridondanti del sistema sono di tipo differente, aumentando così le probabilità che essi non falliranno esattamente nello stesso modo. I sistemi software che sono progettati per la fidatezza possono includere componenti ridondanti che forniscono le stesse funzionalità di altri componenti; essi vengono attivati nel sistema se quelli principali falliscono. Un'altra forma di ridondanza è l'aggiunta di codice di controllo che può rilevare alcuni tipo di problemi.

La diversità e la ridondanza possono essere usate anche nella progettazione. In un processo di sviluppo software, le attività quali convalida del software possono affidarsi a più strumenti o tecniche. Questo aumenta la fidatezza del software perché riduce le probabilità di guasti nei processi, dove l'errore umano porta a errori del software. Per esempio, le attività di convalida possono includere i test dei programmi, le ispezioni manuali e l'analisi statica, come le tecniche di ricerca dei guasti. Una qualsiasi di queste tecniche potrebbe individuare un guasto che sfugge alle altre tecniche.

Tuttavia, l'uso stesso della ridondanza e della diversità può introdurre bug nel software in quanto rendono i sistemi più complessi.

Processi software fidati

I processi software fidati sono processi che sono pensati per produrre un software fidato. Segue una lista degli attributi generali dei processi software fidati:

- **Verificabile**: il processo dovrebbe essere comprensibile a persone diverse da quelle che prendono parte al processo, che possono verificare che gli standard di progettazione sono applicati e che possono dare suggerimenti per il suo miglioramento.
- **Differenti**: il processo dovrebbe includere attività di verifica e convalida ridondanti diverse.
- **Documentabile**: il processo dovrebbe avere un modello di processo definito che stabilisca le attività del processo e la documentazione da produrre durante queste attività.
- **Robusto**: il processo dovrebbe essere in grado di correggere gli errori derivanti dalle singole attività del processo.
- **Standardizzato**: dovrebbe essere disponibile una serie completa di standard di sviluppo che definisce come il software debba essere prodotto e documentato.

Questo significa che un processo fidato deve essere esplicitamente **definito** e **ripetibile**:

Definito: un processo esplicitamente definito è un processo che ha un modello definito; utilizzato come guida nella produzione del software. Devono essere raccolti dei dati durante il processo per dimostrare che il team di sviluppo ha seguito il processo secondo le direttive del modello.

Ripetibile: un processo ripetibile è un processo che non si basa su singole interpretazioni e giudizi, ma può essere applicato a vari progetti con team di sviluppo differenti, indipendentemente dalle persone che si occupano dello sviluppo. Questo è particolarmente importante per i sistemi critici, che spesso hanno un lungo ciclo di sviluppo, durante il quale si verificano cambiamenti significativi nel team.

Come detto precedentemente, i processi fidati adottano la ridondanza e la diversità per ottenere un elevato livello di fidatezza. Spesso includono attività differenti che svolgono lo stesso compito. Per esempio, le ispezioni e i test di un programma hanno lo scopo di identificare bug. Queste due tecniche possono essere utilizzate insieme.

Il seguente elenco riporta alcuni esempi di attività che potrebbero essere incluse in un processo fidato per aumentare il livello di ridondanza e diversità (e quindi la fidatezza):

- Ispezione dei requisiti;
- Gestione dei requisiti;
- Specifica formale;
- Modellazione del sistema;
- Analisi statica;
- Ispezioni dei progetti e dei programmi;
- Pianificazione e gestione dei test.

Oltre alle attività dei processi che si focalizzano sullo sviluppo e i test dei sistemi, occorrono anche i processi di gestione della qualità e delle modifiche dei requisiti. Mentre le prime variano da una società all'altra; l'esigenza di una gestione efficiente della qualità e delle modifiche è universale.

I processi di gestione della qualità stabiliscono una serie di standard di processo e di prodotto.

La gestione delle modifiche, si occupa delle modifiche da apportare a un sistema, garantendo che le modifiche accettate vengano effettivamente implementate e che siano incluse nelle release pianificate del software.

A causa del crescente utilizzo dei metodi agili, sempre più società che realizzano sistemi software fidati e che si affidano a processi plan-driven sono riluttanti ad apportare modifiche alle loro procedure. Esse riconoscono comunque il valore degli approcci agili e stanno studiando il modo in cui rendere più agili i processi di sviluppo software fidato.

Metodi formali

I **metodi formali** nello sviluppo del software sono approcci matematici in cui viene definito un modello formale del software. Poi è possibile analizzare questo modello per cercare errori e incongruenze, dimostrare che un programma è coerente con il modello, o applicare una serie di trasformazioni del modello per generare un programma. La prova dei programmi adesso è supportata da software speciali che eseguono dimostrazioni di teoremi su larga scala, che possono essere applicati a sistemi più grandi. Tuttavia, sviluppare le prove per questo tipo di software è un compito difficile e specialistico; per questo la verifica formale non si è molto diffusa.

Un approccio alternativo è lo sviluppo basato sull'affinamento della specifica. Una **specifiche formale** di un sistema viene affinata attraverso una serie di trasformazioni per generare il software. Poiché le trasformazioni sono affidabili in quanto preservano la correttezza del codice, si può avere la certezza che il programma generato sia coerente con la sua specifica formale. I modelli formali basati sulla verifica dei modelli si basano sulla costruzione o generazione di un modello formale degli stati di un sistema e sull'utilizzo di un verificatore di modelli che controlla che le proprietà del modello siano sempre assicurate. I metodi formali per l'ingegneria del software sono efficaci per scoprire e analizzare errori e omissioni, incongruenze tra specifica e programma.

11. Ingegneria dell'affidabilità

Il modello **difetto-errore-fallimento** si basa sul concetto che gli errori umani generano difetti nel software, i difetti causano errori, e gli errori producono fallimenti.

- **Errore umano**: comportamento umano che si traduce nell'introduzione di difetti nel sistema;
- **Difetto del sistema**: caratteristica di un sistema software che può portare a un errore del sistema.
- **Errore del sistema**: comportamento del sistema non atteso dagli utenti.
- **Fallimento del sistema**: evento che si verifica nel momento in cui il sistema non fornisce il servizio che gli utenti si aspettano.

I difetti di un sistema non necessariamente si traducono in errori, e gli errori non necessariamente si traducono in fallimenti del sistema. La distinzione tra difetti, errori e fallimenti porta a distinguere tre approcci complementari che vengono utilizzati per migliorare l'affidabilità di un sistema:

- **Evitare i difetti (fault-avoidance)**: il processo di progettazione e implementazione dovrebbe utilizzare approcci allo sviluppo del software che aiutano ad evitare errori di progettazione e programmazione così da minimizzare la possibilità di introdurre difetti nel sistema.
- **Identificazione e correzione dei difetti (fault detection and correction)**: i processi di verifica e convalida servono a scoprire e rimuovere i difetti in un programma, prima che questo sia consegnato agli utenti. I sistemi critici richiedono verifiche e convalide esterne per scoprire il maggior numero possibile di difetti prima della consegna del software, e per convincere gli stakeholder e i responsabili della regolamentazione che il sistema è affidabile.
- **Tolleranza dei difetti (fault tolerance)**: il sistema è progettato in modo che i difetti o i comportamenti imprevisti siano rilevati durante l'esecuzione del software e siano gestiti in modo tale che non si verifichi un fallimento del sistema.

L'applicazione delle tecniche di fault-avoidance, fault-detection e fault-tolerance purtroppo non è sempre economicamente vantaggiosa. I costi per trovare e rimuovere i difetti residui in un sistema software salgono esponenzialmente man mano che i difetti vengono identificati e rimossi.

Disponibilità e affidabilità

Se pensiamo ai sistemi come strumenti che forniscono qualche tipo di servizio, allora per **disponibilità** di un servizio si intende che il servizio deve essere attivo e in esecuzione, e per **affidabilità** si intende che il servizio fornisce risultati corretti. L'affidabilità e disponibilità possono essere definite più precisamente in questo modo:

- **Affidabilità**: la probabilità che ha un'operazione di essere svolta senza che il sistema fallisca in un determinato periodo di tempo, in un dato ambiente, per uno scopo specifico.
- **Disponibilità**: la probabilità che un sistema, in un certo momento, sarà operativo e in grado di fornire i servizi richiesti.

L'affidabilità di un sistema non è un valore assoluto, dipende dall'ambiente e dal modo in cui il sistema viene utilizzato. La definizione data di affidabilità di un sistema si basa sul concetto di funzionamento senza fallimenti. Un **fallimento** è un comportamento che non è conforme alla specifica del sistema. Tuttavia:

- le specifiche del software spesso sono incomplete e poco chiare.
- nessuno, tranne gli sviluppatori del sistema, legge i documenti della specifica del software.

Il fallimento dunque non è qualcosa di definibile con estrema obiettività, ma bensì un giudizio espresso dagli utenti di un sistema.

L'affidabilità e la disponibilità sono strettamente correlate, ma a volte una è più importante dell'altra.

Come si misura la **DEPENDABILITY**?
Esistono requisiti funzionali e non funzionali.

La reliability va considerata su **3 componenti diverse** (cause di malfunzionamento):

- **HW**: la probabilita' che un componente hw fallisca
- **SW**: la probabilita' che un componente sw fornisca un output **incorrecto**.
- **OPERATORE**: la probabilita' che un operatore umano faccia un errore.

Il **livello richiesto di reliability va espresso numericamente**, essendo la reliability un attributo **DINAMICO** (legato all'esecuzione).

Per lo studio della reliability si utilizzano delle metriche che mirano a misurare il numero di failure o il tempo tra essi.

Essendo **procedure STATISTICHE**, si ripetono le misurazioni piu' volte e si mediano i risultati.

Tecniche: POFOD, AVAIL, MTTF, ROCOF.

Le **failure** possono avere effetti diversi, perciò vengono classificate in diverse classi con metriche diverse:

- **TRANSIENT**: Accadono solo con certi input.
- **PERMANENT**: Accadono con tutti gli input.
- **RECOVERABLE**: Il sistema puo' ripristinarsi senza l'intervento dell'operatore.
- **UNRECOVERABLE**: E' richiesto l'intervento dell'operatore.
- **NON CORRUPTING**: Le failure non corrompono lo stato del sistema e i dati.
- **CORRUPTING**: Le failure corrompono lo stato del sistema e i dati.

Anche l'**entita del danno** puo essere classificata in diversi livelli:

- **MINOR**
- **MARGINAL**
- **CRITICAL**
- **CATASTROPHIC**

Il **testing del sw** puo' essere ripetuto finche non si raggiunge un determinato livello di reliability.

Esistono:

- **DEFECT TESTING**: L'obiettivo e' individuare le failure per poi rimuovere i bug con il debugging.
- **STATISTICAL TESTING**: Misurare l'affidabilita'. Si fornisce una stima misurando i fallimenti accaduti in un intervallo di tempo.

Misurando il numero di failure/errori mi permette di misurare e **PREDIRRE** la reliability futura del mio sw.

Per testare il sw devo stabilire un **PROFILO OPERATIVO**, ovvero un **set di dati di test che darò in input al sw** (possono essere generati da vecchi dati di altri sistemi).

I profili operativi dovrebbero essere generati automaticamente ogni volta possibile

Requisiti

L'affidabilità di un sistema non dipende semplicemente dalla buona ingegneria del software. Occorre prestare attenzione anche ai dettagli quando vengono definiti i requisiti che determinano la fidatezza del sistema. Questi requisiti di fidatezza sono di due tipi:

- I **requisiti funzionali** che definiscono le **funzioni di controllo e ripristino che devono essere incluse nel sistema e le funzioni che forniscono la protezione contro i fallimenti del sistema e gli attacchi esterni**.
- I **requisiti non funzionali** che definiscono l'**affidabilità e la disponibilità del sistema**.

Come detto precedentemente, l'**affidabilità complessiva di un sistema dipende anche dall'affidabilità dell'hardware**. Oltre a includere i requisiti che compensano i fallimenti del software, ci devono essere anche i requisiti di affidabilità che aiutano a identificare e a risolvere un guasto dell'hardware o un errore dell'operatore.

Metriche di affidabilità e disponibilità

Una **metrica** del software è una **caratteristica del software che può essere effettivamente misurata**.

Quattro metriche principali possono essere utilizzate per specificare l'affidabilità e la disponibilità:

1. **Probabilità di fallimento su richiesta (POFOD: Probability Of Failure On Demand)**: definisce la probabilità che la richiesta di un servizio provochi il fallimento di un sistema. Quando le richieste sono poco frequenti
2. **Tasso di occorrenza dei fallimenti (ROCOF: Rate Of Occurrences Of Failures)**: definisce il numero di fallimenti del sistema che probabilmente si verificheranno in un certo periodo di tempo o in un certo numero di esecuzioni del sistema. Quando le richieste sono frequenti
3. **Tempo medio al fallimento (MTTF: Mean Time To Failure)**: numero medio di unità di tempo tra due fallimenti del sistema.
4. **Disponibilità (AVAIL: Availability)**: indica la probabilità che un sistema potrà essere operativo quando viene effettuata la **richiesta** di un servizio del sistema.

Continua il discorso sui requisiti di affidabilità:

I requisiti di affidabilità non funzionali sono specifiche della disponibilità e dell'affidabilità di un sistema espresse mediante una delle metriche di affidabilità appena descritte. Le specifiche quantitative della disponibilità e dell'affidabilità sono utili per vari motivi:

- Il processo per decidere il livello di affidabilità richiesto aiuta gli stakeholder a capire meglio di cosa hanno effettivamente bisogno. Gli stakeholder capiscono che ci sono vari tipi di fallimenti del sistema e che è costoso raggiungere alti livelli di affidabilità.
- Consente di stabilire quando interrompere i test di un sistema. L'interruzione può avvenire quando il sistema ha raggiunto il livello di affidabilità richiesto.
- Permettono di definire le strategie di progettazione per migliorare l'affidabilità di un sistema.
- Se un ente di regolamentazione deve approvare un sistema prima che venga messo in servizio, è importante fornire la prova obiettiva che è stato raggiunto il livello di affidabilità richiesta per ottenere la certificazione.

Per raggiungere un alto livello di affidabilità e disponibilità in un sistema che fa uso intensivo del software, si usa una combinazione di tecniche fault-avoidance, fault-detection e fault-tolerance. Questo significa che bisogna definire i requisiti di affidabilità funzionali per specificare come il sistema dovrà essere in grado di evitare, identificare e tollerare i difetti del software. Questi requisiti di affidabilità funzionali dovrebbero specificare i difetti da identificare e le azioni da svolgere per garantire che tali difetti non provochino fallimenti del sistema. La specifica dell'affidabilità richiede, quindi, l'analisi dei requisiti non funzionali, la valutazione dei rischi per l'affidabilità e la specifica delle funzioni che evitano questi rischi:

Ci sono quattro tipi di **requisiti di affidabilità funzionali**:

1. **Requisiti di controllo**: specificano i controlli sugli input del sistema per garantire che gli input errati siano identificati prima che siano elaborati dal sistema.
2. **Requisiti di ripristino**: definiscono le azioni che dovrà svolgere il sistema dopo che si è verificato un guasto.
3. **Requisiti di ridondanza**: specificano le funzioni ridondanti del sistema che garantiscono che il fallimento di un singolo componente non provochi la perdita completa dei servizi.
4. **Requisiti del processo**: sono i requisiti di fault-avoidance; assicurano che siano sempre adottate delle buone tecniche nel processo di sviluppo del software.

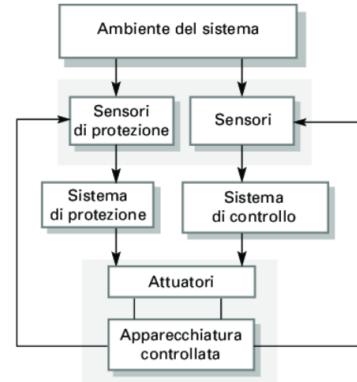
Architetture fidate

Affinché un sistema sia **fault-tolerant**, la sua architettura deve essere progettata per includere hardware e software ridondanti e diversi. La più semplice realizzazione di un'architettura fidata si trova nei server replicati, dove due o più server possono svolgere lo stesso compito.

Esempi di architetture fault-tolerant:

Sistemi di protezione

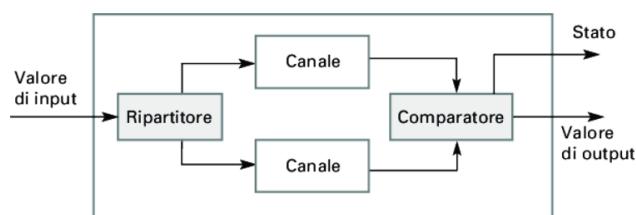
Un sistema di protezione è un sistema specializzato che è associato a qualche altro sistema che, di solito è il sistema di controllo di qualche processo, come ad esempio un processo chimico. I sistemi di protezione monitorano il loro ambiente in maniera autonoma. Se i sensori segnalano un problema che il sistema di controllo non sta gestendo, allora il sistema di protezione viene attivato per arrestare il processo o attuare le procedure predefinite. Un sistema di protezione include soltanto le funzionalità critiche che sono richieste per commutare il sistema controllato da uno stato di errore a uno stato sicuro. Il vantaggio di questo stile architettonico è che il software del sistema di protezione può essere molto più semplice del software che controlla il processo protetto.



Architetture auto-monitorate

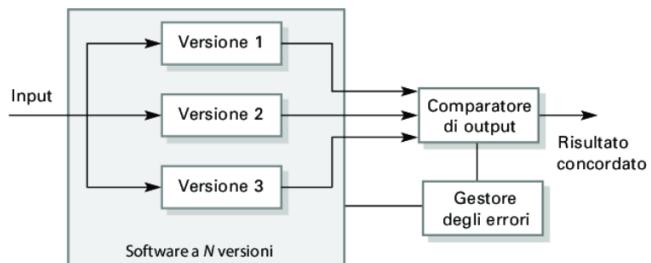
Un'architettura auto-monitorata è un'architettura nella quale il sistema è progettato per monitorare le sue operazioni e svolgere qualche azione se rileva qualche problema. I calcoli sono svolti su canali separati e i risultati confrontati. Se i risultati combaciano allora si ritiene che il sistema stia operando correttamente. Per essere efficienti nel rilevare gli errori del software e i guasti dell'hardware, i sistemi auto-monitorati devono essere progettati seguendo queste linee guida:

- L'hardware utilizzato deve essere diverso in ciascun canale, altrimenti gli stessi errori hardware possono verificarsi in entrambi i canali.
- Il software utilizzato in ciascun canale deve essere diverso, altrimenti lo stesso errore del software può verificarsi in entrambi i canali.



Programmazione a N versioni

Un approccio software fault-tolerant è quello di eseguire N diverse versioni di un sistema in parallelo. Questo approccio alla tolleranza agli errori consiste nel fatto che esse abbiano le specifiche comuni ma che vengano implementate da un certo numero di team. Queste versioni vengono eseguite su hardware diverso. Viene considerato corretto l'output più prodotto dalle N versioni, gli altri vengono considerati errati.



Diversità

Tutte le precedenti architetture fault-tolerant si basano sulla diversità del software per realizzare la tolleranza agli errori. Ciò si basa sull'ipotesi che implementazioni differenti della stessa specifica siano indipendenti. Queste implementazioni non dovrebbero includere errori comuni e non dovrebbero fallire nello stesso modo, contemporaneamente. Tuttavia, la diversità del software è difficile da realizzare perché è praticamente impossibile garantire che ciascuna delle N versioni sia veramente indipendente dalle altre; potrebbero avere gli stessi errori.

Misura dell'affidabilità

Per valutare l'affidabilità di un sistema, occorre raccogliere dati sul suo funzionamento. I dati richiesti possono includere:

- Numero di fallimenti del sistema che si verificano per un determinato numero di richieste del sistema.
- Tempo o numero di transazioni tra i fallimenti del sistema più il tempo trascorso o il numero totale di transazioni.
- Il tempo di recovery (riparazione e riavvio) dopo un fallimento del sistema che ha portato la perdita di un servizio.

Il test dell'affidabilità è un processo statistico che ha lo scopo di misurare l'affidabilità di un sistema. Questo processo ha il compito di misurare l'affidabilità, non quello di trovare gli errori. Il test statistico dell'affidabilità è composto da quattro fasi:

1. Iniziare studiando i sistemi dello stesso tipo per capire come vengono utilizzati in pratica. Lo scopo è definire un profilo operativo.
2. Definire una serie di dati di test che rispecchiano il profilo operativo.
3. Provare il sistema con questi dati e monitorare i risultati ottenuti.
4. Si calcolano le metriche e l'affidabilità con i risultati ottenuti.

12. Ingegneria della sicurezza

La sicurezza è una delle principali proprietà della fidatezza. Un sistema può essere considerato sicuro se opera senza fallimenti catastrofici, ovvero senza provocare morti o feriti tra le persone, danni economici o ambientali. La sicurezza nei sistemi software si raggiunge cercando di capire le situazioni che potrebbero portare a fallimenti correlati alla sicurezza. Il software viene progettato in modo che tali fallimenti non possano verificarsi.

Lo sviluppo dei sistemi a sicurezza critica usa i tre approcci utilizzati per evitare i difetti in ambito di disponibilità e affidabilità del software (fault-avoidance, detection, correction, tollerance) potenziati mediante tecniche guidate dai rischi, che considerano i potenziali incidenti che possono capitare a un sistema software:

- **Evitare i rischi (hazard-avoidance)**: il sistema è progettato in modo da evitare i rischi.
- **Identificare ed eliminare i rischi (hazard detection and removal)**: il sistema è progettato in modo che i rischi siano individuati e rimossi prima che possano provocare incidenti.
- **Limitazione dei danni**: il sistema potrebbe includere delle funzioni di protezione che minimizzano i danni che un incidente potrebbe causare.

Un **rischio** è uno stato del sistema che può causare un incidente. Più nello specifico è una misura della probabilità che il sistema provochi un incidente.

Requisiti

La sicurezza di un sistema non dipende soltanto dalle buone tecniche di ingegneria, ma occorre prestare attenzione ai dettagli quando vengono definiti i requisiti del sistema e vengono inseriti speciali requisiti software che sono orientati a garantire la sicurezza di un sistema. I requisiti di sicurezza sono requisiti funzionali, che definiscono le funzionalità di controllo e recovery che dovrebbero essere incluse nel sistema e le caratteristiche che forniscono la protezione contro i fallimenti del sistema e gli attacchi esterni. Per generare requisiti di sicurezza funzionali, di solito si inizia studiando il dominio applicativo, i regolamenti e gli standard di sicurezza.

Diversamente dai normali requisiti funzionali che definiscono ciò che il sistema deve fare, i requisiti "shall-not" (non deve) definiscono il comportamento del sistema che non è accettabile.

La **specificazione dei requisiti basata sui rischi** è un approccio generale che è utilizzato nell'ingegneria di sistemi critici, dove vengono identificati non solo i rischi cui sono sottoposti i sistemi, ma anche i requisiti per evitare o ridurre tali rischi. La specifica può essere utilizzata per tutti i tipi di requisiti di fidatezza. Per i sistemi a sicurezza critica, essa si traduce in un processo guidato dai rischi che sono stati identificati. Il processo di specifica della sicurezza guidata dai rischi è composto da quattro attività:

1. **Identificazione dei rischi**: questa attività identifica i rischi potenziali di un sistema.
2. **Valutazione dei rischi**: questa attività decide quali rischi sono più o meno pericolosi e/o frequenti.
3. **Analisi dei rischi**: è il processo che analizza le cause principali dei rischi e gli eventi che possono concretizzarli.
4. **Riduzione dei rischi**: questo processo si basa sui risultati dell'analisi e permette di stabilire i requisiti di sicurezza, ovvero i requisiti che definiscono le tecniche per evitare i rischi o per impedire che un rischio degeneri in un incidente, e in caso di incidente, per ridurre al minimo i danni risultanti.

13. Ingegneria della protezione

Oggi è essenziale progettare sistemi in grado di resistere agli attacchi esterni; senza difese appropriate, è inevitabile che il funzionamento del sistema collegato alla rete risulterà compromesso da tali attacchi. Gli hacker potranno utilizzare il sistema per scopi illeciti, accedere a informazioni confidenziali o interrompere i servizi offerti dal sistema.

Per progettare sistemi protetti occorre tenere in considerazione tre fattori essenziali:

1. **Riservatezza:** le informazioni di un sistema potrebbero diventare accessibili a persone o a programmi che non sono autorizzati ad accedere a tali informazioni.
2. **Integrità:** le informazioni di un sistema possono danneggiarsi, diventando inutilizzabili o inaffidabili.
3. **Disponibilità:** l'accesso a un sistema o ai suoi dati potrebbe diventare impossibile.

La **protezione** è un attributo del sistema che rispecchia la capacità di autoprotezione del sistema dagli attacchi esterni o interni. Per molti sistemi la protezione è l'attributo più importante della fidatezza. I controlli che potremmo mettere in atto per migliorare la protezione si basano sui soliti tre concetti fondamentali: evitare i rischi (avoidance); identificare i rischi (detection) e ripristinare le normali funzionalità (recovery).

- Evitare le vulnerabilità (avoidance);
- Identificazione e neutralizzazione degli attacchi (detection);
- Limitare l'esposizione e ripristinare il sistema (recovery).

La protezione è strettamente correlata con gli altri attributi della fidatezza: affidabilità, disponibilità, sicurezza e resilienza. La protezione deve essere garantita se vogliamo creare sistemi affidabili, disponibili e sicuri. Non è una caratteristica facoltativa, che può essere aggiunta in un secondo momento, ma deve essere tenuta in considerazione in tutte le fasi del ciclo di sviluppo, dalla definizione dei requisiti fino alla messa in funzione del sistema.

Esempi di correlazione tra protezione e altre caratteristiche della fidatezza:

- **Protezione e affidabilità:** se un sistema subisce un attacco e il sistema o i suoi dati vengono danneggiati, questo compromette l'affidabilità del sistema.
- **Protezione e disponibilità:** un tipico attacco che viene portato a compimento è l'attacco denial-of-service. Questo compromette la disponibilità del sistema.

16. Ingegneria del software basato sui componenti

I **componenti** sono astrazioni di livello più elevato degli oggetti e sono definiti dalle loro interfacce. Di solito sono più grandi dei singoli oggetti, e tutti i dettagli di implementazione sono nascosti agli altri componenti. L'ingegneria del software basato sui componenti è il processo che definisce, implementa e integra questi componenti indipendenti.

CBSE (Component-based Software Engineering) è diventato un importante approccio allo sviluppo del software per sistemi aziendali su larga scala. Gli elementi fondamentali dell'ingegneria del software basato sui componenti sono diversi:

- **Componenti indipendenti**: sono specificati completamente dalle loro interfacce. Dovrebbe esserci una netta separazione tra l'interfaccia e la sua implementazione, in modo che l'implementazione possa esser sostituita senza modificare altre parti del sistema.
- **Standard dei componenti**: definiscono le interfacce e facilitano l'integrazione dei componenti. Questi standard sono incorporati in un modello di componenti; definiscono come dovrebbero essere specificate le interfacce dei componenti e come i componenti comunicano tra loro.
- **Middleware**: fornisce un supporto software all'integrazione dei componenti. Per far sì che componenti indipendenti e distribuiti lavorino assieme, occorre un supporto middleware che gestisca la comunicazione tra i componenti. Il middleware per il supporto dei componenti gestisce efficacemente i problemi di basso livello e consente agli sviluppatori di concentrarsi sui problemi relativi all'applicazione; questo middleware, inoltre, può fornire un supporto all'allocazione delle risorse, alla gestione delle transazioni, alla protezione e alla simultaneità.
- **Un processo di sviluppo**: che sia adatto all'ingegneria del software basato sui componenti

L'ingegneria del software basato sui componenti include i sani **principi di progettazione** che supportano la realizzazione di software comprensibile e mantenibile.

- Ogni componente è indipendente, quindi non interferisce con le operazioni di un altro componente. I dettagli dell'implementazione sono nascosti. L'implementazione di un componente può essere modificata senza influire sulle parti restanti del sistema.
- I componenti comunicano attraverso interfacce ben definite. Se queste interfacce sono mantenute, un componente può essere sostituito da un altro che fornisce nuove o migliori funzionalità.
- Le infrastrutture dei componenti forniscono una vasta gamma di servizi standard che possono essere utilizzati nei sistemi applicativi; questo riduce la quantità di nuovo codice da sviluppare.

L'ingegneria del software basato sui componenti fornisce supporto sia al riutilizzo del software che all'ingegneria del software distribuito.

Componenti

Un componente è un'entità software indipendente che può essere composta con altri componenti per creare un sistema software. Un componente software è conforme a un modello di componenti standard, può essere consegnato singolarmente e composto senza modifiche secondo uno standard di composizione (o almeno questo in linea teorica).

Un componente può essere immaginato come un fornitore di uno o più servizi, anche se il componente è incorporato nel sistema, anziché essere implementato come servizio.

Immaginare un componente come fornitore di servizi permette di enfatizzare due caratteristiche critiche dei componenti riutilizzabili:

- Il componente è un'entità eseguibile indipendente, che è definita dalle sue interfacce. Non occorre conoscere il suo codice sorgente per utilizzarlo.
- I servizi offerti da un componente sono resi disponibili tramite un'interfaccia, attraverso la quale passano tutte le interazioni.

17. Ingegneria del software distribuito

Tutti i grandi sistemi informatici sono ora sistemi distribuiti. Un **sistema distribuito** è quello in cui **un'applicazione viene eseguita su più computer, anziché su una singola macchina**. Anche quelle applicazioni indipendenti, che vengono eseguite su un PC sono sistemi distribuiti. Esse vengono eseguite su un singolo computer, ma spesso utilizzano sistemi cloud remoti per l'aggiornamento, memorizzazione di dati e altri servizi. **Un sistema distribuito può essere definito come "un gruppo di computer indipendenti che appaiono all'utente come un unico sistema coerente"**.

Quando si progetta un sistema distribuito, ci sono problemi specifici da affrontare, semplicemente perché il sistema è distribuito. Questi problemi derivano dal fatto che varie parti del sistema vengono eseguite su computer gestiti in modo indipendente; inoltre, durante la progettazione bisogna tenere in considerazione le caratteristiche della rete, quali affidabilità e latenza.

I cinque principali **vantaggi** offerti dai sistemi distribuiti sono:

- **Condivisione delle risorse**: un sistema distribuito consente di **condividere le risorse hardware e software associate ai computer di una rete**.
- **Apertura**: i sistemi distribuiti sono sistemi aperti, ovvero progettati su protocolli Internet standard che permettono di **combinare più unità hardware e software di diverse tipologie**.
- **Simultaneità**: in un sistema distribuito **possono operare contemporaneamente più processi** su vari computer della rete.
- **Scalabilità**: teoricamente, i sistemi distribuiti sono scalabili; in pratica, la rete può limitare questa caratteristica.
- **Tolleranza ai guasti**: la disponibilità di diversi computer e la possibilità di **replicare le informazioni permettono ai sistemi distribuiti di tollerare alcuni guasti** hardware e software.

I sistemi distribuiti **sono più complessi di quelli centralizzati**; questo complica la fase di progettazione e di testing. I **problemi** da considerare quando si progetta un sistema distribuito includono la trasparenza, l'apertura, la scalabilità, la protezione, la qualità dei servizi e la gestione dei guasti.

PATTERN

Gli schemi architetturali principali per i sistemi distribuiti sono:

- **Master-slave**;
- **Client-server a due o più livelli**;
- **Componenti distribuiti**: richiedono il middleware per gestire le comunicazioni dei componenti e per consentire agli oggetti di essere aggiunti e rimossi dal sistema.
- **Peer-to-peer**: architetture decentralizzate nelle quali client e server non sono distinti. I calcoli possono essere distribuiti su più sistemi di varie società.

I principali **SVANTAGGI** dei **SISTEMI DISTRIBUITI** sono:

- **COMPLESSITA'**
- **SICUREZZA**: sono piu' suscettibili ad attacchi esterni
- **GESTIBILITA'**

I componenti in un sistema distribuito sono spesso eterogenei tra loro (magari anche implementati con linguaggi diversi) —> ci viene in aiuto il **MIDDLEWARE**.

Il MIDDLEWARE e' un sw che si occupa di **coordinare le interazioni** tra le varie componenti. Spesso e' un sw **OFF THE SHELF** (piuttosto che specifico)

Il MIDDLEWARE **rende astratto** a chi sviluppa sw applicativo di cio' che c'e' sotto, il programmatore non si deve preoccupare

TIPI di SISTEMI DISTRIBUITI:

- sistemi **MULTIPROCESSORE**: composti da piu processori. Un esempio e' **MASTER-SLAVE PATTERN** dove c'e' un processo **MASTER** che coordina e gestisce i processi slave. I processi **SLAVE** si occupano di azioni specifiche (es. l'acquisizione dei dati da un array di un sensore)
- sistemi **CLIENT/SERVER**: i **SERVER** offrono servizi ai **CLIENT** che li richiedono. I client e i server sono processi logici, quindi non c'e' corrispondenza (non per forza) 1:1 con i processori HW (ovvero non e' detto che su un processore giri ad es. solo un processo client o server, ce ne possono essere di piu'). Un esempio sono i **SISTEMI CLIENT-SERVER A LIVELLI** (livelli: presentazione, data handling, application processing layer, database)

Dei sistemi **client-server a livelli** ci sono:

- **DUE LIVELLI**: in cui troviamo i
 - **THIN CLIENT** dove nel client e' implementato solo il livello **presentazione** che si occupa di visualizzare i dati in formato corretto e il resto e' nel server usato quando passiamo da sistema legacy a client server —> alto carico per il server e
 - **THICK CLIENT** dove nel client sono implementati tutti i livelli escluso quello **database** che e' implementato nel server —> client piu' potente
- **MULTILIVELLO**: i diversi livelli sono processi separati che possono essere eseguiti su processori diversi —> rimuove i problemi di scalabilita'.

18. Ingegneria del software orientato ai servizi

I sistemi orientati ai servizi, sono un modo di sviluppare sistemi distribuiti, dove i componenti del sistema sono servizi indipendenti, che vengono eseguiti su computer distribuiti in tutto il mondo. I servizi sono indipendenti dalle piattaforme e dai linguaggi di implementazione. I sistemi software possono essere realizzati componendo servizi locali e servizi esterni offerti da fornitori differenti, con un'interazione continua tra i servizi del sistema.

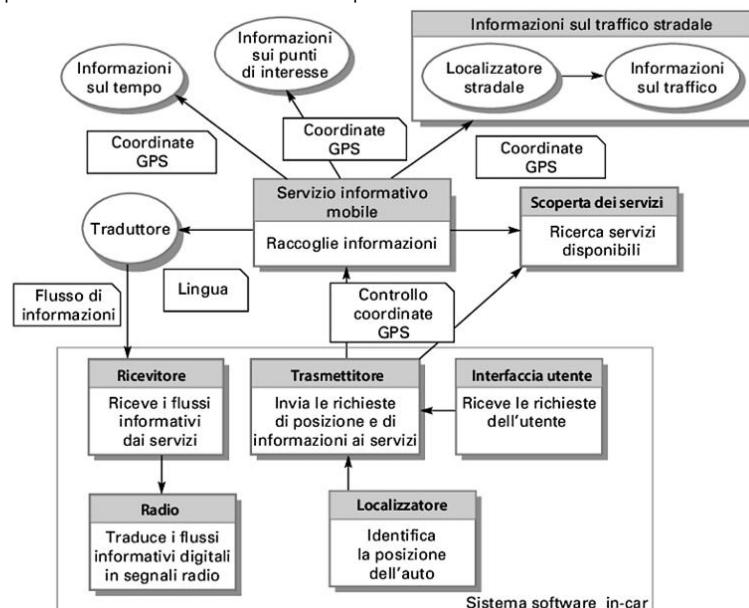
Distinzione tra i concetti di "software come servizio" e "sistemi orientati ai servizi":

- **Software come servizio:** offrire agli utenti funzionalità software in modo remoto sul web, anziché tramite applicazioni installate su dispositivi dell'utente.
- **Sistemi orientati ai servizi:** sistemi implementati utilizzando componenti di servizi riutilizzabili e accessibili da altri programmi, anziché direttamente dagli utenti.

Adottando un approccio orientato ai servizi nell'ingegneria del software, si ottengono alcuni importanti benefici:

- I servizi possono essere offerti da qualsiasi fornitore di servizi dentro o fuori un'organizzazione.
- I fornitori di servizi rendono pubbliche le informazioni sul servizio in modo che qualsiasi utente autorizzato possa utilizzarlo.
- Le applicazioni possono ritardare il collegamento ai servizi finché questi non saranno consegnati o eseguiti.
- È possibile costruire opportunisticamente nuovi servizi. Un fornitore di servizi può riconoscere nuovi servizi che possono essere creati collegando i servizi esistenti in modo innovativo.
- Gli utenti possono pagare i servizi in base all'utilizzo effettivo, anziché alla loro fornitura.
- Le applicazioni possono essere più piccole.

Esempio di un sistema informativo per automobili basato sui servizi:



19. Gestione della progettazione

La gestione dei progetti (project management) è una parte essenziale dell'ingegneria del software. I progetti devono essere gestiti perché l'ingegneria del software professionale è sempre soggetta a vincoli aziendali di budget e di tempi. Il **compito di un project manager** è garantire che un progetto **software soddisfi questi vincoli e abbia un elevato livello di qualità**. La buona gestione non può garantire il successo di un progetto, ma la cattiva gestione di solito ne determina il fallimento.

I criteri per il successo della gestione dei progetti ovviamente variano da progetto a progetto, ma per la maggior parte dei progetti, gli obiettivi più importanti sono:

1. Consegnare il software al cliente entro il termine concordato;
2. Mantenere i costi complessivi entro il budget;
3. Consegnare un software che soddisfa le richieste dell'utente;
4. Mantenere una squadra di sviluppo affiatata e collaudata;

Questi obiettivi sono generici del campo dell'ingegneria. Tuttavia, l'ingegneria del software è diversa dagli altri tipi di ingegneria per varie ragioni che rendono la gestione del software particolarmente impegnativa. Alcune differenze sono elencate qui di sotto:

- Il prodotto è intangibile;
- I grandi progetti sono spesso "unici" (mai realizzati prima);
- I processi software sono variabili e specifici di un'organizzazione.

È difficile fornire una descrizione dei compiti standard di un project manager. I compiti variano notevolmente in funzione dell'azienda e del tipo di software che si sta sviluppando. Alcuni dei più importanti fattori che influiscono sulla gestione dei progetti sono qui elencati:

- Dimensione dell'azienda;
- Clienti del software;
- Dimensione del software;
- Tipologia di software;
- Cultura aziendale;
- Processo di sviluppo del software.

Tuttavia, alcune attività di gestione dei progetti sono comuni a tutte le aziende:

- **Pianificazione dei progetti**: i project manager sono responsabili della pianificazione della stima e della tempistica dello sviluppo dei progetti; inoltre devono assegnare i compiti al personale. Hanno la supervisione del lavoro per garantire che esso sia svolto secondo gli standard, e tengono sotto controllo l'avanzamento del lavoro per verificare che esso venga sviluppato in tempo ed entro il budget previsto. Attività più lunga per un project manager, è un'ATTIVITA' CONTINUA che va continuamente revisionata
- **Gestione dei rischi**: i project manager devono definire i rischi che potrebbero influenzare un progetto, monitorare questi rischi e svolgere le azioni appropriate quando si presenta qualche problema.
- **Gestione del personale**: i project manager sono responsabili della gestione di una squadra di persone. Devono selezionare le persone della loro squadra e definire le modalità operative che possono migliorare l'efficienza del lavoro di gruppo.
- **Reporting**: i project manager hanno il compito di documentare il progresso di un progetto in appositi report da inviare ai clienti e ai manager della società che sviluppa il software.

Segue la descrizione di due compiti fondamentali del project management: gestione dei rischi e pianificazione dei progetti ([capitolo 20](#)).

L'attività di planning è CICLICA, inizia con la definizione di un primo piano e prosegue con una continua attività di monitoraggio al fine di individuare la necessità di eventuali ri-pianificazioni.

Il project Plane contiene l'analisi del rischio e i requisiti SW e HW che sono previsti per lo svolgimento del progetto.

Le MILESTONES sono gli end-point di un processo e sono per noi (azienda). I DELIVERABLES sono dei risultati (di processi intermedi) consegnati all'utente. Spesso milestones e deliverables coincidono.

Scheduling: per farlo si utilizzano dei tool grafici, il manager deve suddividere il progetto in task e stimare tempo e risorse minimizzando le dipendenze dei task per evitare ritardi. Il diagramma delle attività rappresenta le DIPENDENZE (tra attività) e il CRITICAL PATH (attività non posticipabili, pena rallentamento del progetto)

Produttività: non è proporzionale al numero di persone, i.e. se aumenta il personale non è detto che aumenta la produttività perché aumenta le interazioni e perde tempo ad istruire il personale

Gestione dei rischi

Identifico un rischio ed elaboro un piano per minimizzare gli effetti di esso sul progetto

La **gestione dei rischi** è uno dei **compiti più importanti di un project manager**. I rischi possono minacciare un progetto, il software in via di sviluppo o l'organizzazione. La gestione dei rischi consiste nel **prevedere i rischi che potrebbero influire negativamente** sulla tempistica o sulla qualità del software. I **rischi di progetto** possono essere classificati in base alla loro **natura** (tecnica, organizzativa etc.) oppure in base all'oggetto dei loro effetti:

- **Rischi per il progetto**: influiscono sulle risorse e tempistica del progetto.
- **Rischi per il prodotto**: influiscono sulla qualità o sulle performance del software in sviluppo.
- **Rischi per l'azienda**: influiscono sull'azienda/personale.

Esempi di rischi:

Rischio	Influisce su	Descrizione
Turnover del personale	Progetto	Il personale esperto può abbandonare il progetto prima che sia finito.
Cambio della gestione	Progetto	Ci potrebbe essere un cambio nella gestione aziendale con priorità differenti.
Hardware non disponibile	Progetto	L'hardware che è essenziale per il progetto potrebbe non essere consegnato nei tempi previsti.
Modifiche dei requisiti	Progetto e prodotto	Numero di modifiche dei requisiti superiore al previsto.
Ritardi nelle specifiche	Progetto e prodotto	Le specifiche delle interfacce essenziali non sono disponibili nei tempi previsti.
Sottostima della dimensione	Progetto e prodotto	La dimensione del sistema è stata sottostimata.
Prestazioni insoddisfacenti degli strumenti software	Prodotto	Gli strumenti software che supportano il progetto non funzionano come previsto.
Nuove tecnologie	Azienda	La tecnologia su cui si basa il sistema viene sostituita da nuova tecnologia.
Concorrenza sul prodotto	Azienda	Un prodotto concorrente viene immesso sul mercato prima che il sistema sia stato completato.

La gestione dei rischi è importante a causa delle incertezze intrinseche nello sviluppo del software. Queste incertezze scaturiscono dai requisiti definiti in modo approssimativo, dalle modifiche dei requisiti dovute alle mutate esigenze dei clienti, dalle difficoltà nella stima del tempo e delle risorse necessarie allo sviluppo del software, e dalle differenti capacità dei singoli membri della squadra di sviluppo.

Segue uno schema generico della gestione dei rischi:

- **Identificazione dei rischi** può essere effettuata da un singolo (basandosi sull'esperienza) o in gruppo facendo brainstorming
- **Analisi dei rischi** valuto quanto è probabile che succeda e che danni provocherebbe. Classifico i rischi e questa analisi è **PERIODICA**
- **Pianificazione dei rischi** (definizione di piani di prevenzione) 3 strategie:
 - Cercare di evitare
 - Ridurre l'impatto
 - Prepararsi per quando succederà
- **Monitoraggio dei rischi** Controllo se ci sono dei segnali che sta per avvenire il rischio

20. Pianificazione dei progetti

La **pianificazione della progettazione** è uno dei più importanti compiti di un **project manager**. Un manager deve saper **suddividere il lavoro da svolgere in più parti per assegnarle ai membri di una squadra di sviluppo**; deve saper prevedere i problemi che potrebbero nascere e preparare le soluzioni per risolvere tali problemi. Il **piano di un progetto, che viene creato all'inizio della progettazione e aggiornato durante lo sviluppo, è utilizzato per spiegare come il lavoro sarà fatto e per valutare l'avanzamento del progetto**.

La pianificazione della progettazione si svolge in tre fasi nel ciclo di vita di un progetto:

1. **Proposta**: si scrive un contratto di appalto per sviluppare o fornire un sistema software. In questa fase occorre definire un piano per cercare di capire se si hanno le risorse necessarie per completare il lavoro e per valutare il prezzo da offrire al cliente.
2. **Avviamento**: si scelgono le persone che dovranno realizzare il progetto; si definiscono le parti che dovranno essere progressivamente sviluppate per completare il progetto; si stabiliscono le modalità di allocazione delle risorse per sviluppare il progetto e così via.
3. **Revisioni**: **si rivede periodicamente il progetto**, aggiornando il piano di sviluppo in base alle nuove informazioni sul software e sul suo avanzamento.

Concetto di **deliverable** e **milestone**: sono due termini usati nell'ambito della pianificazione della progettazione. Con **deliverable si intende un risultato di progetto concerto**, che si può fornire a un cliente: un report, un prototipo, il software finale etc. Con **milestone** (in italiano pietra miliare) **si indica il punto finale di un'attività di processo (il raggiungimento di un obiettivo prefissato)**.

21. Gestione della qualità

La gestione della qualità del software si occupa di garantire che i sistemi software sviluppati siano conformi agli scopi prestabiliti, ovvero che i sistemi soddisfino le esigenze dei loro utenti, siano eseguiti in modo efficiente e affidabile e siano consegnati in tempo ed entro i limiti dei costi previsti nel budget. L'uso di tecniche di gestione della qualità insieme a nuove tecnologie software e metodi di testing ha prodotto significativi miglioramenti nel livello di qualità del software.

La **gestione della qualità (QM - Quality Management)** formalizzata è particolarmente importante nei team che sviluppano grandi sistemi software di lunga durata. Per questi sistemi, la gestione della qualità è un problema di natura sia organizzativa sia di natura progettuale.

- A **livello organizzativo**, la gestione della qualità si occupa di definire processi e standard che possano portare allo sviluppo di un software di alta qualità.
- A **livello progettuale**, la gestione della qualità richiede l'applicazione di specifici processi di qualità, verificando che questi processi pianificati siano seguiti e garantendo che gli output del progetto soddisfino gli standard definiti per il progetto.

Terminologie:

- **Garanzia della qualità (Quality Assurance)**: processo generale relativo a **come ci si organizza per raggiungere i requisiti di qualità desiderati**, ossia implica la definizione di processi e standard che dovrebbero aiutare a produrre prodotti di alta qualità. Include quattro aspetti da valutare:
 - **scopo**: cosa controllare?
 - **pianificazione**: quando controllare?
 - **esecuzione**: come controllare?
 - **registrazione**: come strutturare e gestire la registrazione della qualità?
- **Controllo qualità (Quality Control)**: processo specifico relativo a **come si procede nelle singole valutazioni necessarie a misurare la qualità**; inoltre è l'applicazione di questi processi e standard di qualità per eliminare quei prodotti che non hanno raggiunto il livello qualitativo richiesto.

Entrambi questi processi fanno parte della QM.

La gestione della qualità fornisce un controllo indipendente sui processi di sviluppo del software. Il **team QM controlla le consegne del progetto per garantire che esse siano coerenti con gli standard e gli obiettivi aziendali**. Il team QM controlla anche la documentazione dei processi, dove sono registrati i compiti che sono stati completati dal team di sviluppo. Il team QM usa questa documentazione per verificare che non siano stati dimenticati importanti compiti o che un gruppo non abbia fatto ipotesi errate sul lavoro svolto da altri gruppi.

Il team QM è di solito responsabile dei processi di testing della release (test del software prima che venga consegnato ai clienti). Il team QM dovrebbe essere indipendente e non far parte del team di sviluppo così da avere una visuale oggettiva.

La **pianificazione formale della qualità** è parte integrante dei processi di sviluppo plan-driven. È il processo che sviluppa un piano di qualità per un progetto. Un piano di qualità definisce le qualità per il software voluto e come queste debbano essere valutate. Un esempio di struttura guida è:

1. **Informazioni sul prodotto**: descrizione del prodotto, del mercato e delle aspettative di qualità.
2. **Piani del prodotto**: date critiche di rilascio e responsabilità per il prodotto.
3. **Descrizioni del processo**: processi di sviluppo e di assistenza che dovrebbero essere utilizzati.
4. **Obiettivi di qualità**: obiettivi e piani di qualità del prodotto, inclusa l'identificazione degli attributi critici della qualità del prodotto.
5. **Rischi e gestione dei rischi**: rischi chiave che possono impattare la qualità del prodotto.

STANDARD

E' una base di confronto.

Può essere:

- DE FACTO: emerge dal mercato
- DE JURE: stabilito per legge
- OPEN: standard disponibile al pubblico
- PROPRIETARY: controllato da una licenza in possesso da un'organizzazione
- PRODUCT: riguarda un prodotto
- PROCESS: riguarda un processo

Quality POLICY: un set di obiettivi generali relativi alla gestione della qualità all'interno dell'azienda.

I due processi principali, che avvengono durante TUTTO IL CICLO DI VITA, sono:

- quality **ASSURANCE**: processo che si occupa di dare delle linee guida sull'organizzazione aziendale da seguire per implementare i requisiti di qualità contenuti nelle policy
COSA: si controllano sia software che processi.
QUANDO: lungo tutto il ciclo di vita del processo
- quality **CONTROL**: processo più specifico che si occupa di controllare la misurazione delle singole valutazioni per mantenere la qualità (come implemento le policy), ovvero verificare che la qualità sia stata rispettata.
 - **PLAN**: documento nel quale definisco quali operazioni di controllo qualità vanno fatte e secondo quale standard.
 - **REVIEW**: è un'azione specifica del team dedicato alla qualità che produce un report di riepilogo del review.
 - Un gruppo di lavoro esamina il software per verificare la qualità e arrivano a dire ok il prodotto è conforme/non conforme.
 - Durante le review si fanno dei check e nel caso in cui il controllo dia esito negativo si propongono azioni correttive.
 - Ogni volta che faccio una review, tutto va documentato e registrato.

ISO 9000 è una famiglia di standard (9001,9002,9003) che mira a fornire alle aziende delle linee guida su come effettuare una gestione della qualità

Coprono le aziende che producono qualcosa, ma a livelli diversi:

- 9001: copre tutte le fasi del ciclo di vita.
- 9002: copre la fase di produzione e installazione.
- 9003: copre la fase di testing ed ispezione.

ISO 9000-3: linee guida all'applicazione di iso 9001 allo sviluppo e mantenimento del software.

Per ogni attività del ciclo di vita (indipendentemente dal modello utilizzato) specifica come i problemi di qualità vadano affrontati.

Per ricevere la certificazione 9000 bisogna chiamare delle società di consulenza specializzate (riconosciute da ISO) che vengono a verificare se operi in modo conforme, in tal caso ti forniscono un attestato di qualità.

Il *MANUALE DELLA QUALITA'* è prodotto ai fini della certificazione ed è specifico dell'azienda, descrive come essa è organizzata.

Software quality: vado a verificare delle caratteristiche (es. safety, security, testability, understandability, ecc.) usando delle metriche nella verifica e dei modelli verso cui fare riferimento.

Una caratteristica contiene più attributi.

Tipi di **caratteristiche**:

- *ESTERNE*: valutate durante l'utilizzo del sw.
- *INTERNE*: valutate facendo l'ispezione delle funzionalità del sw, durante la fase di sviluppo.
- *IN USO*: misurano gli effetti dell'utilizzo del sw in un contesto specifico.

Per ogni attributo abbiamo: definizione, formula (matematica per il calcolo), interpretazione ed utilizzo.

ISO 12207 è uno standard per il LIFE CYCLE, classifica i processi sw in:

- PRIMARI
- DI SUPPORTO
- GENERICI

La gestione della qualità tradizionale è un processo formale che si basa su un'ampia documentazione sui test e la convalida dei sistemi e su come i processi debbano essere eseguiti. A questo proposito, essa è diametralmente opposta allo sviluppo agile, dove l'obiettivo è impiegare il minor tempo possibile per scrivere documenti e formalizzare le modalità di sviluppo software.

Qualità del software

Spesso è impossibile arrivare a un giudizio oggettivo sul fatto che un sistema software soddisfi oppure no la sua specifica.

- È difficile scrivere specifiche del software chiare e complete.
- Le specifiche, di solito, integrano i requisiti definiti da varie classi di stakeholder, e quindi, inevitabilmente sono un compromesso.
- È impossibile misurare direttamente alcune caratteristiche della qualità (ad esempio la manutenibilità).

A causa di questi problemi, **il giudizio sulla qualità del software è un processo soggettivo**. La decisione del team di gestione della qualità per decidere se è stato raggiunto un livello di qualità accettabile richiede che si risponda ad alcune domande sulle caratteristiche del sistema, come ad esempio:

- Il software è stato opportunamente provato? sono stati implementati tutti i requisiti?
- Il software è sufficientemente affidabile da poter essere utilizzato?
- Le prestazioni del software sono accettabili per il normale utilizzo?
- Sono stati seguiti gli standard di programmazione e documentazione durante lo sviluppo?

Attributi di qualità del software sono: sicurezza, comprensibilità, portabilità, protezione, testabilità, utilizzabilità, affidabilità, adattabilità, riutilizzabilità, resilienza, modularità, efficienza, robustezza, complessità, facilità di apprendimento, facilità d'uso, accessibilità, etc.

La progettazione del software è un processo creativo, quindi l'influenza delle capacità delle esperienze delle persone coinvolte è significativa. Indubbiamente, il processo di sviluppo utilizzato ha una significativa influenza sulla qualità del software, ed è più probabile che i buoni processi portino a un software di buona qualità. La gestione e il miglioramento della qualità dei processi può ridurre il numero di difetti nel software. Tuttavia, è difficile misurare attributi di qualità come affidabilità e manutenibilità senza provare il software a lungo.

Standard

Gli standard del software svolgono un ruolo importante nella gestione della qualità del software. Come detto in precedenza, una parte importante della garanzia della qualità è la definizione o selezione degli standard da applicare al processo di sviluppo del software o a un prodotto software.

Gli standard del software sono importanti per tre motivi:

1. Gli standard racchiudono la conoscenza di ciò che è più prezioso e appropriato per una società. Questa conoscenza spesso è acquisita soltanto dopo una lunga serie di prove ed errori; includerla in uno standard aiuta la società a sfruttare questa esperienza passata evitando così di commettere gli stessi errori.
2. Gli standard forniscono un ambiente idoneo per definire il significato di qualità in un particolare contesto. Come detto in precedenza, la qualità del software è soggettiva, e tramite gli standard, è possibile stabilire una base comune per stabilire se è stato raggiunto il livello qualitativo desiderato.
3. Gli standard favoriscono la continuità quando il lavoro svolto da una persona viene continuato da un'altra persona. Essi assicurano che tutti gli ingegneri di una società adottino le stesse tecniche.

Due tipologie di standard per l'ingegneria del software possono essere definiti e sviluppati nella gestione della qualità del software.

- **Standard di prodotto:** si applicano al prodotto software che si sta sviluppando. Includono gli standard per i documenti, come la struttura del documento dei requisiti, gli standard per la documentazione, come le intestazioni dei commenti nella definizione delle classi di oggetti, e gli standard per la codifica, che definiscono come utilizzare un linguaggio di programmazione.
- **Standard di processo:** definiscono i processi da seguire durante lo sviluppo del software. Specificano le buone pratiche da seguire nel processo di sviluppo. Possono includere le definizioni dei processi di specifica, progettazione e convalida del software, la descrizione degli strumenti di supporto e dei documenti che dovrebbero essere scritti durante l'esecuzione di questi processi.

Esempi di standard di prodotto e standard di processo:

Standard di prodotto	Standard di processo
Modulo di revisione del progetto	Procedura di revisione del progetto
Struttura del documento dei requisiti	Invio di nuovo codice per la costruzione del sistema
Formato delle intestazioni dei metodi	Processo di release delle versioni
Stile di programmazione Java	Processo di approvazione del piano del progetto
Formato del piano del progetto	Processo di controllo delle modifiche
Modulo di richiesta delle modifiche	Processo di registrazione dei test

Gli standard devono fornire valore, nella forma di una maggiore qualità del prodotto. Non ha senso definire standard, che sono costosi in termini di tempo e di impegno, che portino miglioramenti solo marginali della qualità.

Gli standard dell'ingegneria del software che sono utilizzati all'interno di una società, di solito, vengono ricavati da standard nazionali e internazionali più generali.

Revisioni e ispezioni

Le revisioni e le ispezioni sono attività della garanzia della qualità che controllano la qualità delle consegne dei prodotti. Questo richiede il controllo del software, la sua documentazione e la registrazione dei processi per scoprire errori e omissioni, come pure le violazioni degli standard. Come detto nel capitolo 8, le revisioni e le ispezioni sono utilizzate durante i test dei programmi come parte del processo generale di verifica e convalida del software.

Durante una revisione, molte persone esaminano il software e la documentazione associata, cercando potenziali problemi e discordanze con gli standard. Le revisioni della qualità si basano sui documenti che sono stati prodotti durante il processo di sviluppo del software. Oltre alle specifiche del software, anche i progetti, il codice, i modelli, i piani di test, le procedure di gestione della configurazione, gli standard dei processi e i manuali degli utenti dovrebbero essere revisionati. La revisione della qualità dovrebbe anche controllare la coerenza e la completezza dei documenti e del codice e, se sono stati definiti gli standard di qualità, verificare che questi siano stati applicati correttamente. Le revisioni non servono solo a verificare la conformità agli standard, servono anche a scoprire problemi e omissioni nella documentazione del software e dei progetti.

Lo scopo delle revisioni e ispezioni è migliorare la qualità del software.

Sebbene ci siano molte varianti nei dettagli delle revisioni, i processi di revisione possono essere strutturati in tre fasi:

1. **Attività di prerevisione:** sono le attività che sono essenziali per la riuscita della revisione. Tipicamente riguardano la pianificazione e la preparazione della revisione: scelta del team di revisione, definizione dei tempi e luoghi della revisione, distribuzione dei documenti da rivedere.
2. **Riunione di revisione:** viene eseguita la revisione.
3. **Attività di post-revisione:** i problemi che sono emersi durante la riunione di revisione devono essere risolti.

Le ispezioni dei programmi sono revisioni nelle quali i membri del team collaborano per trovare i bug nel codice che stanno sviluppando. Le revisioni possono essere parte dei processi di verifica e convalida. Sono complementari ai test, in quanto non richiedono che il programma sia eseguito (sono test statici e non dinamici).

Gestione qualità e sviluppo agile

I metodi agili dell'ingegneria del software si focalizzano sullo sviluppo del codice; minimizzano la documentazione e i processi che non sono direttamente correlati con lo sviluppo del codice ed enfatizzano l'importanza delle comunicazioni informali tra i membri, anziché le comunicazione basate su documenti di progettazione formali. Qualità, nello sviluppo agile, significa qualità del codice; pratiche quali il refactoring e lo sviluppo guidato da test sono utilizzate per garantire che venga prodotto software di qualità.

La gestione della qualità nello sviluppo agile è **informale**, anziché basarsi sulla documentazione. Si affida a una cultura della qualità, dove tutti i membri del team si sentono responsabili della qualità del software. Nello sviluppo agile, la gestione della qualità si basa su buone pratiche condivise, anziché su documenti formali. Alcuni esempi di buone pratiche sono elencati qui di seguito:

- **Verifica prima della consegna:** i programmatore hanno la responsabilità di organizzare le revisioni del loro codice con altri membri del team di sviluppo prima che il codice sia inserito nel sistema.
- **Non spezzare mai la compilazione:** non è ammesso che ciascun membro del team di sviluppo accetti un codice che provochi il fallimento dell'intero sistema software. Ogni membro deve verificare le modifiche del suo codice rispetto all'intero sistema ed essere certo che tutto il codice operi come previsto.
- **Correggere gli errori quando vengono scoperti:** il codice del sistema appartiene al team, non ai singoli membri. Quindi se un programmatore scopre un errore o un'anomalia, deve risolvere questi problemi, anziché segnalarli allo sviluppatore originario.

I processi agili raramente usano processi formali di revisione o ispezione. L'approccio informale alla gestione della qualità adottato nei metodi agili è particolarmente efficace nello sviluppo di prodotti software dove la società che sviluppa il software controlla anche la sua specifica. Tuttavia, se si sta sviluppando un grande sistema per un cliente esterno, gli approcci agili con minima gestione della qualità (scarsa documentazione) potrebbero essere impraticabili.

Misure software

Le misure del software riguardano la quantificazione di qualche attributo di un sistema software, come la complessità o l'affidabilità. Confrontando i valori misurati tra loro e con gli standard che si applicano in una società, è possibile trarre delle conclusioni sulla qualità del software o valutare l'efficienza dei processi software, degli strumenti e dei metodi adottati.

L'obiettivo a lungo termine delle misure del software è utilizzare i risultati delle misure per giudicare la qualità del software. Teoricamente, un sistema potrebbe essere valutato utilizzando un range di metriche per misurare i suoi attributi. Dalle misure fatte si potrebbe dedurre un valore della qualità

del sistema. Tuttavia, il giudizio automatizzato della qualità è improbabile che diventi una realtà nel prossimo futuro.

Una **metrica** del software è una caratteristica del software, della documentazione, del sistema o del processo di sviluppo che può essere effettivamente misurata. Esempi di metriche sono: la dimensione di un prodotto espressa in righe di codice, numero di giorni-persona richiesti per sviluppare il sistema etc.

Le metriche del software possono essere metriche di **controllo** o di **previsione**.

- **metriche di controllo**: sono a supporto della gestione dei processi. Di solito sono associate ai processi software. Esempi di metriche di controllo sono lo sforzo medio e il tempo richiesto per riparare i difetti rilevati.
- **metriche di previsione**: aiutano a prevedere le caratteristiche del software. Sono associate al software stesso. Esempi di queste metriche sono la complessità ciclomatica di un modulo, la lunghezza media degli identificatori di un programma etc.

Ci sono tre tipi di **metriche di processo** che possono essere utilizzate:

- Il tempo impiegato da un particolare processo per essere completato;
- Le risorse richieste da un particolare processo;
- Il numero di occorrenze di un particolare evento.

Le metriche di controllo e previsione influiscono entrambe sulle decisioni di gestione.

Le misure di un sistema software possono essere utilizzate in due modi:

- Assegnare un valore agli attributi di qualità di un sistema;
- Identificare i componenti del sistema la cui qualità è inferiore allo standard.

È difficile fare delle misure dirette di molti degli attributi della qualità del software. Gli attributi di qualità come manutenibilità, comprensibilità e utilizzabilità sono attributi esterni che sono correlati al modo in cui gli utenti sperimentano il software. Questi attributi sono influenzati da fattori soggettivi, come esperienza e conoscenza, e quindi non possono essere misurati in modo obiettivo. Per valutare questi attributi occorre misurare qualche attributo interno del software e supporre che questi abbiano una correlazione con le caratteristiche di qualità che interessano.

Metriche prodotto

Le metriche di prodotto sono metriche di previsione che vengono utilizzate per quantificare gli attributi interni di un sistema software. Esempi di metriche di prodotto sono la dimensione del sistema, misurata in righe di codice, e il numero di metodi associati a ciascuna classe di oggetti.

Le metriche di prodotto si suddividono in due classi:

- **metriche dinamiche**: sono raccolte dalle misure effettuate su un progetto in esecuzione. Queste metriche possono essere raccolte durante i test del sistema o dopo che il sistema è stato consegnato agli utenti. Un esempio può essere il numero di bug report.
- **metriche statiche**: sono raccolte dalle misure eseguite sulle rappresentazioni del sistema, come il progetto, il programma o la documentazione.

Esercizi di teoria

Domande esami 2018:

❖ **Cos'è e a cosa serve uno standard?**

Uno standard è una regola o un riferimento di base per il confronto che viene utilizzata per valutare le dimensioni, il contenuto o la qualità di un oggetto o un'attività. Nell'ambito dell'ingegneria del software, gli standard del software racchiudono la conoscenza e le best-practice da attuare durante tutta la fase di sviluppo del software al fine di evitare errori commessi in passato. Gli standard forniscono anche uno strumento per il controllo e la definizione di qualità del prodotto software, in quanto stabiliscono una linea comune di metriche e fattori da tenere in considerazione.

❖ **Quali aspetti legati al software sono oggetto degli standard illustrati nel corso?**

L'aspetto principale di un software che viene influenzato dagli standard è la qualità. Il processo di gestione della qualità spesso si affida all'utilizzo di standard per verificare appunto, il livello di qualità del prodotto software e dei processi di sviluppo. Avendo gli standard un ruolo importante nella gestione della qualità, di conseguenza hanno anche un ruolo importante in tutti quegli aspetti ad essa legati, come: sicurezza, affidabilità, usabilità, protezione, robustezza, efficienza etc.

❖ **I requisiti di un sistema software sono spesso incompleti, mal capiti e/o mutevoli; e questa situazione si ripercuote ovviamente sulle specifiche del sistema. Si commenti questa affermazione.**

A livello teorico, in un processo software esiste sempre una fase ben definita di ingegneria dei requisiti (ricerca, analisi, documentazione e verifica dei requisiti) che porta alla creazione di un documento formale che include requisiti dell'utente e del sistema che rappresenta ciò che il team di sviluppo dovrà produrre. In realtà però, il processo di ingegneria dei requisiti è complesso e pieno di difficoltà, a livello pratico quindi, più che essere un processo ben definito in una sola fase è un processo iterativo (nei processi plan-driven è comunque distinguibile una macro-fase di ingegneria dei requisiti, ma questo processo continua comunque durante lo sviluppo del software; nei processi agili invece questa fase è per principio di natura iterativa). Le principali difficoltà nascono dal fatto che i requisiti non sono sempre chiari sin dalla prima fase di ricerca e analisi (i vari stakeholder possono avere idee poco chiare; non sanno cosa vogliono e cosa può offrire loro un prodotto software; vari stakeholder possono avere idee contrastanti; a volte le idee sono chiare ma ci sono dei fraintendimenti tra stakeholder e ingegneri dei requisiti, che portano alla errata comprensione dei requisiti etc.). Tutte queste difficoltà hanno come conseguenza che i requisiti di un sistema variano nel tempo, sia perché durante le prime fasi di progettazione si acquisisce maggiore conoscenza del prodotto da creare, sia per le difficoltà elencate precedentemente; questa continua possibilità di mutazione e variazione dei requisiti fa sì che l'ingegneria dei requisiti sia un processo iterativo, in cui le fasi di ricerca, analisi, documentazione e verifica si ripetano e si intreccino tra loro, con un continuo feedback reciproco. Spesso capita che le operazioni di ing. dei requisiti vengano svolte anche durante la fase di progettazione e di sviluppo (ad es. negli approcci agili allo sviluppo del software). La fase che gestisce le modifiche dei requisiti è detta gestione dei requisiti.

❖ **Una conseguenza di quanto riportato nella domanda precedente è che sono stati proposti diversi modelli dei processi software che cercano di ovviare a tali criticità. Si illustrino caratteristiche e motivazioni dei principali modelli.**

I principali modelli dei processi software che cercano di ovviare a queste criticità sono quelli basati su un approccio agile (sviluppo rapido del software; ad esempio l'extreme programming). La principale caratteristica legata ai requisiti del sistema che caratterizza i modelli agili è che i processi di ingegneria del software, progettazione e implementazione sono intrecciati; di conseguenza NON c'è una macro-fase iniziale di ingegneria dei requisiti; è assente una specifica di sistema dettagliata e i documenti dei progetti sono minimi. Il software viene sviluppato con un approccio incrementale e

gli utenti (stakeholder finali) sono spesso resi partecipi nella specifica e valutazione di ogni incremento. Questa forte collaborazione fa sì che si crei un rapporto informale che però permette di controllare meglio la natura mutevole dei requisiti. I metodi agili ritengono che "la risposta al cambiamento" sia più importante di seguire un piano ben definito. Per gestire la mutazione dei requisiti, i metodi agili hanno sviluppato la tecnica delle storie utente, che permette di integrare la deduzione dei requisiti con lo sviluppo. Una storia utente è uno scenario d'uso in cui potrebbe trovarsi un utente finale; queste storie utente sono la rappresentazione di ciò che il sistema deve fare e di conseguenza sono analoghi ai requisiti.

- ❖ **Si illustri la strutturazione e gli scopi del processo di progettazione architetturale, collocandolo nell'ambito del ciclo di vita del software.**

La progettazione architetturale si occupa dell'organizzazione di un sistema software e della progettazione della sua struttura complessiva. Nell'ambito di ciclo di vita del software, la progettazione architetturale si colloca tra ingegneria dei requisiti e la fase di progettazione del software, in quanto identifica i principali componenti di un sistema e le loro relazioni, analizzando i requisiti ottenuti. L'output del processo di progettazione architetturale è un modello architetturale che descrive come il sistema sarà organizzato in funzione dei componenti principali. La scelta di come strutturare il software è importante perché influenzera le prestazioni, la robustezza, la manutenibilità etc. del sistema; infatti l'architettura ha un'influenza predominante sulle caratteristiche non funzionali del sistema; mentre i singoli componenti su quelle funzionali (in quanto implementano i singoli servizi; quindi i singoli requisiti funzionali). Il processo di progettazione architetturale dipende fortemente dal sistema che si sta sviluppando e dall'esperienza del team; esso è più una serie di decisioni da prendere che una serie di attività ben definite da seguire. L'architettura di un sistema software può basarsi su un particolare schema o stile architetturale. Uno schema architetturale è una descrizione dell'organizzazione del sistema, per esempio un'organizzazione client-server o un'architettura a strati (esempi di schemi architetturali: a strati, client-server, MVC, repository, orientato ai servizi, per sistemi distribuiti). Per scomporre le unità strutturali del sistema, bisogna scegliere la strategia di scomposizione dei componenti in sottocomponenti; infine, nel processo di modellazione del controllo, occorre sviluppare un modello generale delle relazioni di controllo tra le varie parti del sistema e decidere come controllare l'esecuzione dei componenti. A causa della stretta relazione tra le caratteristiche non funzionali del sistema e l'architettura hardware sottostante, la scelta dello schema architetturale e della struttura dipende pesantemente dai requisiti non funzionali del sistema.

- ❖ **Si illustrino caratteristiche, differenze, ambiti applicativi, pro e contro dei due modelli repository e client server.**

L'architettura client-server è organizzata come un insieme di servizi e server associati e di client che accedono e utilizzano tali servizi. Le architetture client-server sono tipicamente utili negli ambiti di sistemi distribuiti (anche se possono essere implementate architetture client-server su una singola macchina). Questa architettura permette di separare e rendere indipendenti i vari componenti del sistema, di conseguenza questa architettura ha un'elevata manutenibilità.

L'architettura a repository concettualizza una struttura in cui più componenti interattivi condividono i dati, salvati tipicamente in un database centrale (repository). Questo fa sì che i componenti non interagiscano tra loro ma solo mediante la repository. I componenti sono quindi indipendenti, e questo risulta essere un vantaggio. Il punto debole di un'architettura a repository è la repository stessa, un guasto della repository renderebbe inutilizzabile l'intero sistema.

❖ Si inquadri e si illustri l'utilizzo della tecnica equivalence partitioning nell'ambito del black-box testing. Quali le motivazioni che giustificano il suo utilizzo e quali i vantaggi?

Il test black-box è un metodo di test del software che esamina la funzionalità di un'applicazione senza scrutare nelle sue strutture o funzionamenti interni (tipico dei sistemi legacy).

Il partizionamento di equivalenza o il partizionamento di classe di equivalenza (ECP) è una tecnica di test del software che divide i dati di input di un'unità software in partizioni di dati equivalenti da cui è possibile derivare casi di test. In linea di principio, i casi di test sono progettati per coprire ogni partizione almeno una volta.

❖ Si illustri il concetto di dependability del software e le sue quattro principali dimensioni.

La fidatezza di un sistema software è una proprietà che rispecchia il livello di fiducia che l'utente può riporre in esso. La fidatezza (dependability) è un termine che fa riferimento e rappresenta l'unione di quattro principali caratteristiche di un software, ovvero:

- Disponibilità: indica la probabilità che un sistema sia attivo e in grado di fornire i servizi utili per il quale è stato sviluppato.
- Affidabilità: indica la probabilità che il sistema in un determinato periodo di tempo fornisca correttamente i servizi secondo le aspettative dell'utente.
- Sicurezza: indica la stima delle probabilità che il sistema possa causare danni a persone o all'ambiente.
- Protezione: indica la stima delle probabilità che il sistema possa resistere a intrusioni accidentali o deliberate.

Un sistema si dice quindi fidato (elevata fidatezza) se ha buone qualità nelle caratteristiche appena elencate. La fidatezza è una caratteristica fondamentale di tutti i sistemi, ma soprattutto di quelli critici: sistemi dove i guasti hanno un elevato rischio di causare danni a persone, danni economici e ambientali etc. (vedi software di pilotaggio).

❖ Si definisca e si illustri una metrica per misurare l'availability (disponibilità).

Per misurare le caratteristiche della fidatezza sopra elencate, tipicamente vengono utilizzate delle metriche: una metrica del software è una caratteristica del software può essere effettivamente misurata. Le metriche che spesso vengono usate per misurare la disponibilità del software sono:

- Probabilità di fallimento su richiesta (POFOD): probabilità che la richiesta di un servizio provochi il suo fallimento.
- Tasso di occorrenza dei fallimenti (ROCOF): numero di fallimenti del sistema che statisticamente si verificano in un certo lasso di tempo.
- Tempo medio al fallimento (MTTF): tempo medio tra due fallimenti del sistema consecutivi.
- Disponibilità (AVAIL): probabilità che un sistema sarà operativo quando viene richiesto un servizio.

❖ Cosa significa validazione del software? Come si fa?

La validazione (o convalida) fa parte di un processo più generale di verifica e convalida del software (V&V, Verification and Validation). Lo scopo della validazione del software è garantire che esso rispetti le attese del cliente. Il suo scopo va ben oltre il semplice controllo di conformità alla specifica per dimostrare che il software fa ciò che il cliente ha richiesto. La convalida è essenziale perché, le definizioni dei requisiti non sempre riflettono i reali desideri o necessità dei clienti. L'obiettivo ultimo dei processi di verifica e convalida è stabilire, con un buon grado di confidenza, che il software è adatto al suo scopo. Ciò significa che il sistema deve essere adatto all'uso cui è destinato. Il principale approccio per eseguire la validazione del software è il testing: l'obiettivo dei test del software è dimostrare che un programma svolge i compiti per i quali è stato sviluppato e identificare eventuali

errori. I test hanno l'obiettivo di dimostrare che i requisiti funzionali e non funzionali siano rispettati. I test di convalida possono anche richiedere ispezioni e revisioni statiche del software. Anche il controllo della qualità fa parte della convalida del software.

❖ **Cosa la differenzia dalla verifica del software e cosa la accomuna?**

Come accennato nella risposta precedente, la verifica e la validazione (convalida) fanno parte di un processo più generale di verifica e convalida (V&V) del software. Il ruolo dei processi di verifica e convalida è essenzialmente controllare che il software che si sta sviluppando soddisfi le sue specifiche e offra le funzionalità richieste da chi ha acquistato il software. La differenza sostanziale tra verifica e convalida è che:

- la verifica è il processo che controlla se il software soddisfa i requisiti funzionali e non funzionali. Fornisce l'obiettiva conferma che il software soddisfa le specifiche.
- la validazione è un processo più generale che mira a garantire che il software rispetti le attese del cliente. Fornisce la conferma che il software rispetta i requisiti e aspettative del cliente.

Sono due fasi che in comune hanno l'obiettivo ultimo, ovvero: stabilire con un buon grado di confidenza, che il software è adatto al suo scopo. Ciò significa che il sistema deve essere adatto all'uso cui è destinato.

❖ **Si illustri lo scopo della scomposizione modulare.**

Dopo aver scelto l'organizzazione generale del sistema, si deve decidere l'approccio da usare per la scomposizione dei sottosistemi in moduli. Un modulo è solitamente un componente di un sistema che fornisce uno o più servizi ad altri moduli, fa uso dei servizi forniti da altri moduli, e non è normalmente considerato un sistema indipendente. Tipicamente i moduli sono formati da una serie di componenti più piccoli (che sono indipendenti). Il principale scopo della scomposizione modular è l'aumento della manutenibilità del sistema e della riusabilità

❖ **Si illustrino i due concetti di coesione e di accoppiamento. Si evidenzi anche perché è importante progettare moduli con forte coesione e basso accoppiamento.**

La coesione è una misura del livello di correlazione tra diverse funzionalità presenti all'interno di un modulo, ossia del suo livello di omogeneità funzionale. L'accoppiamento è una misura delle connessioni (dipendenze) tra due componenti del sistema. È importante progettare moduli con forte coesione perché questo indica un elevato livello di manutenibilità del sistema, ogni modifica è più rapida ed economica se svolta in un componente unico, invece che andare a modificare più componenti del sistema con il rischio di dover eseguire molte più modifiche. Un buon metodo per avere alta coesione nei moduli è applicare i principi solidi (principalmente il SRP). Avere basso accoppiamento indica che i moduli del sistema sono prevalentemente indipendenti e hanno scarse relazioni con altri moduli, questo basso livello di accoppiamento fa sì che una modifica non vada a impattare su altri moduli. Decentralizzando lo stato e facendo comunicare i moduli tramite messaggi (e non condividendo variabili) si raggiunge un basso livello di accoppiamento (loose coupling).

❖ **Si illustri il concetto di requisiti software.**

I requisiti di un sistema sono la descrizione dei servizi che il sistema deve fornire e dei suoi vincoli operativi. Il processo di riceva, analisi, documentazione e verifica di questi servizi e vincoli è chiamato ingegneria dei requisiti. I requisiti possono essere suddivisi in requisiti utente (requisiti di alto livello che dichiarano quali servizi il sistema deve fornire) e requisiti di sistema (forniscono la descrizione dettagliata delle funzioni, dei servizi e dei vincoli operativi, cosa effettivamente deve essere implementato).

- ❖ Cosa sono rispettivamente la definizione dei requisiti e la specificazione dei requisiti? Quali le differenze? Come se ne rappresentano rispettivamente i risultati?

La definizione dei requisiti e la specificazione dei requisiti sono differenti fasi del processo di ingegneria dei requisiti, inoltre differiscono nel livello di dettaglio dei "requisiti" raccolti. La definizione dei requisiti consiste nella scoperta e analisi dei requisiti utente e requisiti di sistema (descritti nella risposta precedente) tramite l'interazione con gli stakeholder, mentre la specificazione dei requisiti consiste nella produzione (conversione dei requisiti in un formato standard – le specifiche del sistema) di un documento dei requisiti, con descrizioni tecniche/dettagliate del software e di cosa deve fare, che verrà poi utilizzato dagli ingegneri come punto di partenza per la fase di progettazione e implementazione e come documento formale rappresentante i requisiti utenti e del sistema.

- ❖ Si illustrino gli scopi dei due processi fondamentali della gestione della qualità del software, ossia
 - (1) l'assicurazione della qualità (software quality assurance)
 - (2) il controllo di qualità (software quality control).

La gestione della qualità del software (QM – Quality Management) si occupa di garantire che i sistemi software sviluppati siano conformi agli scopi prestabiliti, ovvero che i sistemi soddisfino le esigenze dei loro utenti, che i processi siano eseguiti in modo efficiente e affidabile e che il software sia consegnato entro i limiti di tempo e budget previsti. La garanzia (assicurazione / assurance) della qualità è la definizione di processi e standard che dovrebbero essere poi seguiti al fine di produrre prodotti di alta qualità. Il controllo della qualità è l'applicazione di questi processi e standard per eliminare quei prodotti che non hanno raggiunto i livelli di qualità voluti. Il processo di assicurazione della qualità è più generale ed è relativo a come ci si organizza per raggiungere i requisiti di qualità, mentre il controllo è più specifico ed è relativo a come si procede nelle singole valutazioni necessarie a misurare la qualità e nelle esecuzioni di processi di controllo.

- ❖ Cos'è un sistema di qualità (quality system)?

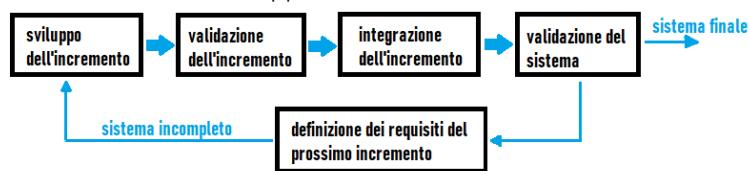
Un sistema di qualità è un sistema dove la qualità è un concetto critico. Un quality system è un sistema che segue una serie di standard e normative al fine di ottenere elevati livelli di qualità.

- ❖ Cosa è il manuale della qualità (quality manual)?

- ❖ Si fornisca un modello dei processi relativamente allo sviluppo incrementale (consegna incrementale o incremental software development): si utilizzi un grafico per illustrare struttura e legami dei vari processi e si descrivano gli scopi di ciascun processo.

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, finché non si ottiene il sistema richiesto. Le attività di specifica, sviluppo e convalida sono intrecciate anziché separate, con molti feedback veloci tra le varie attività. Questo approccio può essere plan-driven, agile, o una combinazione dei due. In un approccio plan-driven gli incrementi sono pianificati precedentemente, nell'approccio agile invece dipendono dalla situazione generale di sviluppo. Questo approccio è pensato per quei software in cui i requisiti e le specifiche di sistema variano costantemente. Ciascun incremento (versione) del sistema apporta qualche funzionalità aggiuntiva richiesta dall'utente.

Un modello dei processi che utilizza un approccio incrementale è di natura ciclica (ad esempio l'XP):



❖ **Cos'è un incremento e come ad esso vengono assegnati i requisiti?**

Quando si parla di incrementi, stiamo parlando dei modelli di sviluppo software basati sullo sviluppo incrementale. Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, attuando un processo iterativo fino a quando non si ottiene il sistema richiesto. Un incremento è quindi l'aggiunta di una nuova funzionalità alla versione precedente, che viene poi testata e convalidata. I requisiti, negli approcci incrementali vengono scelti e discussi con il cliente (negli approcci agili) o pianificati in precedenza (negli approcci plan-driven) prima della progettazione e sviluppo dell'incremento stesso.

❖ **Che dimensioni può avere un incremento in termini di tempo necessario per il suo sviluppo?**

Tipicamente lo sviluppo incrementale punta a sviluppare il software in piccoli incrementi, implementabili in 1-2 settimane. Questo perché punta a uno sviluppo rapido del software (metodi agili).

Domande esami 2017:

❖ **Quali sono i principali elementi che caratterizzano i metodi agili?**

I metodi agili (o metodi di sviluppo agile del software) nascono dall'esigenza di produrre software utile e di qualità velocemente, in una realtà commerciale competitiva dove i tempi di consegna sono un fattore fondamentale. I processi di sviluppo sono caratterizzati da fasi di specifica, progettazione e implementazione intrecciate (a differenza dei processi plan-driven dove sono fortemente sequenziali). Con un approccio agile, un sistema viene sviluppato tipicamente seguendo un approccio di sviluppo incrementale, in cui gli utenti finali (in generale gli stakeholder) sono coinvolti nella specifica e valutazione di ciascun incremento. I principi dei metodi agili possono essere riassunti dai punti elencati dal manifesto dello sviluppo agile del software:

- individui e interazioni sono più importanti dei processi e strumenti;
- il software funzionante è più importante della documentazione dettagliata;
- la collaborazione con il cliente è più importante della negoziazione dei contratti;
- rispondere al cambiamento (in riferimento ai requisiti che mutano velocemente) è più importante che seguire un piano.

I metodi agili sono molto efficaci nello sviluppo di software di piccole/medie dimensioni e dove c'è chiara necessità di gestire numerose mutazioni dei requisiti del sistema. I metodi agili sono supportati da una serie di tecniche e metodologie, tra queste, le più importanti sono:

- proprietà collettiva: il software è sotto la responsabilità di ogni sviluppatore della software house.
- integrazione continua: appena un task è stato implementato, va integrato e validato nel sistema.
- pianificazione incrementale: invece che seguire un approccio guidato da piani, viene eseguita una pianificazione dei task da implementare per ciascuna release del sistema.
- cliente on-site: il cliente deve essere sempre disponibile per mediare e aiutare nello sviluppo del sistema.
- utilizzo della programmazione a coppie
- refactoring
- progettazione semplice, volta a soddisfare i requisiti correnti, niente di più
- piccole release
- sviluppo con test iniziali
- user stories: i requisiti del software cambiano sempre. Per gestire questi cambiamenti, i metodi agili non hanno un'apposita attività di ingegneria dei requisiti, ma integrano la deduzione dei requisiti con lo sviluppo. Per semplificare questo approccio, fu sviluppata l'idea delle storie utente, dove una storia è uno scenario d'uso in cui potrebbe trovarsi un utente finale del sistema.

❖ Come si può descrivere l'Extreme Programming in termini di Incremental development?

L'extreme programming incarna totalmente l'essenza dello sviluppo rapido del software, attuando praticamente in totalità le metodologie e tecniche proposte dallo sviluppo agile. L'XP segue quindi un approccio incrementale: nell'XP i requisiti del sistema sono rappresentati su story card (user stories) e per ciascun incremento vengono determinate quali task (derivati dalle user stories) implementare per ciascun incremento. Appena un task è implementato esso viene integrato nel sistema (tecnica dell'integrazione continua). Dopo ogni integrazione di codice nel sistema, esso deve superare tutti i test.

❖ Si illustrino le due principali tecniche di verifica del software.

La verifica fa parte di un processo più generale di "verifica e convalida" del software (V&V, Verification and Validation). Il processo di V&V ha lo scopo di controllare che il software sviluppato offra le funzionalità richieste e che soddisfi le aspettative del cliente. La fase di verifica è il processo che controlla che il software soddisfi tutti i requisiti funzionali e non funzionali; mira a fornire quindi l'obiettiva conferma che il software è stato realizzato correttamente e che rispetta tutte le specifiche. Le principali tecniche di verifica del software sono il testing e le ispezioni (e le revisioni).

Il testing mira a dimostrare che il sistema funziona correttamente utilizzando una serie di test case che riflettono l'uso previsto del sistema (test di convalida) e scovare difetti (bug) nel software (test dei difetti). Esistono tre stadi di test:

- Test dello sviluppo: include tutte le attività di test che sono svolte dal team di sviluppo di un sistema software. Si suddivide in: test delle unità, dei componenti e del sistema.
- Test della release: è il processo che testa una particolare release di un sistema che dovrà essere utilizzata all'esterno del team di sviluppo (release per il cliente o per un team che sviluppa sistemi correlati).
- Test degli utenti: gli utenti testano il sistema, dando i propri input e suggerimenti (alpha/beta test, test di accettazione in cui il cliente accetta e completa l'acquisto del software).

Il testing è una tecnica di verifica dinamica in quanto richiede che il software venga eseguito.

Le ispezioni e le revisioni sono tecniche di verifica statica in quanto non è richiesto che il software venga eseguito. Consistono nell'analisi e controllo i requisiti del sistema, gli schemi di progettazione, il codice sorgente e i test proposti per il sistema. Hanno dei vantaggi in quanto possono scovare errori non facilmente scovabili via testing: errori di dipendenza tra classi/componenti ad esempio. Le ispezioni e revisioni inoltre possono controllare aspetti non "testabili" come: qualità del codice, facilità di manutenzione, errate scelte di design etc.

❖ Cos'è e a cosa serve il path testing?

In ingegneria del software, il path testing (test del percorso) è un metodo "white box" per la progettazione di test case. Il metodo analizza il diagramma di flusso di controllo di un programma per trovare un insieme di percorsi di esecuzione linearmente indipendenti.

❖ Cos'è la complessità ciclotomica e come può essere utile per il path testing?

La complessità ciclotomica è una metrica software utilizzata per misurare la complessità strutturale di un programma. Tale metrica misura direttamente il numero di cammini linearmente indipendenti che caratterizzano il grafo del flusso di controllo di un programma di conseguenza è utile per dare un valore concreto da utilizzare nel metodo del path testing.

❖ Cos'è e perché è importante il processo di Software Project Management?

La gestione dei progetti (project management) è una parte essenziale dell'ingegneria del software. I progetti devono essere gestiti perché l'ingegneria del software professionale è sempre soggetta a

vincoli aziendali di budget e di tempi. La buona gestione non può garantire il successo di un progetto, ma la cattiva gestione di solito ne determina il fallimento.

❖ **In cosa consistono le specifiche attività eseguite da un SW Project Manager?**

È difficile fornire una descrizione dei compiti standard di un project manager. I compiti variano notevolmente in funzione dell'azienda e del tipo di software che si sta sviluppando. Tuttavia, alcune attività di gestione dei progetti sono comuni a tutte le aziende: pianificazione dei progetti, risk management, gestione del personale e reporting. I project manager sono responsabili della pianificazione della stima e della tempistica dello sviluppo dei progetti; inoltre devono assegnare i compiti al personale. Hanno la supervisione del lavoro per garantire che esso sia svolto secondo gli standard, e tengono sotto controllo l'avanzamento del lavoro per verificare che esso venga sviluppato in tempo ed entro il budget previsto. I project manager devono definire i rischi che potrebbero influenzare un progetto, monitorare questi rischi e svolgere le azioni appropriate quando si presenta qualche problema. I project manager sono responsabili della gestione di una squadra di persone. Devono selezionare le persone della loro squadra e definire le modalità operative che possono migliorare l'efficienza del lavoro di gruppo. I project manager hanno il compito di documentare il progresso di un progetto in appositi report da inviare ai clienti e ai manager della società che sviluppa il software.

❖ **Quali sono le differenze tra i concetti di Deliverable e di Milestone?**

Sono due termini usati nell'ambito della pianificazione della progettazione. Con deliverable si intende un risultato di progetto concerto, che si può fornire a un cliente: un report, un prototipo, il software finale etc. Con milestone (in italiano pietra miliare) si indica il punto finale di un'attività di processo (il raggiungimento di un obiettivo prefissato).

❖ **Nell'ambito del concetto di Dependability, si illustrino i seguenti tre concetti e le relative relazioni: errore umano (human error), bug (errore), failure (malfunzionamento), evidenziando anche in che specifico momento si possono verificare.**

Il modello difetto-errore-fallimento si basa sul concetto che gli errori umani causano difetti nel sistema, i difetti causano errori, gli errori causano fallimenti.

- Errore umano: comportamento umano che si traduce nell'introduzione di difetti nel sistema (durante la fase di progettazione, testing, implementazione, specifica etc.)
- Difetti del sistema (bug): caratteristica di un sistema software che può portare a un errore del sistema (a run-time)
- Errore del sistema: comportamento del sistema non atteso dagli utenti (a run-time)
- Fallimento del sistema: evento che si verifica nel momento in cui il sistema non fornisce il servizio che gli utenti si aspettano (a run-time).

❖ **Si definisca il concetto di affidabilità del software e si forniscano due metriche per la sua misura.**

L'affidabilità è uno delle quattro dimensioni fondamentali della fidatezza. Se pensiamo ai sistemi software come strumenti che forniscono qualche tipo di servizio, l'affidabilità può essere informalmente definita come la capacità del sistema di fornire i risultati corretti. Più precisamente, l'affidabilità è la probabilità di un'operazione di essere svolta senza che il sistema fallisca in un determinato periodo di tempo, in un dato ambiente, per uno scopo specifico. Delle metriche per misurare l'affidabilità sono:

- Probabilità di fallimento su richiesta (POFOD): definisce la probabilità che la richiesta di un servizio provochi il fallimento del sistema.

- Tasso di occorrenza di fallimenti (ROCOF): definisce il numero di fallimenti del sistema che statisticamente si verificheranno in un certo lasso di tempo.
 - Tempo medio al fallimento (MTTF): tempo medio tra l'occorrenza di due fallimenti.
 - Disponibilità (AVAIL): probabilità che un sistema sarà operativo all'effettuazione di una richiesta di servizio.
-
- ❖ Si illustrino i passaggi (step/fasi) che caratterizzano la progettazione orientata alle funzioni (function-oriented design), evidenziando i ruoli del diagramma DFD (Data Flow Diagram) di partenza, del diagramma strutturale e delle minispec prodotte come risultato.
 - ❖ In riferimento alla domanda precedente: in quale fase del processo generale di progettazione si inseriscono queste attività?
 - ❖ In quali situazioni e per quali tipologie di sistemi di elaborazione è adatto l'approccio function-oriented design?

Domande esami 2015:

- ❖ Si fornisca una definizione dell'architettura client server, indicando anche pregi e difetti ed in quale parte della progettazione architettonica viene considerata.
- ❖ Si indichino le differenze tra le seguenti tipologie di architetture client server: thin-client, fat-client, three-tier e multi-tier.
- ❖ Si illustri il concetto di software testing, evidenziandone gli elementi fondamentali.
- ❖ Nel corso sono state esaminate numerose tecniche di testing, classificate in gruppi e differenziate secondo diversi criteri. Si spieghino i principali criteri utilizzati.

Domande esami 2014:

- ❖ Si illustri il concetto ed il ruolo degli stakeholder (portatori di interesse) nell'ambito del processo di sviluppo software.
- ❖ In riferimento alla domanda precedente: In quali fasi del ciclo di vita intervengono?
- ❖ In riferimento alla domanda precedente: Quali sono le criticità?
- ❖ Si illustri sinteticamente in cosa consistono le principali management activity (attività di gestione) nell'ambito di un progetto software?

- ❖ Cos'è e a cosa serve un activity diagram o network (rete di attività)?
- ❖ Cos'è il percorso critico (critical path)?
- ❖ Nell'ambito dell'approccio delineato dallo standard ISO 9000, si illustrino i due distinti processi di Quality Assurance (Assicurazione della Qualità) e di Quality Control (Controllo della Qualità).
- ❖ Nell'ambito del ciclo di vita del software, quando viene svolto il processo di Quality Control e come?

Domande esami 2013:

- ❖ Si illustri il processo di ingegnerizzazione dei requisiti software.
- ❖ Si illustrino le tecniche di elicazione e di analisi dei requisiti.
- ❖ Si illustri l'idea delle Tecniche Agili per lo sviluppo del SW, fornendone la principale motivazione.
- ❖ Cos'è e quali sono gli elementi principali dell'eXtreme Programming.

Domande esami 2012:

- ❖ Come e quando i 2 distinti processi di SW Quality – SQA (Assicurazione della Qualità) e di Quality Control (Controllo delle Qualità) contribuiscono in una software house alla produzione di software di qualità?
- ❖ Cos'è un processo software (software process)? Si descriva il processo di sviluppo del software basato su incrementi successivi (incremental software development). Si illustri come tale approccio rappresenti una combinazione dei due approcci evolutivo e waterfall, evidenziandone poi pregi e criticità.

Domande esami 2011:

- ❖ Si illustri il concetto di software specification (specifica software)? In che relazione è con il processo di specificazione dei requisiti? Da cosa si parte per ottenere le specifiche software e come si procede?
- ❖ Cosa significa architettura distribuita? Quali sono i principali tipi di architetture a strati (tier-architecture) e come si differenziano? Cos'è il middleware?

- ❖ Progettazione architetturale: si indichi quali fasi del ciclo di vita lo precedono e quali lo seguono?
Si illustri lo scopo dei tre sottoprocessi che lo compongono.
- ❖ Si illustri il processo di **statistical testing** utilizzato per misurare l'affidabilità.
- ❖ A cosa serve il profilo operativo (**operational profile**)?

Il profilo operativo del software riflette come questo verrà utilizzato nella pratica. Consiste in una specifica delle classi di input e nella probabilità che questi avvengano.